Received August 8, 2021, accepted August 23, 2021, date of publication August 30, 2021, date of current version September 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3108801

High-Performance Software Load Balancer for Cloud-Native Architecture

JUNG-BOK LEE, TAE-HEE YOO, EO-HYUNG LEE, BYEONG-HA HWANG, SUNG-WON AHN, AND CHOONG-HEE CHO, (Member, IEEE)
Department of Cloud, Kakao Enterprise Corporation, Seongnam 13494, South Korea

Corresponding author: Choong-Hee Cho (kay.cho@kakaoenterprise.com)

ABSTRACT Driven by increasing in the demand for cloud computing, cloud providers are constantly seeking configuration mechanisms designed to simply install a reliable and easy-to-manage cloud architecture—similar to installing an operating system on a computer using a thumb drive. Accordingly, cloud software components can be packaged into lightweight and portable containers, and then be easily deployed and managed through orchestration tools such as Kubernetes. Similarly, load balancers can also be deployed in containerized cloud environments and managed as a container, simplifying the process of scaling in or out according to the network status or amounts of incoming traffic. In this study, we implemented a containerized high-performance load balancer that distributes traffic using eBPF/XDP within the Linux kernel, which can easily be managed via Kubernetes. We compared the performance of the proposed load balancer with iptables DNAT and loopback based on the RFC2544 performance standard, and also performed tests simulating real-world traffic patterns by using IMIX traffic streams. Our experimental results indicate that the throughput performance of the proposed load balancer is considerably better than that of iptables DNAT; the difference in performance increased with decreasing packet size. The difference in performance between the loopback (representing the theoretical maximum performance limit) and the proposed load balancer was minimal.

INDEX TERMS Cloud, datacenter, direct server return, load balancer, multitenant networks.

I. INTRODUCTION

A cloud computing system [1] can be seen as composed of various IT resources including physical machines, virtual machines, etc. and various *components* such as computing applications, networking, etc. running on them. Owing to the abundance of management targets, it remains difficult to build and control cloud infrastructure in terms of cloud service providers (CSPs), and the construction of datacenters providing infrastructure as a service (IaaS) involves many complex considerations. Any human intervention required to manage cloud resources or components inevitably introduces high costs in solving operational problems, leading to difficultly in consistently restoring the managed system elements. For this reason, CSPs should carefully strategize to automate cloud deployment and operation to minimize the required human intervention. Owing to the nature of cloud computing systems, heterogeneous hypervisors inevitably comprise the operating system (OS) of the components; that is, a component runs for hypervisors across generations. If a dependency on a hypervisor where each component is executed exists, automating the deployment and operation of the components becomes difficult owing to the number of cases that must be considered. The same applies to the construction of new datacenters (or regions); additional datacenters are inevitably required for increasing numbers of users. When a CSP builds a new datacenter, significant efforts are required to operate components dependent on a specific OS if the environment in which components are run differed from that of existing datacenters. For this reason, in most clouds, components are packaged into containers to remove dependencies on specific OSs. A cloud-native architecture refers to an environment in which each component is packaged and can be deployed independently on any infrastructure. On the CSP side, we emphasize the need to incorporate a cloud-native architecture in datacenter operations. Because containerized components can be easily reproduced and ported to any infrastructure, containers have become an essential tool

The associate editor coordinating the review of this manuscript and approving it for publication was Yougan Chen.

for automated deployment and operation. For containerized components, a container orchestration tool can be used to manage the components together, as opposed to separately. In addition, container orchestration tools include features for automatic container placement and replication, container failover recovery, scaling in and out by adding or removing containers, etc. The well-known container orchestration tools [2] include Docker Swarm [3], Kubernetes [4], and Apache Mesos [5]. In particular, through Kubernetes, which specializes in container service deployment and management, cloud administrators can automate the deployment and operation of components in a cluster unit by declaring a state without directly commanding the operation. In other words, resources can be defined, and their state maintained as desired. A cluster is defined as a clustered set of nodes to run these resources. Components can be grouped according to their roles and dependencies and can be operated in multiple clusters.

Clouds configured by CSPs with multiple clusters involve concerns, with reliable and automatic deployment as well as management of multiple clusters. In this study, we refer to our cloud architecture as an installable cloud, in which multiple clusters capable of IaaS services can be easily deployed and managed in an environment where only bare metal devices and the switch fabric to which they connect are provided. Our proposed LB is deployed and run on this architecture.

The control of network traffic in the cloud is just as important as automation. Designing a datacenter network able to handle exponentially increasing traffic as well as rapid changes in traffic patterns is a challenge for cloud service providers. Owing to the steep increase in certain traffic streams, loads may become concentrated on certain functions, causing operational abnormalities and affecting other related functions sequentially. Therefore, to cope with rapidly changing traffic patterns without difficulty, datacenters must provide reliable and scalable cloud services. When using the container orchestration tool, containerized cloud components can be easily deployed and scaled in and out on demand; in addition, duplicate containers ensure reliable operation. In container-based clouds supporting IaaS, applications can easily be created and removed for user services to cover their incoming traffic. Accordingly, the datacenter requires a load balancer (LB) that distributes traffic to multiple containers or VMs with traffic assigned a representative IP, namely a virtual IP (VIP), for a specific service as a destination. In particular, LBs should be able to deliver incoming traffic to multiple containers/VMs whose number and location change frequently. In addition, they should be scalable and able to reliably deliver traffic in any situation. To meet these requirements, LBs can also be containerized and deployed through container orchestration tools. The use of a container-based LB enables the appropriate deployment and management of the cloud resources according the choice of the designers.

Among several load-balancing modes. the most well-known include proxy, inline, and direct server return (DSR). In particular, DSR is mostly classified as L2DSR and L3DSR depending on whether the destination MAC address or the destination IP address is changed. The proposed LB uses the L2DSR mode, which modulates a destination MAC address and a VLAN ID to reduce the load on LBs and increase the response speed by responding to clients directly without passing through the LB. Therefore, the LB does not require any state information about the traffic. The proposed LB does not treat UDP and TCP packets differently. Stateful implementations exhibit performance differences for each protocol, but because L2DSR is stateless, the role and behavior of the LB are exactly the same even if the protocol is different.

There are several challenges in terms of container-based cloud computing systems. First, a policy on the deployment of a containerized LB (API for creation and deletion, status monitor, etc.) should be defined along with the establishment of packet-matching rules. Second, because containers typically run on Linux, the load-balancing performance on Linux must be sufficient to the same extent as commercial physical equipment. In this study, we present an LB architecture addressing these challenges. We automate the LB deployment, and the rules of load balancing based on custom resource definition (CRD), where CRD is a custom object provided by the Kubernetes API. We implement LB using the extended Berkeley Packet Filter and eXpress Data Path (called eBPF/XDP) to ensure sufficient load-balancing performance for the cloud.

The contributions of our work are as follows. First, we define a suitable cloud-native environment to deploy and run our proposed load balancer. Second, in this study, we present a container-based L2DSR LB architecture that can be deployed and operated using Kubernetes. Third, we implement an eBPF/XDP-based high-performance LB. Fourth, our LB and its architecture support a variety of network protocols (e.g., VLAN, VXLAN, and Geneve) that enable a multitenant environment by logically dividing the network.

The remainder of this study is organized as follows. Section II describes the background of the technologies that underlie the proposed architecture. In Section III, we discuss the relevant literature related to LBs in cloud architectures. Section IV describes the cloud-native architecture supporting our proposed load balancer. Section V proposes not only a method of deploying LBs as containers and the establishment of packet-matching rules, but also the implementation of high-performance LBs. Section VI discusses the performance of the proposed LB. Finally, Section VII presents our conclusions and indicates some potential avenues for future research.

II. BACKGROUND

Before examining the proposed architecture in detail, we provide some background on the components and tools used herein. At present, companies or organizations operating cloud systems are struggling to implement their own components for IaaS. They also commonly adopt OpenStack [1], an open-source project, which is becoming a standard for IaaS components. More than 150 companies have been participating in this project since 2010. OpenStack consists of several sub-projects with the purpose of controlling available resources [7] such as computing (called Nova [8]), networking (Neutron [9]), object storage (Swift [10]), block storage (Cinder [11]), identity (Keystone [12]), dashboard (Horizon [13]), database (using MariaDB [14] by default), advanced message queue protocol (using RabbitMQ [15] by default), load-balancing (Octavia [16]), provisioning (Ironic [17]), etc.

Nova supports compute instance (virtual machine) provisioning services. It requires Keystone, which authenticates users, and Neutron, which provides virtual and physical networks used by VMs, and Glance, which provides a compute image repository. Swift provides a block storage service to add/remove disks of instances, and Cinder provides a service to independently store user account data. Among the methods of managing OpenStack resources and services, two typical methods are commonly used: OpenStack Client (a command-line interface tool) and Horizon, which is a web interface allowing cloud administrators and users to manage various OpenStack resources and services. A relational database management system such as MariaDB can be used as an OpenStack database. OpenStack uses message queues such as RabbitMQ to exchange and coordinate operations and state information among services. Octavia provides users with APIs to create, modify, and delete LBs, and supports multiple provider drivers (plugins for various entities that actually perform load balancing). It is possible to hierarchically define which traffic loads are balanced to which backends with a logical model consisting of LBs, listeners, pools, and members. In the LB created in Octavia, one or more listeners with the specified port number (e.g., 80 for HTTP) of the traffic to be received can be registered. Each listener can specify a pool along with the load balancing method (e.g., round-robin) and the members that will finally receive the balanced traffic can be registered in the pool with their IP and port information. Because the proposed LB uses L2DSR mode, it needs a MAC address of backends. Therefore, if a member is added to Octavia along with its IP and subnet information, Octavia queries Neutron with the information and receives the relevant MAC address. While Nova supports the provisioning of virtual machines, Ironic is a service for provisioning bare metal physical devices and includes plugins that can interact with the bare metal hypervisors. With Ironic, heterogeneous hardware devices can be managed like virtual machines through a unified interface.

OpenStack provides IaaS by harmonizing various services, assuming that more than one service may encounter problems simultaneously. In clouds with several services like OpenStack and dependencies between them, it is not easy to reproduce previous sets of services. As noted in the previous chapter, the deployment of components can be automated via orchestration. Specialized orchestration tools such as Kubernetes [4] are indispensable in orchestrating containerized cloud components. Kubernetes features the ability to update or modify applications without interrupting the service, as well as a self-healing function that creates a duplicate container immediately and maintains the service even if a specific container fails. A pod, the smallest unit, contains containers, has an IP address, and shares a namespace, network, etc. A deployment defines how Kubernetes should create and update pods. In addition, Kubernetes includes various resources such as Services and Ingresses for external communication of pods, jobs and DaemonSets that control conditions or number of pods, and ConfigMaps and Secrets corresponding to metadata.

In many cases, components are built with containers and deployed in clusters of different environments. However, many operators struggle with the complexity of the setup and procedures involved in deploying components in different environments. To solve this problem, Helm charts [18] are a collection of files that describe a related set of Kubernetes resources. Only configuration values that vary depending on the deployment environment are defined in advance, and components can be deployed to the Kubernetes cluster by combining the template and configuration values. OpenStack, which consists of various components, also has a Helm chart under the name of OpenStack-Helm project that allows it to be easily, resiliently, and flexibly deployed in various Kubernetes environments. OpenStack packaged as a Helm chart ensures stable cloud operation by ensuring that components are restored to the same state in the same cluster through Kubernetes even if a problem occurs. In addition, when the load on components increases, it is easy to scale out the components by simply changing the setting values of the Helm chart. For example, if we want to increase the number of Neutron Server processes providing APIs for networking control, we can increase the pod.replicas.server value in the values.yaml file for the Neutron helm chart.

III. RELATED WORK

The cloud computing field is currently undergoing active and extensive research. Recent studies have implemented architectures or test environments using a combination of OpenStack (to support virtualization and control of cloud resources) and Kubernetes (to manage applications in the form of containers). With OpenStack and Kubernetes, Kristiani *et al.* [19] implemented an edge computing architecture and configured test environments for an availability manager [20], distributed cloud applications using network function chains [21], and conducted a performance analysis of a container networking interface (CNI) [22]. Yang and Huang [23] implemented a microservices-based OpenStack monitoring system by utilizing Kubernetes. In contrast, in this study, we provide a practical cloud architecture that enables IaaS services using these two tools.

Several researchers have also attempted to develop Kubernetes itself in terms of availability [24]–[26] and QoS [27]. Through the findings of Abdollahi Vayghan *et al.* [24], we can check the availability of container-based applications in some scenarios by using Kubernetes. For high availability and integrity with strong consistency, Netto *et al.* 25] proposed a layer called Koordinator between the client and stateful containers. Similarly, Abdollahi Vayghan *et al.* [26] proposed a state controller allowing for state replication and automated redirection to service available entities. Research efforts have also been conducted with a focus on increasing the size of clusters while guaranteeing QoS and efficiently using resources [27]. The present work does not deal with the enhancement of the functionality of Kubernetes itself, but instead focuses on using its functions to create the desired IaaS cloud functionalities.

Load balancing can be categorized as (i) link load balancing (for balancing traffic across links [28]) and (ii) server load balancing (for balancing traffic between servers). The scope of this work is limited to server load balancing. In the early period of the evolution of cloud computing, the performance and scalability of load balancing also attracted the attention of researchers [29], [30]. Along these lines, Fayoumi [29] presented a simulation model designed to verify load balancing performance, and Patel *et al.* [30] provided a function to modify packets in every host to improve the scalability of LBs. In addition to the basic role of an LB in distributing traffic, research has also been conducted on algorithms that enable resource-efficient [31] or energy-efficient [32] traffic distribution. Load-balancing algorithms may be classified into two types, including (i) static load-balancing algorithms (which distribute traffic equivalently among all available target servers (e.g., round robin) or VMs) and (ii) dynamic load-balancing algorithms (which distribute traffic according to the capacity of all available servers or VMs (e.g., least connections) [31]–[33]). Static LB algorithms [31] are stable and resource-efficient, but they do not respond properly in case of failure. Conversely, dynamic LB algorithms respond appropriately in case of failures, but they are less stable when using a relatively large number of resources. Thus, the use of each algorithm involves advantage and disadvantages. However, Mohammed *et al.* [31] managed to combine the advantages of both algorithms. In terms of important metrics of LB such as average response time, the well-known load-balancing algorithms (round robin, throttled, and equally spread current execution, etc.) have been compared [34]–[36]. Our proposed LB is a type of static LB algorithm because the target is determined based on a hash function without considering the current state or behavior of targets while selecting the target.

Table 1 is the summarization of existing works comparable to the proposed LB. A few studies [37]–[39] have been conducted in which LBs were deployed in the form of a container. An LB was implemented at the Internet Protocol Virtual Server (IPVS) level of the Linux kernel [37] or as a containerized Nginx proxy [38], [39]. As the performance requirements to handle packets in Linux are becoming progressively more demanding, solutions such as eBPF have emerged [40], [41]. Whereas BPF [42] is a lightweight VM that runs programs injected from the user space and is attached to specific hooks in the kernel, eBPF is an extended version of the same that supports maps (key/value stores without any restriction on size) and an expanded set of registers and instructions, among other additions. XDP is a type of eBPF hook, which is a set of commands that operate within the device driver (DD). Thus, eBPF and XDP are generally used together in many cases (e.g., for network functions such as firewalls [43] or as a network traffic visualization tool [44]). For an even better performance than eBPF/XDP-based LBs, Firestone *et al.* [45] designed SmartNIC to offload host networking to hardware.

TABLE 1. Summary of comparison of works related to the proposed LB.

Ref.	Implemented Location	Container -based	Remark
[37]	Linux Kernel	○	internet protocol virtual server (IPVS) load balancer
[38], [39]	Linux Userspace	○	NGINX load balancer
[40]	Linux Kernel		eBPF/XDP-based iptables
[41], [43]	Linux Kernel		eBPF/XDP-based firewall and packet filtering
[44]	Linux Kernel		eBPF/XDP-based packet filtering in Kubernetes cluster
[45]	SmartNIC		offloading host networking to hardware
LB we propose	Linux Kernel	○	eBPF/XDP-based load balancer

IV. CLOUD-NATIVE ARCHITECTURE WITH LOAD BALANCER

A. KUBERNETES CLUSTERS

In this section, before describing the LB architecture in further detail, we describe the cloud-native environment in which the LB is deployed and operated. Fig. 1 shows how each cluster with different roles is created and which pods are deployed to the related clusters. Note that we show only the essential clusters in this figure: Ring 0, OpenStack, LB, and Shared clusters. For each physical machine (server) in the datacenter, one port (eth0) is connected to the top-of-rack (TOR) switch, also called a leaf switch, of the service network, and the other port (eth1) is connected to the TOR switch of the management network. Fig. 1 shows a cloud environment with greater emphasis on the service network side; although all servers are connected to the management network, they are omitted in this figure.

We assumed that each server in the initial datacenter has no configuration and OS, and is connected to only one TOR in the service fabric.

1) RING 0 CLUSTER

At specific servers, the cloud administrator manually installs a cluster called Ring 0 (name taken from the Linux kernel). This cluster not only installs and manages other clusters on other servers but also manages itself. Three open-source projects were used for cluster creation and management. The

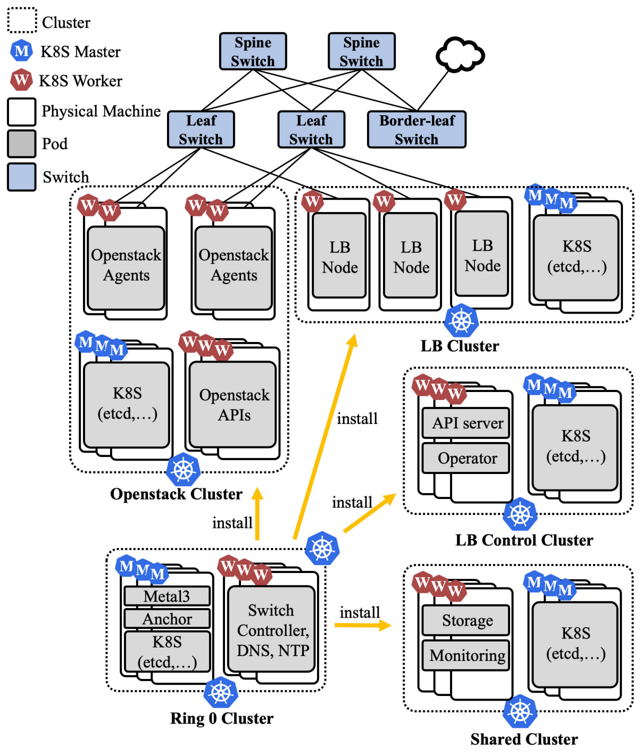


FIGURE 1. **Various clusters in the cloud and their relationships.**

first project is the Cluster API [46], which is a Kubernetes sub-project providing APIs to create and manage a cluster with various cloud providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), as well as bare-metal hosts. The second project is the Metal3 [47] project, which acts as a plug-in for the Cluster API to provision bare metal hosts. Using Metal3, servers to be provisioned can be defined and managed as a CRD, which is a custom object provided by the Kubernetes API. The third project, Ironic, provides APIs that join physical machines to the cloud. When the custom resource (CR) related to the bare metal is modified through the Cluster API, it is detected by the operator in Metal3. Then, the provisioning is reflected on the corresponding physical machine by the Ironic API. During this process, a pre-designated OS image is installed on each server with labeling jobs, indicating the intended usage of each server. Physical machines can be provisioned in a Kubernetes environment. As shown in Fig. 1, various clusters were created according to their purpose. The Ring 0 cluster uses a Kubernetes provisioning script (called Anchor, which we have defined in advance) for the installation of other clusters. These scripts in Ring 0 include information on host images to be installed, host labels, etc. The Kubernetes applications for each cluster are defined as Helm charts. Kubernetes applications for each cluster are deployed by Helm in bare metal systems provisioned by Ring 0. Applications deployed in each cluster are managed as separate repositories.

Cluster API, which installs and manages other clusters, is installed only on the Ring 0 cluster. This begs the question of which cluster deploys and manages Ring 0. Ideally Ring 0 would deploy and manage itself, but this behavior is not supported by the Cluster API. Therefore, we set Ring 0 to see itself as a different cluster and manage it. To make this possible, several steps are necessary. First, manually configure the initial network related to Ring 0. This network provides communication between nodes of a Ring 0 cluster. Second, install Ring 0 cluster manually on one PM which is a first master node of Ring 0 in the form of a cluster installed by Ring 0 through Cluster API. Third, install Metal3 and Cluster API in the installed Ring 0 cluster. Fourth, create CRD for itself in Ring 0 cluster to be managed through Metal3. Fifth, after provisioning another PM as the second master of the Ring 0 cluster, join it to the Ring 0 cluster. Sixth, move the Metal3 and Cluster API from the first master to the second master. As a final step, remove the first master from the cluster.

If this process is performed, the Ring 0 cluster corresponding to the remaining master can not only provision other clusters, but also manage itself.

2) SHARED CLUSTER

The shared cluster contains applications related to storage and monitoring, which require common access from other clusters. Services that are commonly used in multiple clusters are deployed and run on a shared cluster.

3) OPENSTACK

The OpenStack and LB clusters warrant greater attention. Specifically, the OpenStack cluster includes two groups of workers. In the first group, OpenStack API servers such as Nova, Neutron, Keystone, Cinder, Glance, Horizon, and Octavia are operated as pods. We deploy pods by changing configuration values (user account, node label, IP information, number of pods, etc.) of OpenStack-Helm [48] to suit our cluster environment. The second worker group involves VMs created by users, and agents such as a DHCP-agent, OpenVSwitch (OVS)-agent, and Nova-compute support the VMs. In the terminology used in OpenStack, the workers in the second group are regarded as compute nodes.

4) LB AND LB CONTROL CLUSTERS

The proposed approach includes two clusters related to LB. The first cluster is the LB cluster containing pods for load balancing. Herein, we refer to pods that actually receive and distribute traffic as LB nodes (LBNs). Each worker node in the LB cluster runs only one LBN as a DaemonSet. When a node is added to the cluster to which a DaemonSet is applied, a pod related to the DaemonSet is also newly created and runs on the added node. Conversely, when a node is removed from the cluster or a DaemonSet is deleted, the pod created by the DaemonSet is cleaned up. The second cluster is the LB control cluster, which contains the API server and LB operator acting as the controller of LBNs located in the LB cluster, which are described in detail in Section VI.C. The proposed architecture uses two clusters related to LB so that the controllers remain unaffected even if a problem occurs in the LB cluster that handles service traffic. Conversely, even if a problem occurs in the controller pods in the controller cluster, not only does the LB cluster remain unaffected, but the number of controllers is maintained declaratively based on the template defined in the LB controller cluster, to enable the failure to be overcome.

B. MULTITENANT NETWORKS

For multitenant networks, we logically segment networks of different tenants using VLAN ID (VID). Even if the VMs of different tenants have the same IP, the traffic of different networks can be distinguished by their VIDs. As shown in Fig. 1, each compute node has a hypervisor installed, and VMs of users are run on its hypervisor. When VM-related traffic goes out of a node, it is delivered to the TOR physically connected to the compute node. The user traffic travels on a two-tier leaf-spine fabric that consists of leaf, border leaf, and spine switches. The external router is omitted here. When a packet generated in a compute node exits the node, a VLAN header is attached along with the VID of the corresponding network. Because the leaf-spine fabric is setup with VXLAN-EVPN, a VXLAN header is attached to a packet instead of a VLAN header. Each leaf switch has a VXLAN tunnel endpoint (VTEP), which is the start/end point of a VXLAN tunnel. Thus, each leaf switch encapsulates and decapsulates the original user data frames. In a leaf switch, a specific VID is mapped to a specific VNI value of VXLAN by VTEP. Through VXLAN-EVPN, each leaf shares the learned MAC addresses; thus, traffic flooding to all the leaf switches can be minimized. This leads to minimization of the broadcast, unknown-unicast, and multicast (BUM) traffic within the network, thus eliminating various problems that may occur owing to a broadcast storm.

C. LB SETUP PROCEDURE AND L2DSR

For an LBN to perform load balancing, some tasks need to be performed in advance. When a server that is not in use and is connected to a leaf node is joined to the LB cluster with LBN labeling, an LBN is installed as a pod on the server and assigned to one of the LBN groups. Three LBNs are grouped together to handle the same VIP. Therefore, when the LBN is initially installed, a group ID is assigned, and the operator knows which LBN groups exist. In other words, when one LB object is created by Octavia, three LBNs are provided by default. Currently, three LBNs are included in one group. However, this number is not fixed, and changes can be considered depending on the traffic situation of the network. To change the scale of LBNs when the incoming traffic is too bursty, two approaches should be prepared in advance. First, a policy to increase or decrease the number of LBNs should be established referring to the number of VIPs or bandwidth. Second, a component that checks whether the state conforms to the policy is needed. Methods related to these considerations are left for future research as being beyond the scope of the present work. The external

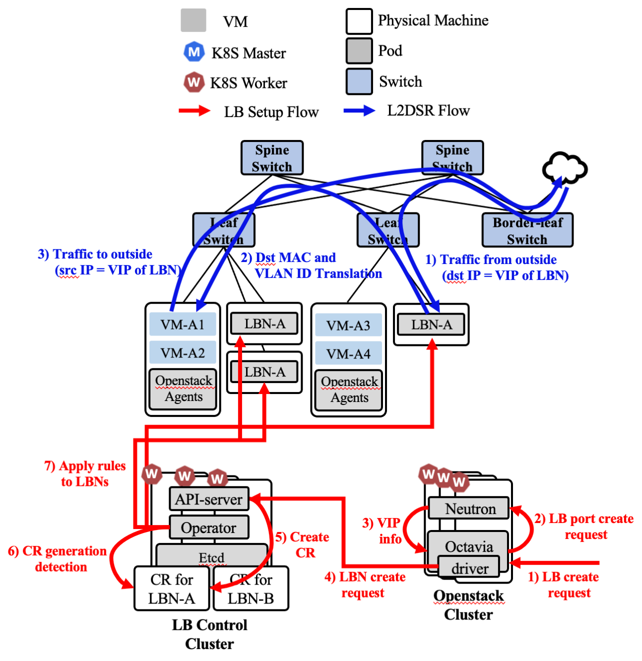


FIGURE 2. Process of setting up LBNs and L2DSR by LBN.

router distributes traffic evenly across the three LBNs through the equal-cost multi-path (ECMP) routing protocol. Because each LBN and the external router are configured to establish a BGP peer, the inbound traffic can be routed from the external router to each LBN.

Fig. 2 shows the components associated to LBNs and the scenario in which LBNs are deployed and used in the cloud. The drivers, API server, operator, and LBNs represent the components we implemented for the proposed architecture. When one or more LBN groups are ready for load balancing, the LBNs must apply prepared rules that specify which packets to send and where. We describe the procedure in which the rules are applied to LBNs and the procedure in which L2DSR is performed by the LBN for service packets as follows.

1) LBN CONTROL PATH

The red arrows in Fig. 2 indicate the series of steps in which packet-matching rules are applied to the LBNs. LB creation begins when Octavia receives a request to create an LB. This request can be made through the OpenStack client or OpenStack horizon. When Octavia receives the request, it sends a request to Neutron to create a port to be used by the LB. When the port is successfully created, Neutron returns a VIP to be used by the LB in response. When the LB creation API is called, the driver sends the LB creation request with the generated port information, including the VIP, to the API server in the LB cluster. The API server creates, modifies, and removes a custom resource (CR) on the basis of the CRD, which is predefined for an LBN group. One CR refers to one LBN group, and CRs are maintained in etcd, which is a key-value store used as the backing store of Kubernetes for all the cluster data. The operator monitors the CR for the LBN group in etcd and controls the corresponding LBNs according to the contents of the CR creation, modification, and deletion. This process is called reconciliation. The operator can change the state of the CR according to the state of the LBN. When the operator recognizes the creation of a new CR, the same rules for traffic destined for the VIP information included in the CR are applied to all the LBNs belonging to the corresponding LBN group. Subsequently, the inbound traffic flows in the order indicated by the blue arrows.

2) DATA PATH OF LBN

Fig. 2 shows an example in which LBN-A, an LBN in group A, distributes traffic to VM-A1, VM-A2, VM-A3, and VM-A4. In particular, the blue arrows indicate a case in which the inbound traffic destined for the VIP of LBN-A is forwarded to VM-A1 by a hashing algorithm in LBN-A. The details of the algorithm are discussed in the next section. The traffic destined for a VIP corresponding to a specific LBN group passes through an external router and a border leaf, and finally arrives at one of the LBNs belonging to the LBN group. The LBN performs hashing based on the header information of the received packet using pre-configured hash tables. Through this hashing, a MAC address and VID are obtained from the related target VM pool. Then, the destination MAC and VID of the received packet are replaced by the obtained information, and the packet is transferred to the nearby leaf switch. In the leaf switch, the packet is encapsulated with VXLAN. Because leaf switches share learned MAC addresses with each other, packets with the destination MAC of VM-A3 are delivered to the TOR switch connected to the server with VM-A1, and the VXLAN header is then decapsulated. At VM-A1, the VIP of LBN-A accepts these packets destined for the VIP of LBN-A because it is set as a loopback IP. The VM responds by swapping the source and destination IPs after processing the packet. Similarly, the leaf node receiving the packet performs VXLAN encapsulation, and VXLAN decapsulation is performed at the border leaf. Subsequently, it is delivered to the external router and exits. As we adopted L2DSR, in which outbound traffic does not flow through LBNs, this leads to a nominal load on the LBNs.

V. IMPLEMENTATION OF LBN

Owing to the nature of cloud datacenters, numerous VMs are generally used, and they can be easily created and removed. LBs that perform load balancing of traffic to these constantly changing VMs should also be easily created and removed according to the network conditions. Physical LBs do not allow flexible handling depending on the amount of traffic and thus entail high financial costs. Therefore, we developed an LB in the form of a container on Linux, which can be easily controlled at the software level. A container-based LB incurs a lower financial cost compared to a physical LB because a general server that is cheaper than a commercial physical LB can be used. Hence, we are actually replacing a part of the commercial physical LB that we are using in a production datacenter with the LB presented here.

A. PERFORMANCE CONSIDERATIONS

The considerations for implementing a software LB comparable to a commercial LB in a Linux environment are as follows. The first is the location of the LB implemented in Linux, and the second is the implementation method. If we look at Linux on the network side from the bottom to the top levels, the layers are usually considered as follows: network interface controller (NIC), DD, traffic controller (TC), Netfilter, TCP stack, socket layer, and Userspace. The lower the layer of the data path in which a packet is processed, the better is the performance. In particular, transferring packets from the DD to the network stack, it is necessary to allocate the packets to the sk_buff structure along with the memory copy, which creates some overhead in terms of performance. We refer to a study [49] that has solved this problem at a low level of the Linux operating system; the performance was improved using the eXpress Data Path (XDP) [50], [51] and extended Berkeley Packet Filter (eBPF) virtual machine for the forwarding plane that processes packets in the kernel. We implemented the LB by exploiting eBPF/XDP to handle packets in the DD layer before they are delivered to the networking layer of the kernel.

B. CONTROLLER AND CORE OF LBN

Fig. 3 shows the LBN core in the Linux kernel and the LBN controller in Userspace. The LBN controller aims to inject the program of the LBN core and update the hash tables that can be used by the LBN core. The proposed approach uses two types of hash tables. The first type is used to check whether there a hash table exists for the VIP of the incoming packet, and the result of hashing for the first type of the hash table is the physical memory address of the VIP. The second type is a hash table composed of backend VM information (VID, MAC address) related to a specific VIP. These hash tables are maintained in BPF maps, which are key/value stores that reside in the kernel, and they can only be updated using an LBN controller. When hash tables need to be changed owing to the addition of a rule, the controller takes the hash tables in the kernel, updates them, and then overwrites them in the kernel BPF maps. Therefore, the hash tables handled by the LBN controller are the same as those handled by the LBN core. As in the example presented in Fig. 3, when a packet enters the LBN server, two hash functions are run to obtain the target information (the destination MAC and VID) for fast matching. A hash table corresponding to the second type exists for each VIP. Therefore, the hash table for the corresponding VIP is found through the first hash function, which requires the destination IP (VIP), destination port, protocol, VID, and segment type (=VLAN) as parameters. A segment type is defined such that other protocols can be used instead of a VLAN. The VLAN is now set as default. A specific VIP table is filled with information about the target VMs to which traffic is to be delivered. For example, the tables for VIPs 1, 2, and 3 are full of entries for three (A, B, C), two (D, E), and two (F, G) VMs, respectively. After finding a hash table for the corresponding VIP through the first hash function, the VM

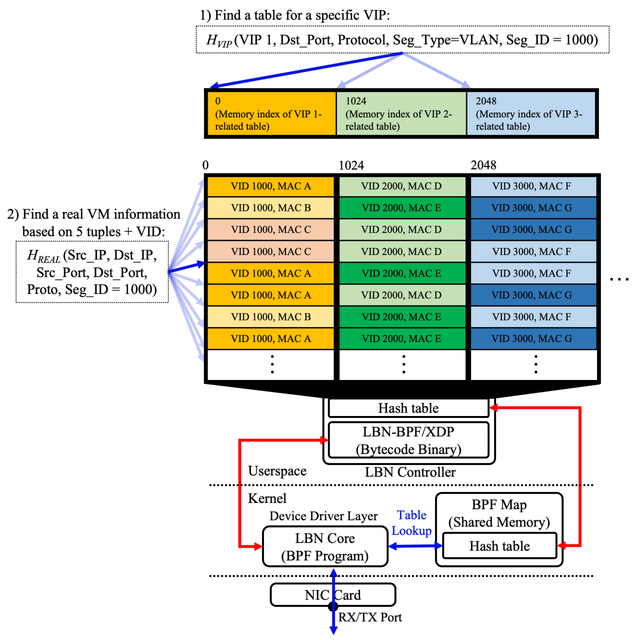

1) Find a table for a specific VIP:
H_{VIP} (VIP 1, Dst_Port, Protocol, Seg_Type=VLAN, Seg_ID = 1000)
0 (Memory index of VIP 1-related table)
1024 (Memory index of VIP 2-related table)
2048 (Memory index of VIP 3-related table)
2) Find a real VM information based on 5 tuples + VID:
H_{REAL} (Src_IP, Dst_IP, Src_Port, Dst_Port, Proto, Seg_ID = 1000)
VID 1000, MAC A
VID 2000, MAC D
VID 3000, MAC F
Hash table
LBN-BPF/XDP (Bytecode Binary)
Userspace
LBN Controller
Kernel
Device Driver Layer
Table Lookup
BPF Map (Shared Memory)
LBN Core (BPF Program)
Hash table
NIC Card
RX/TX Port

FIGURE 3. **LBN architecture across linux user space and kernel space, and hash table format and rule matching description of incoming packets.**

information to which the packet is to be delivered can be obtained through the second hash function, the parameters of which comprise five tuples and the VID of the packet header.

C. HASHING CONSIDERATIONS

Here, we consider how to arrange the entries in each hash table. As is known, a VM can easily join and leave the real target pool; hence, the rules in each hash table can be replaced frequently. Even if only one VM entry is added or deleted from the hash table, the positions of the other existing entries may be changed for reasons such as maintaining the ratio of entries in the hash table. This is called hashing disruption, in which the target VM for a specific VIP is changed because the entry hashing value changes. In other words, an unrelated connection is terminated unintentionally owing to hashing disruption. Eventually, the process of establishing a connection with another target is required. Therefore, studies have investigated hash table mechanisms that support minimal disruption. To solve these problems, we use the Maglev hashing mechanism [52], which is an improved version of the existing consistent hashing algorithm [53]. Maglev hashing allows the locations of existing entries to change as little as possible when new entries are added to the hash table or existing entries are deleted. According to Eisenbud *et al.* [52], entries are positioned in the hash table through two processes, namely permutation, which creates a two-dimensional array by obtaining a non-overlapping order for each entry and population which rotates from entry to entry and places entries fairly. The number of VM entries that our hash table supports is MAX_VIPS $\times$ (MAX_VMS $\times$ FR). MAX_VIPS is the number of VIPs supported by one LBN. The default value is 4096, and it can be changed according to network conditions. MAX_VMS represents the number of VMs that can be handled by each VIP. The reason for multiplying MAX_VMS by FR, which stands for the free ratio of the number of VMs, is that hashing disruption decreases as the size of the table increases, and sufficient space is required when a weight is assigned for each VIP.

VI. EXPERIMENTS

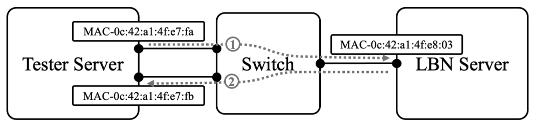
In this section, we describe the performance of the proposed LBN. It is important to determine how close the performance of LBN implemented as software is to the rate at which packets flow on a link. The LB proposed here is a commercial model used in actual datacenters. To prove that the performance of our LB could replace the existing physical commercial LB, the performance was measured using the RFC2544 standard [54], which is widely used for measuring the performance of physical equipment such as firewalls and routers. By specifying the experimental conditions, it is possible to accurately compare the performances of the products of different manufacturers. In addition, to meet the requirements of a datacenter, we performed tests that simulated real-world traffic patterns by using IMIX (Internet Mix) [55] traffic streams, in which several frames are mixed in a specific ratio corresponding to their frame size. The IMIX traffic imitates the traffic patterns that may occur in an actual cloud network. We first detail the experimental setup and then show the performance of the RFC2544 and IMIX tests.

A. EXPERIMENTAL SETUP

To measure the pure performance of LB by excluding other factors from this experiment, the experimental environment suggested by RFC2544 was configured (See Fig. 4). As defined in RFC2544, we constructed an experimental environment by connecting two servers using a single switch. One server acted as a tester and the other server as a device under test (DUT). For each test scenario, the tester server generated packets, sent them to the switch, and received its response to obtain performance metrics such as throughput. We installed TRex, which is an open-source traffic generator for stateful and stateless use cases, and realized low-cost and high-speed operation using Intel's DPDK. Using this tool, we were able to perform tests conforming to RFC standards by simply writing a scenario as a Python script, and their results were comparable to those of commercial physical testing equipment. Note that all CPU cores of the tester server were set for TRex packet transmission, except for the minimum cores to run the tester server. In the DUT server, we installed an LBN that changed the destination MAC address and VID in the header of the packet received from the switch and returned to the switch.

For all the reported runtimes, each server was configured with two 2.10-GHz CPUs (with total of 16 cores, 11 MB L3 cache), and four 32-GB RAM units were used. The Ubuntu 20.04 operating system with Linux kernel 5.4 was installed on each server. A Mellanox Dual-Port 25G NIC card was used in each server. One port of the testing server was configured

Test block diagram



Translation of Header Fields

①

Source IP	172.19.0.2
Destination IP	VIP
VLAN ID	1000
Source MAC	0c:42:a1:4f:e7:fa
Destination MAC	0c:42:a1:4f:e8:03

➡

②

Source IP	172.19.0.2
Destination IP	VIP
VLAN ID	2000
Source MAC	0c:42:a1:4f:e8:03
Destination MAC	0c:42:a1:4f:e7:fb

Scenarios for VIPs

Scenario	Range of VIP	#VIP
A	172.20.0.8	1
B	172.20.0.1 ~ 172.20.0.128	128

FIGURE 4. Test block diagram with test information for translation of header fields and scenarios for VIPs.

to perform RX only, while the other port performed TX only. Meanwhile, the DUT server had only one port, which performed both RX and TX. To check packets including VLAN headers at the kernel level, the rxvlan and txvlan offload functions of the kernel were turned off on both the servers. In the tester server, we installed the MLX5 poll mode driver, which is the driver for Mellanox NIC cards to use DPDK.

Iptables belonging to the Linux Netfilter project provides functions for filtering and controlling (NAT, etc.) packets. Among these functions, DNAT is a function that can be used simply for load balancing. In other words, packets can be distributed according to DNAT rules which change the destination address of incoming packets. Therefore, we chose Iptables DNAT as a comparison target to verify the performance of the LBN. In addition, a loopback test was performed to transmit packets by directly connecting the TX and RX ports of the tester server to obtain the upper limit of the load-balancing performance.

One LBN in the proposed approach performs load balancing with multiple VMs per VIP. Therefore, we consider the number of VIPs and VMs as modifiable parameters in the experiment. In the scenario in which an LBN was used, two cases were considered, including a cause with 1 VIP and 255 VMs, and another with 128 VIPs and 4000 VMs. The destination IP of the traffic sent from the tester was VIP and it was randomly selected. The range of the selected VIPs is the same as the range of the list of VIPs of rules added to the hash table of the LBN. Note that LBNs belonging to the same group had the same rules. Because this test was for L2DSR, the destination IP of the matched traffic did not change, whereas the VID and destination MAC address were changed to the VID and MAC address of one of the related real target VMs. To make Iptables DNAT perform the same operation as LBN, the rules for 128 VIPs were applied to Iptables DNAT equally. The switch connecting each server acted as a bridge that only flooded incoming packets to the port connected to the other server. For a general L2DSR configuration, each

TABLE 2. Experimental results.

				Loopback		DNAT		LBN (#VIPs=1, #Real-servers=255)		LBN (#VIPs=128, #Real-servers=4000)	
	Size	Proto.	Metric	L2	L1	L2	L1	L2	L1	L2	L1
RFC2544	64	UDP	BPS (GB)	19.26	24.93	0.95	1.22	15.29	19.78	14.64	18.95
			PPS (MB)	35.41		1.74		28.10		26.91	
		TCP	BPS (GB)	19.26	24.93	0.92	1.19	15.25	19.75	14.61	18.98
			PPS (MB)	35.41		1.69		28.17		26.96	
	128	UDP	BPS (GB)	21.34	24.57	1.78	2.05	20.83	23.99	20.73	23.87
			PPS (MB)	20.21		1.73		19.73		19.63	
		TCP	BPS (GB)	21.33	24.51	1.66	1.91	20.82	23.92	20.76	23.87
			PPS (MB)	20.28		1.57		19.80		19.64	
	256	UDP	BPS (GB)	22.99	24.76	3.58	3.85	22.93	24.69	22.91	24.67
			PPS (MB)	11.06		1.72		11.02		11.01	
		TCP	BPS (GB)	22.93	24.75	3.70	3.98	22.96	24.62	22.98	24.64
			PPS (MB)	11.01		1.78		11.09		11.00	
	512	UDP	BPS (GB)	24.04	24.97	6.97	7.24	23.96	24.88	23.88	24.81
			PPS (MB)	5.82		1.69		5.80		5.79	
		TCP	BPS (GB)	24.03	24.94	6.60	6.86	24.00	24.82	23.81	24.84
			PPS (MB)	5.82		1.60		5.81		5.78	
	1024	UDP	BPS (GB)	24.45	24.92	13.60	13.86	24.37	24.85	24.42	24.89
			PPS (MB)	2.97		1.65		2.96		2.97	
		TCP	BPS (GB)	24.47	24.95	12.23	12.46	24.35	24.83	24.49	24.87
			PPS (MB)	2.98		1.49		2.96		2.97	
	1280	UDP	BPS (GB)	24.62	24.95	15.39	15.63	24.50	24.88	24.55	24.93
			PPS (MB)	2.39		1.50		2.39		2.39	
		TCP	BPS (GB)	24.68	24.97	12.27	12.46	24.46	24.84	24.54	24.92
			PPS (MB)	2.39		1.19		2.38		2.40	
	1510	UDP	BPS (GB)	24.64	24.97	15.29	15.49	24.56	24.89	24.62	24.95
			PPS (MB)	2.03		1.26		2.03		2.03	
		TCP	BPS (GB)	24.62	24.94	12.30	12.47	24.52	24.85	24.61	24.94
			PPS (MB)	2.04		1.01		2.03		2.04	
IMIX	BPS (GB)			21.19	24.95	1.36	1.60	20.56	24.28	20.56	24.27
	PPS (MB)			23.51		1.52		23.21		23.19	

port of a switch attached to an LBN server was set to the trunk mode to accept traffic attached to various VLANs; however, in the experimental environment, the ports were set to the access mode so that only specific VLANs could be sent and received. Table 2 shows the results of our experiments. Based on these data, the results are analyzed in the next section.

B. RFC2544 PERFORMANCE TEST

In this section, we specify the parameters of the experiment conducted according to the RFC2544 standard and show the test results based on throughput achieved (in bps and pps).

We tested seven frame sizes: 64, 128, 256, 512, 1024, 1280, and 1510. Specifically, four types of tests were performed

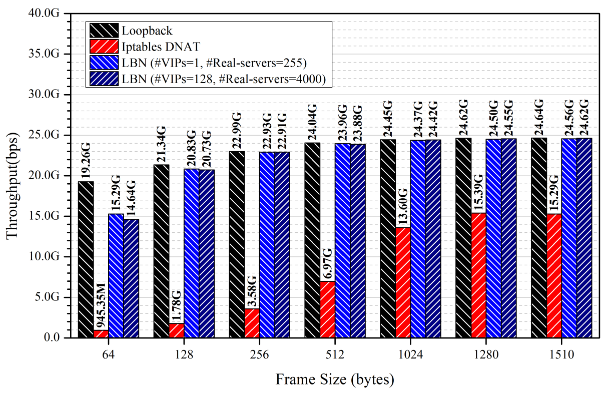


FIGURE 5. UDP Throughput (in bps at L2 level) per frame size for RFC2544.

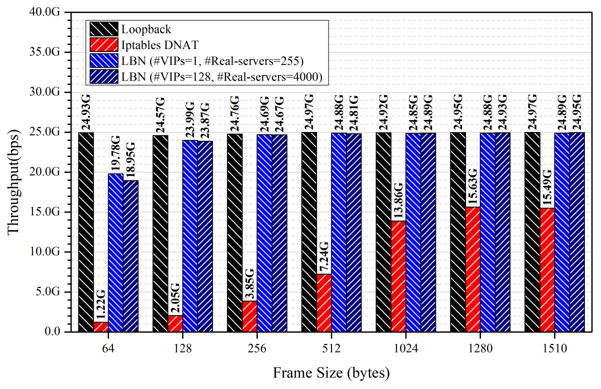


FIGURE 6. UDP throughput (in bps at L1 level) per frame size for RFC2544.

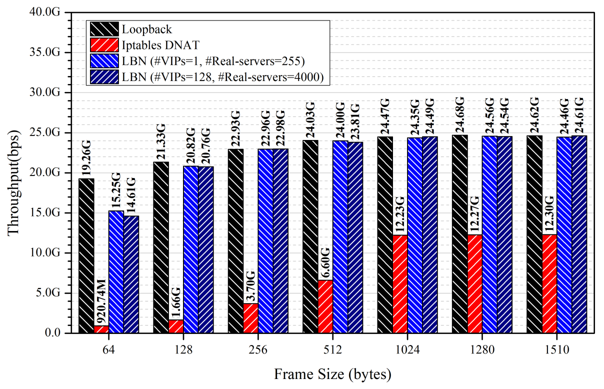


FIGURE 7. TCP throughput (in bps at L2 level) per frame size for RFC2544.

for each frame size: loopback, DNAT of Iptables, scenario A for LBN (#VIP = 1, #Real-VM = 255), and scenario B for LBN (#VIP = 128, #Real-VM = 4000). We transmitted UDP/TCP traffic with each Ethernet frame size for 60 s at the maximum speed of the 25G NIC card in one trial, and the average throughput of the three trials was used as the final result. The interval for each trial was set to 30 s to avoid interference between trials. In particular, in the case of TCP packets for experiment, each packet is set to create a connection by enabling a SYN flag.

Figs. 5, 6, and 7 show the throughput of each scenario when traffic consisting only of frames of a certain size is transmitted with the full rate of the 25G NIC card. In particular, Figs. 5 and 6 show the throughput (in bps) at the L2 and L1 levels, respectively. Fig. 8 shows the throughput (in pps) at the L2 level. In the results for all the frame sizes, the loopback performance can be seen as the best performance that the LB can achieve.

In Figs. 5 and 6, it can be seen that the smaller the frame size, the lower the bps rate, owing to the increase in the overhead of handling frames. Fig. 5 shows the results after the decapsulation of both the headers of layers 1 and 2 of the frames, and Fig. 6 shows the results after the decapsulation of only the header of layer 1 of the frames; thus, the performance shown in Fig. 6 was better.

Three pertinent questions may arise from the given figures. First, how close is the performance of the proposed LBN to that of the loopback interface? Second, how different are the performances of Iptables DNAT and the LBN? Third, how much is the difference in performance for varying numbers of rules applied to the LBN? For the bps rate for L1 and L2, when the frame size was 64, the difference between loopback and LBN was less than approximately 5–6 Gbps. In other words, the performance of the proposed LBN was only approximately 24% lower than the realistic maximum performance. When the frame size was 128, the difference was only approximately 3%, and the performance was nearly constant at larger frame sizes. Apart from the unavoidable performance degradation for handling the frame size of 64, i.e., the smallest frame size, the LBN showed remarkably high performance. In the worst case (when the frame size was 64), the performance of the LBN was 16 times better than that of Iptables DNAT. Even when the difference was the smallest (when the frame size was 1510), they differed by a factor of approximately 1.6. When comparing the two scenarios with different numbers of rules for LBN, the worst-case (when the frame size was 64) performance decreased by approximately 4.2% as the number of rules increased. However, because there is nearly no difference at the other frame sizes, we can see that the number of rules does not affect the performance significantly. Fig. 7 shows the performance of each frame size for TCP traffic at the L2 level. We can see that the performance is almost the same between Fig. 5 and Fig. 7. This indicates that the LBN is an L2DSR and a stateless LB. Fig. 8 shows the results of Fig. 5 in terms of pps. In contrast to the above, it can be seen that the smaller the frame size, the higher the pps rate, because more packets were sent. When the frame size was 64, the difference between loopback and LBN was less than approximately 7–8 Mpps. Of course, the percentage values for the three questions were the same as those mentioned in Figs. 5 and 6.

C. IMIX PERFORMANCE TEST

In this section, we describe the parameters of the IMIX experiment and show the test reports with the throughput figures. The actual Internet traffic mix changes over time.

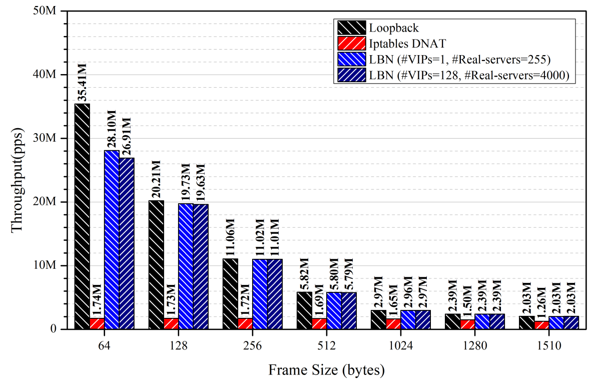


FIGURE 8. UDP throughput (in pps at L2 level) per frame size for RFC2544.

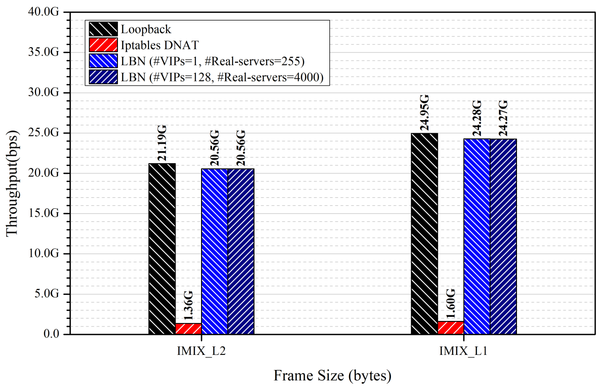


FIGURE 9. Throughput (in bps at L2 level) for IMIX.

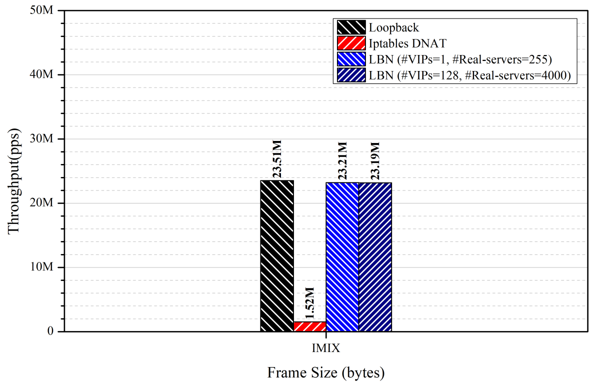


FIGURE 10. Throughput (in pps at L2 level) for IMIX.

However, there is widely adopted packet distribution for quick approximations [55]. The ratio of each frame size used in our experiment was as follows: 58.333%, 33.333%, and 8.333% for frames with sizes of 40, 576, and 1500, respectively. Because the test was conducted by mixing traffic of various frame sizes, it was similar to the actual network environment. Figs. 9 and 10 show the bps (at the L2 and L1 levels) and pps results, respectively, for this IMIX traffic. The difference in bps measured at the L2 level between loopback and LBN was less than approximately 0.7 Gbps. In other words, LBN was only approximately 2% lower than the theoretical maximum performance. The performance of the proposed LBN was 27 times better than that of Iptables DNAT. When comparing the two scenarios with different numbers of rules for the LBN, there was only a marginal difference. Similar to the RFC255 test, we can see that the number of rules did not significantly affect the performance of the LBN.

D. LBN DEPLOYMENT TEST

First, it was necessary to check whether our proposed LB was deployed quickly in an environment where the cloud was composed of components packaged in containers. Accordingly, we first checked how quickly an LBN was prepared. When the LBN label was attached to the worker of the LB cluster, Kubernetes installed the LBN to the worker node. The preparation time of LBN required less than 1 s, and frequently was occupied by downloading the container image (about 100 MB) of LBN. Therefore, it required about 3 s or less for a general server to transform into a high-performance LB, although the exact time depended on the network speed. Because the container image contained libraries used by LB, there was room to reduce the size of the container image to shorten the deployment time. Additionally, it may be recommended to preload the image into the node.

Second, we needed to check how quickly the rules made in Octavia were applied to the LBN. When we added a set of LB, listener, pool, and member in Octavia, a target rule for one VIP was created and delivered to LBN. When we tested this pattern multiple times, it took about 3 s on average, and most of the bottleneck was identified as the time required for Octavia, Keystone, Neutron, etc. in OpenStack.

VII. CONCLUSION

In this study, we introduced an installable cloud that enables IaaS services by utilizing open-source solutions, such as Kubernetes, Cluster API, and OpenStack. For this architecture, we defined the Ring 0 cluster for deployment and management of clusters, the OpenStack cluster for IaaS services, the shared cluster for an accessible area in different clusters, and clusters for LB. To deploy LBNs and apply rules in LBNs in this containerized environment, we implemented drivers of Octavia, the API server, the operator, etc. We were able to implement an LB with sufficient performance in a commercial cloud environment by processing packets using eBPF/XDP in the Linux kernel. Our experimental results indicate that the throughput of the proposed LBN was significantly better than that of Iptables DNAT, and the difference in performance increased as the packet size decreased. As the difference in performance between loopback (representing the theoretical maximum performance limit) and LBN was minimal, it can be concluded that the proposed LB can be used in commercial applications. The proposed LB is based on eBPF/XDP and already exhibits excellent performance; our future goal is to achieve hardware-level performance by utilizing SmartNIC. We believe that the LB mechanism outlined herein will find application outside of cloud computing in the future.

ACKNOWLEDGMENT

The authors would like to thank the Cloud Team, Kakao Enterprise, for informative discussions. They would also especially thank Cheng-Dae Choe and Woo-Chan Go for their generous support.

REFERENCES

[1] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Gener. Comput. Syst.*, vol. 79, pp. 849–861, Feb. 2018.

[2] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2019, pp. 1–6.

[3] M. Rouse. (Aug. 2016). *What is Docker Swarm?*. TechTarget. [Online]. Available: https://searchitoperations.techtarget.com/definition/Docker-Swarm

[4] *Kubernetes*. Accessed: Sep. 2021. [Online]. Available: https://kubernetes.io/

[5] *Apache Mesos*. Accessed: Sep. 2021. [Online]. Available: https://mesos.apache.org/

[6] *OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://github.com/openstack

[7] T. Rosado and J. Bernardino, "An overview of openstack architecture," in *Proc. 18th Int. Database Eng. Appl. Sym.*, 2014, pp. 366–367.

[8] *Nova Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/nova/latest/

[9] *Neutron Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/neutron/latest/

[10] *Swift Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/swift/latest/

[11] *Cinder Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/cinder/latest/

[12] *Keystone Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/keystone/latest/

[13] *Horizon Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/horizon/latest/

[14] *MariaDB*. Accessed: Sep. 2021. [Online]. Available: https://mariadb.org/

[15] *RabbitMQ*. Accessed: Sep. 2021. [Online]. Available: https://www.rabbitmq.com/

[16] *Octavia Sub-Project of OpenStack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/octavia/latest/

[17] *Ironic Sub-Project of Openstack*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/ironic/latest/

[18] *Helm*. Accessed: Sep. 2021. [Online]. Available: https://helm.sh/

[19] E. Kristiani, C.-T. Yang, Y. T. Wang, and C.-Y. Huang, "Implementation of an edge computing architecture using openstack and kubernetes," in *Proc. Int. Conf. Inf. Sci. Appl.* Singapore: Springer, 2018, pp. 675–685.

[20] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," 2019, *arXiv:1901.04946*. [Online]. Available: https://arxiv.org/abs/1901.04946

[21] H. R. Kouchaksaraei, T. Dierich, and H. Karl, "Pishahang: Joint orchestration of network function chains and distributed cloud applications," in *Proc. 4th IEEE Conf. Netw. Softwarization Workshops (NetSoft)*, Montreal, QC, Canada, Jun. 2018, pp. 344–346, doi: 10.1109/NETSOFT.2018.8460134.

[22] Y. Park, H. Yang, and Y. Kim, "Performance analysis of CNI (container networking interface) based container network," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2018, pp. 248–250, doi: 10.1109/ICTC.2018.8539382.

[23] M. Yang and M. Huang, "An microservices-based openstack monitoring tool," in *Proc. IEEE 10th Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Beijing, China, Oct. 2019, pp. 706–709, doi: 10.1109/ICSESS47205.2019.9040740.

[24] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, San Francisco, CA, USA, Jul. 2018, pp. 970–973, doi: 10.1109/CLOUD.2018.00148.

[25] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A service approach for replicating Docker containers in kubernetes," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Natal, Brazil, Jun. 2018, pp. 00058–00063, doi: 10.1109/ISCC.2018.8538452.

[26] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Sofia, Bulgaria, Jul. 2019, pp. 176–185, doi: 10.1109/QRS.2019.00034.

[27] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, "Dynamically adjusting scale of a kubernetes cluster under QoS guarantee," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Tianjin, China, Dec. 2019, pp. 193–200, doi: 10.1109/ICPADS47876.2019.00037.

[28] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, "Load balancing in data center networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 2324–2352, 3rd Quart., 2018, doi: 10.1109/COMST.2018.2816042.

[29] A. G. Fayoumi, "Performance evaluation of a cloud based load balancer severing Pareto traffic," *J. Theor. Appl. Inf. Technol.*, vol. 32, no. 1, pp. 28–34, Oct. 2011.

[30] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud scale load balancing," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, New York, NY, USA, Aug. 2013, pp. 207–218, doi: 10.1145/2486001.2486026.

[31] M. A. Mohammed, R. A. Hasan, M. A. Ahmed, N. Tapus, M. A. Shanan, M. K. Khaleel, and A. H. Ali, "A focal load balancer based algorithm for task assignment in cloud environment," in *Proc. 10th Int. Conf. Electron., Comput. Artif. Intell. (ECAI)*, Iasi, Romania, Jun. 2018, pp. 1–4, doi: 10.1109/ECAI.2018.8679043.

[32] I. Ahmad, M. I. K. Khalil, and S. A. A. Shah, "Optimization-based workload distribution in geographically distributed data centers: A survey," *Int. J. Commun. Syst.*, vol. 33, no. 12, p. e4453, Aug. 2020, doi: 10.1002/dac.4453.

[33] S. G. Eladl, N. I. Ziedan, and T. S. Gaafar, "Cloud computing load balancing using genetic and throttled hybrid algorithm," *Int. J. Eng. Technol.*, vol. 11, no. 3, pp. 606–626, Jun. 2019, doi: 10.21817/ijet/2019/v11i3/191103041.

[34] S. Patel, R. Patel, H. Patel, and S. Vahora, "CloudAnalyst: A survey of load balancing policies," *Int. J. Comput. Appl.*, vol. 117, no. 21, pp. 21–24, May 2015, doi: 10.5120/20679-3525.

[35] M. R. Mesbahi, M. Hashemi, and A. M. Rahmani, "Performance evaluation and analysis of load balancing algorithms in cloud computing environments," in *Proc. 2nd Int. Conf. Web Res. (ICWR)*, Tehran, Iran, Apr. 2016, pp. 145–151, doi: 10.1109/ICWR.2016.7498459.

[36] A. Singh and R. Kumar, "Performance evaluation of load balancing algorithms using cloud analyst," in *Proc. 10th Int. Conf. Cloud Comput., Data Sci. Eng. (Confluence)*, Noida, India, Jan. 2020, pp. 156–162, doi: 10.1109/Confluence47617.2020.9058017.

[37] K. Takahashi, K. Aida, T. Tanjo, and J. Sun, "A portable load balancer for kubernetes cluster," in *Proc. Int. Conf. High Perform. Comput. Asia–Pacific Region*, New York, NY, USA, Jan. 2018, pp. 222–231, doi: 10.1145/3149457.3149473.

[38] M. Pleshakov. (Dec. 2016). *NGINX and NGINX Plus Ingress Controllers for Kubernetes Load Balancing*. [Online]. Available: https://www.nginx.com/blog/nginx-plus-ingress-controller-kubernetes-load-balancing/

[39] D. Zhang, "Resilience enhancement of container-based cloud load balancing service," *PeerJ Preprints*, Apr. 2018, Art. no. e26875v1.

[40] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating Linux security with eBPF iptables," in *Proc. ACM SIGCOMM Conf. Posters Demos*, Aug. 2018, pp. 108–110.

[41] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with Linux eBPF," in *Proc. 30th Int. Teletraffic Congr. (ITC)*, Vienna, Austria, Sep. 2018, pp. 209–217, doi: 10.1109/ITC30.2018.00039.

[42] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. USENIX Winter Conf. (USENIX)*, Berkeley, CA, USA, 1993, pp. 1–11.

[43] D. Anant, R. Huang, and P. Mehra, "eBPF/XDP based firewall and packet filtering," in *Proc. Linux Plumbers Conf.*, 2018, pp. 1–5.

[44] Y.-E. Choe, J.-S. Shin, S. Lee, and J. W. Kim, "eBPF/XDP based network traffic visualization and dos mitigation for intelligent service protection," in *Proc. Int. Conf. Emerg. Internetworking, Data Web Technol.* Cham, Switzerland: Springer, 2020, pp. 458–468.

[45] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, and H. K. Chandrappa, "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018 pp. 51–66.
[46] *Cluster API*. Accessed: Sep. 2021. [Online]. Available: https://cluster-api.sigs.k8s.io/
[47] *Metal3*. Accessed: Sep. 2021. [Online]. Available: http://metal3.io/
[48] *Openstack-Helm*. Accessed: Sep. 2021. [Online]. Available: https://docs.openstack.org/openstack-helm/latest/
[49] C. Hopps. (Sep. 2019). *Katran: A High Performance Layer 4 Load Balancer*. [Online]. Available: https://github.com/facebookincubator/katran
[50] T. Herbert and A. Starovoitov. (Mar. 2016). *The Express Data Path (XDP): Programmable and High Performance Networking Data Path*. Accessed: Oct. 2, 2018. [Online]. Available: https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf
[51] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. EXperiments Technol.*, Dec. 2018, pp. 54–66.
[52] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 1–14.
[53] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput. (STOC)*, 1997, pp. 654–663.
[54] S. Bradner and J. McQuaid. *Benchmarking Methodology for Network Interconnect Devices*. Accessed: Sep. 2021. [Online]. Available: https://tools.ietf.org/html/rfc2544
[55] *IMIX Traffic*. Accessed: Sep. 2021. [Online]. Available: https://en.wikipedia.org/wiki/Internet_Mix

JUNG-BOK LEE received the B.S. degree in computer science and engineering from Konyang University, Nonsan, South Korea, in 2010. In 2017, he worked as part of onLine Plus Corporation Cloud Team. Since 2019, he has been working with the Cloud Team, Kakao Enterprise Corporation. His current research interests include cloud networks and DC architecture.

TAE-HEE YOO received the B.S. degree in computer engineering from Daegu University, Gyeongsan, South Korea, in 2015. He started his career as a Firewall Software Engineer at Future Systems. Since 2020, he has been working as a Cloud Networking Software Engineer with Kakao Enterprise Corporation. He has also been contributing to Netfilter, Virtual Interface, TC, BPF, XDP, and many other areas in the Linux kernel networking stack opensource project.

EO-HYUNG LEE received the M.S. degree in computer science from the University of Yonsei, South Korea. He worked as a Cloud Engineer and a Cloud Architect. He is currently developing cloud services. He also loves working with open source and Linux. He has been doing cloud and software engineering for about 12 years, developing various and complex cloud environments. He has presented at various events, such as "Kubernetes forum Seoul" and "Open Infrastructure and Cloud Native Days Korea," on topics such as "How to Debug the Pod Which is Hard to Debug," "Immutable Kubernetes architecture," and "truly understanding container." His research interests include cloud automation and architecture design.

BYEONG-HA HWANG was born in Masan, South Korea, in 1988. He received the M.S. degree in information and communication engineering from the University of Ajou, Suwon, South Korea. His interest in network data sending and receiving, led him to a career of a network engineer. Since the global cloud market has been growing rapidly and the core of cloud is networks according to Hwang, he joined Kakao Enterprise Corporation, to create a cloud network. His current efforts are to create scalable and stable cloud networks architecture.

SUNG-WON AHN received the M.S. degree in information and telecommunication engineering from Soongsil University, Seoul, South Korea, in 2019. He has worked as a part of the Cloud Network Team, TMAX Cloud Corporation. Since 2021, he has been working with the Cloud Team, Kakao Enterprise Corporation. His current research interest includes cloud networks.

CHOONG-HEE CHO (Member, IEEE) was born in Seoul, South Korea, in 1986. He received the B.S. degree in computer science from Sahmyook University, Seoul, in 2010, the M.S. and Ph.D. degrees in information and communication network technology from Korea University of Science and Technology (UST), Daejeon, and the Ph.D. degree in telecommunication networks and optimization, in 2019. He worked as a Postdoctoral Researcher with Korea Advanced Institute of Science and Technology (KAIST). He is currently working with Kakao Enterprise Corporation. His research interests include cloud networks and DC architecture.