PAPER

A Portable Load Balancer with ECMP Redundancy for Container Clusters

Kimitoshi TAKAHASHI[†,††], ***Nonmember***, **Kento AIDA**[†††,†], ***Member***, **Tomoya TANJO**[†††], **Jingtao SUN**[†††], ***Nonmembers***, ***and*** **Kazushige SAGA**[†††], ***Member***

SUMMARY
Linux container technology and clusters of the containers are expected to make web services consisting of multiple web servers and a load balancer portable, and thus realize easy migration of web services across the different cloud providers and on-premise datacenters. This prevents service to be locked-in a single cloud provider or a single location and enables users to meet their business needs, e.g., preparing for a natural disaster. However existing container management systems lack the generic implementation to route the traffic from the internet into the web service consisting of container clusters. For example, Kubernetes, which is one of the most popular container management systems, is heavily dependent on cloud load balancers. If users use unsupported cloud providers or on-premise datacenters, it is up to users to route the traffic into their cluster while keeping the redundancy and scalability. This means that users could easily be locked-in the major cloud providers including GCP, AWS, and Azure. In this paper, we propose an architecture for a group of containerized load balancers with ECMP redundancy. We containerize Linux ipvs and exabgp, and then implement an experimental system using standard Linux boxes and open source software. We also reveal that our proposed system properly route the traffics with redundancy. Our proposed load balancers are usable even if the infrastructure does not have supported load balancers by Kubernetes and thus free users from lock-ins.
key words: *redundancy, ECMP, load balancer, container, BGP*

1. Introduction

Recently, Linux containers have drawn a significant amount of attention because they are lightweight, portable, and reproducible. Linux containers are generally more lightweight than virtual machine (VM) clusters, because the containers share the kernel with the host operating system (OS), even though they maintain separate execution environments. They are generally portable because the process execution environments are archived into tar files, so whenever one attempts to run a container, the exact same file systems are restored from the archives even when totally different data centers are used. This means that containers can provide reproducible and portable execution environments. For the same reasons, Linux containers are attractive for web services as well, and it is expected that web services consisting of container clusters would be capable of being migrated easily for a variety of purposes. For example disaster recovery, cost performance improvements, legal compliance, and shortening the geographical distance to customers are the main concerns for web service providers in e-commerce, gaming, Financial technology(Fintech) and Internet of Things(IoT) field.

Kubernetes[1], which is one of the most popular container cluster management systems, enables easy deployment of container clusters. Since Kubernetes hides the differences in the base environments, users can easily deploy a web service on different cloud providers or on on-premise data centers, without adjusting the container cluster configurations to the new environment. This allows a user to easily migrate a web service consisting of a container cluster even to the other side of the world as follows: A user starts the container cluster in the new location, route the traffic there, then stop the old container cluster at his or her convenience. This is a typical web service migration scenario.

However, this scenario only works when the user migrates a container cluster among major cloud providers including Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure. Kubernetes does not provide generic ways to route the traffic from the internet into container cluster running in the Kubernetes and is heavily dependent on cloud load balancers, which is external load balancers that are set up on the fly by cloud providers through their application protocol interfaces (APIs). These cloud load balancers distribute incoming traffic to every server that hosts containers. The traffic is then distributed again to destination containers using iptables destination network address translation (DNAT)[13], [18] rules in a round-robin manner. The problem happens in the environment with a load balancer that is not supported by the Kubernetes, e.g. in an on-premise data center with a bare metal load balancer. In such environments, the user needs to manually configure the static route for inbound traffic in an ad-hoc manner. Since the Kubernetes fails to provide a uniform environment from a container cluster viewpoint, migrating container clusters among the different environments will always be a burden.

In order to solve this problem by eliminating the dependency on cloud load balancers, we have proposed a containerized software load balancer that is run by Kubernetes as a part of web service container clusters in the previous work[23]. It enables a user to deploy a web service in different environments without modification easily because the web service itself includes load balancers. We containerized Linux ker-

Manuscript received January 1, 2015.
Manuscript revised January 1, 2015.
[†]The author is with the School of Multidisciplinary Sciences, Dept. of Informatics, The Graduate University for Advanced Studies, Chiyoda-ku, Tokyo, Japan.
[††]The author is with the Cluster Computing Inc., Chiyoda-ku, Tokyo, Japan.
[†††]The author is with the National Institute of Infomatics, Chiyoda-ku, Tokyo, Japan.
DOI: 10.1587/trans.E0.??.1

nel's Internet Protocol Virtual Server (IPVS)[4] Layer 4 load balancer using an existing Kubernetes ingress[3] framework, as a proof of concept. We also proved that our approach does not significantly degrade the performance, by comparing the performance of our proposed load balancer with those of iptables DNAT load balancer and the Nginx Layer 7 load balancing. The results indicated that the proposed load balancer could improve the portability of container clusters without performance degradation compared with the existing load balancer.

However, the way to route traffic from the Internet to load balancers while keeping redundancy has not been discussed in our previous work, even though the redundancy is always needed to improve the availability of services in modern systems. This is because, standard Layer 2 redundancy protocols, e.g., Virtual Router Redundancy Protocol(VRRP)[19] or OSPF[20], which uses multicast, cannot be used in many network environments for containers. Furthermore, providing uniform methods independent of the infrastructures such as various cloud environments and the on-premise data center is much more difficult.

In this paper, we extend the previous work and propose a software load balancer architecture with Equal Cost Multi-Path(ECMP)[31] redundancy by running a Border Gateway Protocol(BGP) agent container together with ipvs container. Although major cloud providers do not currently provide BGP peering service for their users, the authors expect they start such service, once this approach is proven beneficial. For the cloud environment without BGP peering service, we can still launch our proposed load balancer without ECMP redundancy by sending out API request to automatically set up a route to the load balancer, from inside the ipvs container.

In order to demonstrate the feasibility of the proposed load balancer, we containerize an open source BGP software, exabgp[26], and also containerize Linux kernel's ipvs load balancer. Then we launch them as a single pod, which is a group of containers that share a single net namespace using Kubernetes. We launch multiple of such pods and form a cluster of load balancers. We demonstrate the functionality and evaluate preliminary performance.

The contributions of this paper are as follows: Although there have been studies regarding redundant software load balancers especially from the major cloud providers[21], [22], their load balancers are only usable within their respective cloud infrastructures. This paper aims to provide a redundant software load balancer architecture for those environments that do not have load balancers supported by Kuberenetes. Since proposed load balancer architecture uses nothing but existing Open Source Software(OSS) and standard Linux boxes, users can build a cluster of redundant load balancers in their environment.

The rest of the paper is organized as follows. Section 2 highlights related work that deals specifically with container cluster migration, software load balancer containerization, and load balancer related tools within the context of the container technology. Section 3 discusses problems of the existing architecture and proposes our solutions. In Section 4, we explain experimental system in detail. Then, we show our experimental results and discuss obtained characteristics in Section 5, which is followed by a summary of our work in Section 6.

2. Related Work

This section highlights related work, especially that dealing with container cluster migration, software load balancer containerization, load balancer tools within the context of the container technology and scalable load balancer in the cloud providers.

(1) Container cluster migration:

Kubernetes developers are trying to add federation[2] capability for handling situations where multiple Kubernetes clusters† are deployed on multiple cloud providers or on-premise data centers, and are managed via the Kubernetes federation API server (federation-apiserver). However, how each Kubernetes cluster is run on different types of cloud providers and/or on-premise data centers, especially when the load balancers of such environments are not supported by Kubernetes, seems beyond the scope of that project. The main scope of this paper is to make Kubernetes usable in environments without supported load balancers by providing a containerized software load balancer.

(2) Software load balancer containerization:

As far as load balancer containerization is concerned, the following related work has been identified: Nginx-ingress[5], [6] utilizes the ingress[3] capability of Kubernetes, to implement a containerized Nginx proxy as a load balancer. Nginx itself is famous as a high-performance web server program that also has the functionality of a Layer-7 load balancer. Nginx is capable of handling Transport Layer Security(TLS) encryption, as well as Uniform Resource Identifier(URI) based switching. However, the flip side of Nginx is that it is much slower than Layer-4 switching. We compared the performance between Nginx as a load balancer and our proposed load balancer in this paper. Meanwhile, the kube-keepalived-vip[7] project is trying to use Linux kernel's ipvs[4] load balancer capabilities by containerizing the keepalived[8]. The kernel ipvs function is set up in the host OS's net namespaces and is shared among multiple web services, as if it is part of the Kubernetes cluster infrastructure. Our approach differs in that the ipvs rules are set up in container's net namespaces and function as a part of the web service container cluster itself. The load balancers are configurable one by one, and are movable with the cluster once the migration is needed. The kube-keepalived-vip's approach lacks flexibility and portability whereas ours provide them. The swarm mode of the Docker[11], [12] also uses ipvs for internal load balancing, but it is also considered as part of Docker swarm infrastructure, and thus lacks

†The *Kubernetes cluster* refers to a server cluster controlled by the Kubernetes container management system, in this paper.

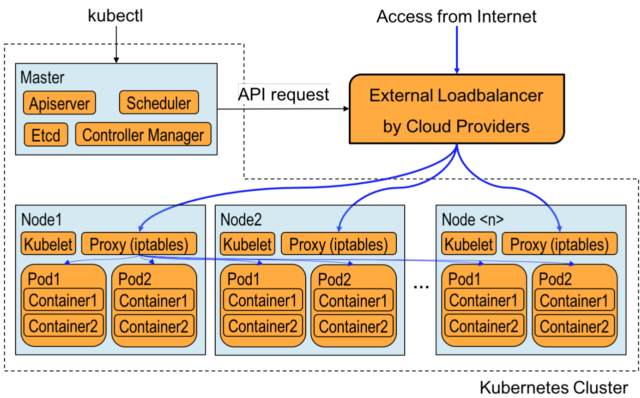


Fig. 1: Conventional architecture of a Kubernetes cluster.

the portability that our proposal aims to provide.

(3) **Load balancer tools in the container context:**

There are several other projects where efforts have been made to utilize ipvs in the context of container environment. For example, GORB[9] and clusterf[10] are daemons that setup ipvs rules in the kernel inside the Docker container. They utilize running container information stored in key-value storages like Core OS etcd[14] and HashiCorp's Consul[15]. Although these were usable to implement a containerized load balancer in our proposal, we did not use them, since Kubernetes ingress framework already provided the methods to retrieve running container information through standard API.

(4) **Cloud load balancers:**

As far as the cloud load balancers are concerned, two articles have been identified. Google's Maglev[21] is a software load balancer used in Google Cloud Platform(GCP). Maglev uses modern technologies including per flow ECMP and kernel bypass for userspace packet processing. Maglev serves as the GCP's load balancer that is used by the Kubernetes. Maglev is not a product that users can use outside of GCP nor is an open source software, while the users need open source software load balancer that is runnable even in on-premise data centers. Microsoft's Ananta[22] is another software load balancer implementation using ECMP and windows network stack. Ananta can be solely used in Microsoft's Azure cloud infrastructure[22]. The proposed load balancer by the author is different in that it is aimed to be used in every cloud provider and on-premise data centers.

3. Proposed Architecture

Here we discuss a general overview of the proposed load balancer architectures.

3.1 Problems of Kubernetes Cluster

Problems commonly occur when the Kubernetes container management system is used outside of recommended cloud providers(such as GCP or AWS). Figure 1 shows an exemplified Kubernetes cluster. A Kubernetes cluster typically consists of a master and nodes. They can be physical servers or VMs. On the master, daemons that control the Kubernetes cluster are typically deployed. These daemons include, Apiserver, Scheduler, Controller-manager and Etcd. On the nodes, the kubelet daemon will run *pods*, depending the PodSpec information obtained from the apiserver on the master. A *pod* is a group of containers that share same net namespace and cgroups, and is the basic execution unit in a Kubernetes cluster.

When a service is created, the master will schedule where to run *pods* and kubelets on the nodes will launch them accordingly. At the same time, the masters will send out requests to cloud provider API endpoints, asking them to set up external cloud load balancers. The proxy daemon on the nodes will also setup iptables DNAT[13] rules. The Internet traffic will then be evenly distributed by the cloud load balancer to nodes, after which it will be distributed again by the DNAT rules on the nodes to the designated *pods*. The returning packets will follow the exact same route as the incoming ones.

This architecture has the followings problems: 1) Having cloud load balancers whose APIs are supported by the Kubernetes daemons is a prerequisite. There are numerous load balancers that are not supported by the Kubernetes. These include the bare metal load balancers for on-premise data centers. In such cases, users are required to set up the routing manually depending on the infrastructure. However, this approach significantly degrades the portability of container clusters. 2) Distributing the traffic twice, first on the external load balancers and second on each node, complicates the administration of packet routing. Imagine a situation in which the DNAT table on one of the nodes malfunctions. In such a case, only occasional timeouts would be observed, which would make it very difficult to find out which node was malfunctioning.

In short, 1) Kubernetes is optimized only for limited environments where the external load balancers are supported, and 2) the routes incoming traffic follow are very complex. To address these problems, we propose a containerized software load balancer with ECMP redundancy for environments without a cloud load balancer.

3.2 Proposed architecture: a potable load balancer

Figure 2 shows the proposed Kubernetes cluster architecture, which has the following characteristics: 1) Each load balancer itself is run as a *pod* by Kubernetes. 2) Load balancer configurations are dynamically updated based on information about running *pods*. The proposed load balancer can resolve the conventional architecture problems, as follows: Since the load balancer itself is containerized, the load balancer can run in any environment including on-premise data centers, even without external load balancers that is supported by Kubernetes. The incoming traffic is directly distributed to designated *pods* by the load balancer. It makes

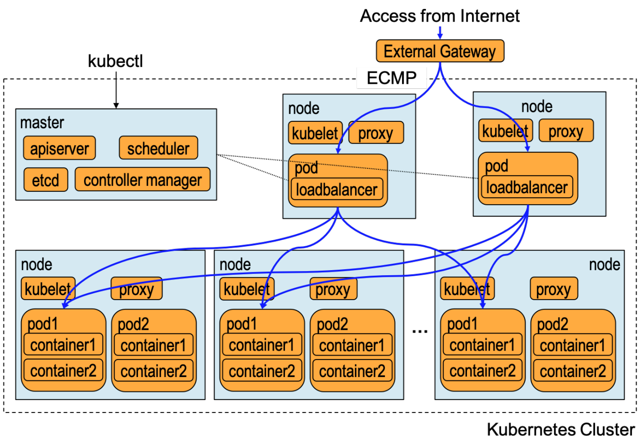


Fig. 2: Kubernetes cluster with proposed load balancer.

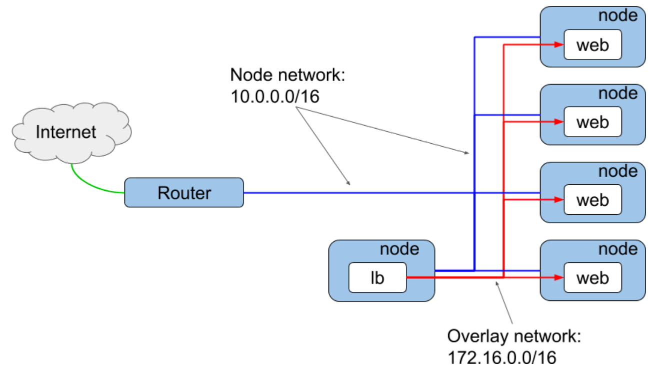


Fig. 3: The network architecture of an exemplified container cluster system.
A load balancer(lb) pod(the white box with "lb") and web pods are running on nodes(the blue boxes). The traffic from the internet are forwarded to the lb pod by the upstream router using the node network, and the distributed to web pods using the overlay network.

the administration, e.g. finding malfunctions, easier.

We designed the proposed load balancer using three components, ipvs, keepalived, and a controller. These components are placed in a Docker container image. The ipvs is a Layer-4 load balancer capability, which is included in the Linux kernel 2.6.0 released in 2003 or later, to distribute incoming Transmission Control Protocol(TCP) traffic to *real servers*[†][4]. For example, ipvs distributes incoming Hypertext Transfer Protocol(HTTP) traffic destined for a single destination IP address, to multiple HTTP servers(e.g. Apache HTTP or nginx) running on multiple nodes in order to improve the performance of web services. Keepalived is a management program that performs health checking for *real servers* and manages ipvs balancing rules in the kernel accordingly. It is often used together with ipvs to facilitate ease of use. The controller is a daemon that periodically monitors the *pod* information on the master, and it performs various actions when such information changes. Kubernetes provides ingress controller framework as the Go Language(Golang) package to implement the controllers. We have implemented a controller program that feeds *pod* state changes to keepalived using this framework.

3.3 Proposed architecture: load balancer redundancy

While containerizing ipvs makes it runnable in any environment, it is essential to discuss how to route the traffic to the ipvs container. We propose redundant architecture using ECMP for load balancer containers usable especially in on-premise data centers. We first explain overlay network briefly to understand requirements for the architecture in (1), then present the proposed architecture with ECMP redundancy in (2). We also present an alternative architecture using VRRP as a comparison in (3), which we think is not as good as the architecture using ECMP.

[†]The term, *real servers* refers to worker servers that will respond to incoming traffic, in the original literature[4]. We will also use this term in the similar way.

(1) **Overlay network**

In order to discuss load balancer redundancy, the knowledge of the overlay network is essential. We briefly explain an abstract concept of overlay network that is common to existing overlay network including flannel[25] and calico[24].

Fig. 3 shows schematic diagram of network architecture of a container cluster system. Suppose we have a physical network(node network) with IP address range of 10.0.0.0/16 and an overlay network with IP address range of 172.16.0.0/16. The node network is the network for nodes to communicate with each other. The overlay network is the network setups for containers to communicate with each other. An overlay network typically consists of appropriate routing tables on nodes, and optionally of tunneling setup using ipip or vxlan. The upstream router usually belongs to the node network. When a container in the Fig. 3 communicates with any of the nodes, it can use its IP address in 172.16.0.0/16 IP range as a source IP, since every node has proper routing table for the overlay network. When a container communicates with the upstream router that does not have routing information regarding the overlay network, the source IP address must be translated by Source Network Address Translation(SNAT) rules on the node the container resides.

The SNAT caused a problem when we tried to co-host multiple load balancer containers for different services on a single node, and let them connect the upstream router directly. This was due to the fact that the BGP agent used in our experiment only used the source IP address of the connection to distinguish the BGP peer. The agent behaved as though different BGP connections from different containers belonged to a single BGP session because the source IP addresses were identical due to the SNAT.

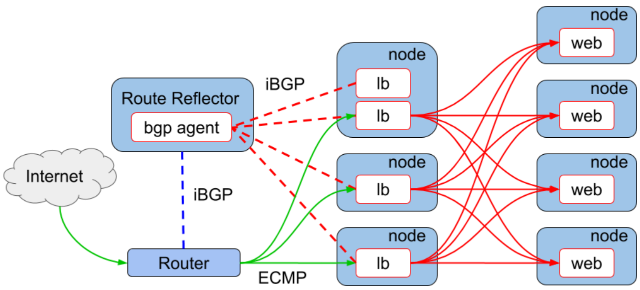


Fig. 4: The proposed architecture of load balancer redundancy with ECMP.
The traffic from the internet is distributed by the upstream router to multiple of lb pods using hash-based ECMP and then distributed by the lb pods to web pods using Linux kernel's ipvs. The ECMP routing table on the upstream router is populated using iBGP.

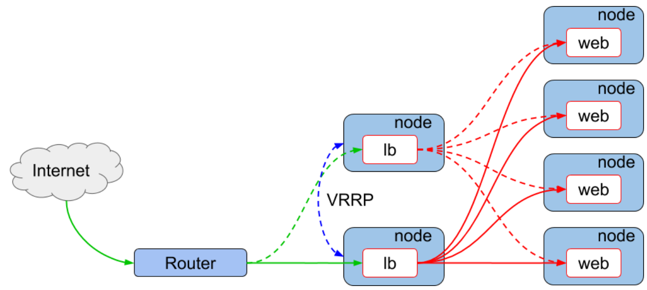


Fig. 5: An alternative redundant load balancer architecture using VRRP.
The traffic from the internet is forwarded by the upstream router to a active lb node and then distributed by the lb pods to web pods using Linux kernel's ipvs. The active lb pod is selected using VRRP protocol.

(2) **Redundancy with ECMP**

Fig. 4 shows our proposed redundancy architecture with ECMP for software load balancer containers. The ECMP is a functionality a router often supports, where the router has multiple next hops with equal cost(priority) to a destination, and generally distribute the traffic depending on the hash of the flow five tuples(source IP, destination IP, source port, destination port, protocol). The multiple next hops and their cost are often populated using the BGP protocol. The notable benefit of the ECMP setup is the fact that it is scalable. All the load balancers that claims as the next hop is active, i.e., all of them are utilized to increase the performance level. Since the traffic from the internet is distributed by the upstream router, the overall throughput is determined by the router after all. However, in practice, there are a lot of cases where this architecture is beneficial. For example, if a software load balancer is capable of handling 1 Gbps equivalent of traffic and the upstream router is capable of handling 10 Gbps, it still is worthwhile launching 10 of the software load balancer containers to fill up maximum throughput of the upstream router.

We place a node with the knowledge of the overlay network as a route reflector, to deal with the complexity due to the SNAT. A route reflector is a network routing component for BGP to reduce the number of peerings by aggregating the routing information[30]. In our proposed architecture we use it as a delegator for load balancer containers towards the upstream router.

By using the route reflector, we can have the following benefits. 1) Each node can accommodate multiple load balancer containers. This was not possible when we tried to directly connect load balancers and the router through SNAT. 2) The router does not need to allow peering connections from random IP addresses that may be used by load balancer containers. Now, the router only need to have the reflector information as the BGP peer definition.

Since we use standard Linux boxes for route reflectors, we can configure them as we like; a) We can make them belong to overlay network so that multiple BGP sessions from a single node can be established. b) We can use a BGP agent that supports dynamic neighbor (or dynamic peer), where one only needs to define the IP range as a peer group and does away with specifying every possible IP that load balancers may use.

The upstream router does not need to accept BGP session from containers with random IP addresses, but only from the router reflector with well known fixed IP address. This may be preferrable in terms of security especially when a different organization administors the upstream router. Although not shown in the Fig. 4, we could also place another route reflector for redundancy.

(3) **Redundancy with VRRP**

Fig. 5 shows an alternative redundancy setup using the VRRP protocol that was first considered by the authors, but did not turn out to be preferable. In the case of VRRP, the load balancer container needs to run in the node net namespace for the following two reasons. 1) When fail over occurs, the new master sends gratuitous Address Resolution Packets(ARP) packets to update the ARP cache of the upstream router and Forwarding Data Base(FDB) of layer 2 swicthes during the transition. Such gratuitous ARP packets should consist of the virtual IP address shared by the load balancers and the MAC address of the node where the new master load balancer is running. Programs that send out gratuitous ARP with node MAC address should be in the node net namespace. 2) Furthermore, the active load balancer sends out periodic advertisement using UDP multicast packet to inform existence of itself. The receiving load balancer in backup state stays calm unless the VRRP advertisement stops for a specified duration of time. The UDP multicast is often unsupported in overlay network used by container cluster environment, and hence the load balancer needs to be able to use the node net namespace. Running containers in the node net namespace loses the whole point of containerization, i.e., they share the node network without separation. This requires the users' additional efforts to avoid conflict in VRRP configuration for multiple services.

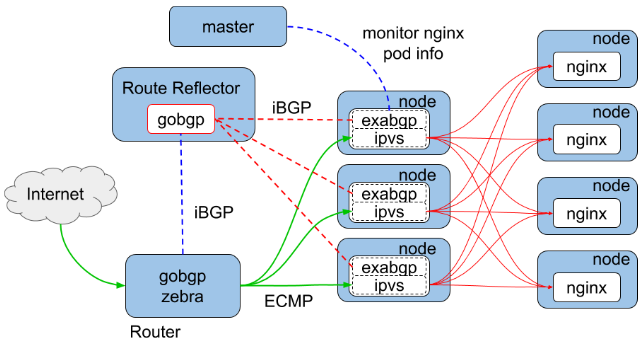


Fig. 6: An experimental container cluster with proposed redundant software balancers.
The master and nodes are configured as Kubernetes's master and nodes on top of conventional Linux boxes, respectively. The route reflector and the upstream router are also conventional Linux boxes.

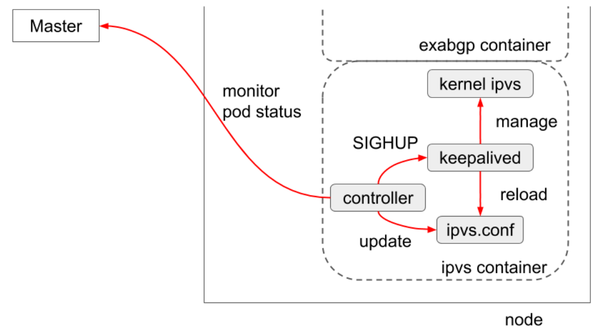


Fig. 7: Implementation of ipvs container.

VRRP programs also support unicast advertisement by specifying IP addresses of peer load balancers before it starts. However, container cluster management system randomly assign IP addresses of containers when it launches them, and it is impossible to know peer IPs in advance. Therefore the unicast mode is not feasible in container cluster environment.

The other drawback compared with the ECMP case is that the redundancy of VRRP is provided in Active-Backup manner. This means that a single software load balancer limits the overall performance of the entire container cluster. Therefore we believe the ECMP redundancy is better than VRRP in our use cases.

4. Implementation

Here we discuss the implementation of the experimental system to prove the concept of our proposed load balancers with ECMP redundancy in detail.

4.1 Experimental system architecture

Fig. 6 shows the schematic diagram of proof of concept container cluster system with our proposed redundant software load balancers. All the nodes and route reflector are configured using Debian 9.5 with self compiled linux-4.16.8 kernel. The upstream router also used conventional linux box using the same OS as the nodes and route reflector. For the Linux kernel to support hash based ECMP routing table we needed to use kernel version 4.12 or later. We also needed to enable kernel config option CONFIG_IP_ROUTE_MULTIPATH[28] when compiling, and set the kernel parameter fib_multipath_hash_policy=1 at run time. In the actual production environment, proprietary hardware with the highest throughput is often deployed, but we could still test some of the required advanced functions by using a Linux box.

Each load balancer pod consists of an exabgp container and an ipvs container. The ipvs container is responsible for distributing the traffic toward the IP address that a service uses, to web(nginx) server pods. The ipvs container monitors the availability of web server pods and manages the load balancing rule appropriately. The exabgp container is responsible for advertising the route toward the IP address that a service uses, to the route reflector. The route reflector aggregates the routing information advertised by load balancer pods and advertise them to the upstream router.

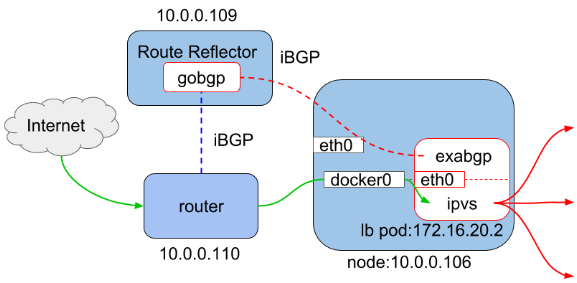
The exabgp is used in the load balancer pods because of the simplicity in setting as static route advertiser. On the other hand, gobgp is used in the router and the route reflector, because exabgp did not seem to support add-path[29] needed for multi-path advertisement and Forwarding Information Base(FIB) manipulation[26]. The gobgp supports the add-path, and the FIB manipulation through zebra[27]. The configurations for the router is summarised in Appendix C.

The route reflector also uses a Linux box with gobgp and overlay network setup. The requirements for the BGP agent on the route reflector are dynamic-neighbours and add-paths features. The configurations for the route reflector is summarised in Appendix B.

4.2 Ipvs container

The proposed load balancer needs to dynamically reconfigure the ipvs balancing rules whenever *pods* are created or deleted. Fig. 7 is a schematic diagram of ipvs container to show the dynamic reconfiguration of the ipvs rules. Two daemon programs, controller and keepalived, run in the container are illustrated. The keepalived manages Linux kernel's ipvs rules depending on the ipvs.conf configuration file. It is also capable of health-checking the liveliness of *real server*, which is represented as a combination of the IP addresses and port numbers of the target *pods*. If the health check to a *real server* fails, keepalived will remove that *real server* from the ipvs rules.

The controller monitors information concerning the running *pods* of a service in the Kubernetes cluster by consulting the apiserver running in the master. Whenever *pods* are created or deleted, the controller notices the change and automatically regenerate an appropriate ipvs.conf and issue SIGHUP to keepalived. Then, keepalived will reload the



(a)

```
[BGP announcement]
    route 10.1.1.0/24 next-hop 10.0.0.106

[Routing in node net namespace]
    ip netns exec node \
            ip route replace 10.1.1.0/24 dev docker0

[Accept as local]
    ip route add local 10.1.1.0/24 dev eth0
```

(b)

Fig. 8: (a) Network path by the exabgp container. (b) Required settings in the exabgp container.

ipvs.conf and modify the kernel's ipvs rules accordingly. The actual controller[16] is implemented using the Kubernetes ingress controller[3] framework. By importing existing Golang package, "k8s.io/ingress/core/pkg/ingress", we could simplify the implementation, e.g. 120 lines of code.

Keepalived and the controller are placed in the docker image of ipvs container. The ipvs is the kernel function and namespace separation for container has already been supported in the recent Linux kernel.

Configurations for capabilities were needed when deploying the ipvs container: adding the CAP_SYS_MODULE capability to the container to allow the kernel to load required kernel modules inside a container, and adding CAP_NET_ADMIN capability to the container to allow keepalived to manipulate the kernel's ipvs rules. For the former case, we also needed to mount the "/lib/module" of the node's file system on the container's file system.

4.3 BGP software container

In order to implement the ECMP redundancy, we also containerized exabgp using Docker. Fig.8 (a) shows a schematic diagram of the network path realized by the exabgp container. We used exabgp as the BGP advertiser as mentioned earlier. The traffic from the Internet is forwarded by ECMP routing table on the router to the node, then routed to ipvs container.

Fig.8 (b) summarises some key settings required for the exabgp container. In BGP announcements the node IP address, 10.0.0.106 is used as the next-hop for the IP range 10.1.1.0/24. Then on the node, in order to route the packets toward 10.1.1.0/24 to the ipvs container, a routing rule to the dev docker0 is created in the node net namespace. A routing rule to accept the packets toward those IPs as local is also required in the container net namespace. A configuration of exabgp is shown in Appendix A.

5. Evaluation

In order to understand the feasibility of the proposed load balancer architecture, we evaluated it with the following three criteria; (1) Basic functionality and Portability: We evaluated the load balancer functionality using physical servers in on-premise data center and compared performance level with existing iptables DNAT and nginx as a load balancer. We also carried out the same performance measurement in GCP and AWS to show the containerized ipvs load balancer is runnable even in the cloud environment. (2) Redundancy and Scalability: We evaluated ECMP functionality by watching routing table updates on the router when the new load balancer is added or removed. We also evaluated the performance level by changing the number of load balancers.

The following subsections explain the evaluation in detail.

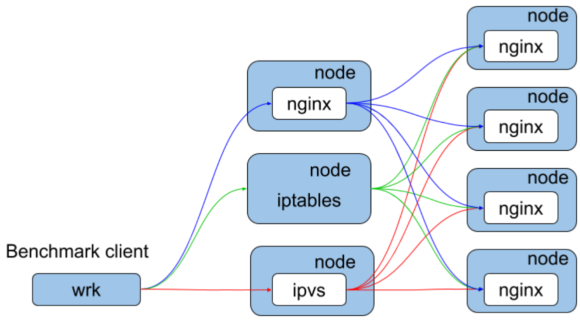
5.1 Basic functionality and portability

Throughput measurements were carried out in order to examine the basic functionality of the containerized ipvs load balancer. Fig. 9 shows the schematic diagram of the throughput measurement and summarizes experimental conditions. We measured the performance of the load balancers using a benchmark program called wrk[17]. Multiple nginx *pods* are deployed on multiple nodes as web servers in the Kubernetes cluster. In each nginx *pod*, single web server thread that returns the IP address of the *pod* is running. We then set up the ipvs, iptables DNAT, and nginx load balancers on one of the nodes for each and performed the throughput measurement. The throughput is measured by sending out HTTP requests from the wrk towards a load balancer and by counting the number of responses the benchmark host received as shown in Fig. 9 (a).

Fig. 9 (b) shows an example of the command-line for wrk and the corresponding output. The command-line in Fig. 9 (b) will generate 40 wrk program threads and allow those threads to send out a total of 800 concurrent HTTP requests over the period of 30 seconds. The output example shows information including per-thread statistics, error counts, throughput in [Request/sec] and data rate in [Transfer/sec]

Fig. 9 (c) shows hardware and software configuration used in our experiments. We used a total of eight servers; six servers for Nodes, one for the load balancer and one for the benchmark client, with all having the same hardware specifications. The software versions used for Kubernetes, web server and load balancer *pods* are also summarized in the figure.

Fig. 10 (a) shows the throughput of the proposed ipvs



(a)

```
[Command line]
 wrk -c800 -t40 -d30s http://172.16.72.2:8888/
-c: concurrency, -t: # of thread, -d: duration

[Output example]
 Running 30s test @ http://10.254.0.10:81/
  40 threads and 800 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    15.82ms   41.45ms   1.90s    91.90%
    Req/Sec     4.14k   342.26     6.45k    69.24%
  4958000 requests in 30.10s, 1.14GB read
  Socket errors: connect 0, read 0, write 0, timeout 1
Requests/sec: 164717.63
Transfer/sec:     38.86MB
```

(b)

```
[Hardware Specification]
  CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading)
  Memory: 32GB
  NIC: Broadcom BCM5720 Giga bit
 (Node x 6, LB x 1, Client x 1)

[Node Software]
  OS: Debian 8.7, linux-3.16.0-4-amd64
  Kubernetes v1.5.2
  flannel v0.7.0
  etcd version: 3.0.15

[Container Software]
  Keepalived: v1.3.2 (12/03,2016)
  nginx : 1.11.1(load balancer), 1.13.0(web server)
```

(c)

Fig. 9: (a) Benchmark setup. (b) Benchmark commandline and output example. (c) HW SW spec.

container load balancer. The performance of the nginx and the iptables DNAT as the load balancers are also presented for comparison. As we increased the number of the nginx pods(web servers) from 1 to around 14, the throughput increased almost linearly and after which it saturated. The increase indicates that the load balancer functions properly because it increased throughput by distributing HTTP requests to multiple of the web servers. The saturated performance level indicates the maximum performance of the load balancer, which could be determined either by network bandwidth, hardware performance for load balancer or that for the benchmark client. In this specific experiment, the performance level was limited by the 1 Gbps bandwidth of experimental network[23], which is revealed by packet level analysis using tcpdump. While nginx did not show any benefit as the load balancer, the performance of the ipvs load balancer container showed equivalent performance level as the un-containerized iptables DNAT. This means that our proposed ipvs container load balancer is at least as good as the un-containerized iptables' load balancing in the 1 Gbps network.

Fig. 10 (b) and Fig. 10 (c) show the load balancer performance levels that are measured in GCP and AWS, respectively. These are aimed to show that our proposed load balancer can be run in cloud environments and also functions properly.

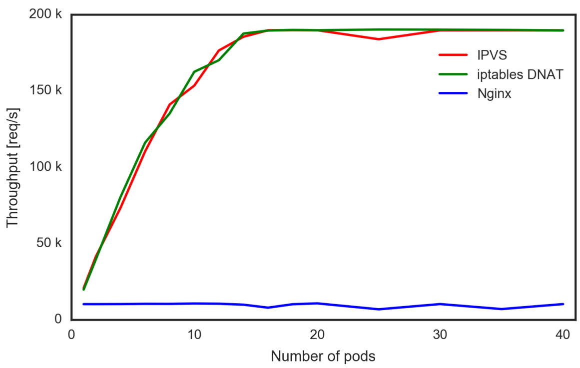
Both results show similar characteristics as the experiment in an on-premise data center in Fig. 10 (a), where throughput increased linearly to a certain level that is determined by either network speed or machine specifications. Since in the cases of cloud environments we can easily change the machine specifications, especially CPU counts, we measured throughput with several conditions of them. From the first look of the results, since changing CPU counts changed the load balancer's throughput saturation levels, we thought VM's computation power limited the performance levels. However, since there are cases in the cloud environment, where changing the VM types or CPU counts also changes the network bandwidth limit, a detailed analysis is further required in the future to clarify which factor limits the throughput. Still, we can say that the load balancers can be run in GCP and AWS, and function properly until they reach the infrastructure limitations.

5.2 Redundancy and Scalability

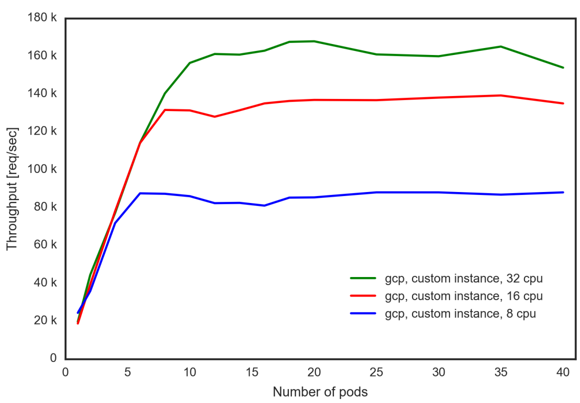
The ECMP redundancy is expected to give us the load balancer redundancy and scalability since all the load balancer containers act as active. We evaluated the behavior of the ECMP routing table updates, by changing the number of the load balancers. After that, we also measured the throughput when multiple of the ipvs container load balancers are deployed.

Fig. 11a shows the schematic diagram of the experimental setup and also summarizes hardware and software specifications. Notable differences from the previous throughput experiment in Fig. 9 are as follows; 1) Each load balancer pods now consists of both an ipvs container and an exbgp container. 2) The routing table at the router, which also acts as a benchmark client, is updated by BGP protocol through a route reflector. 3) The router's NIC has been changed to 10 Gbps card since now we have multiple of ipvs container load balancers that will use up 1 Gbps bandwidth. 4) All of the software has been updated to the most recent versions at the time of the experiment.

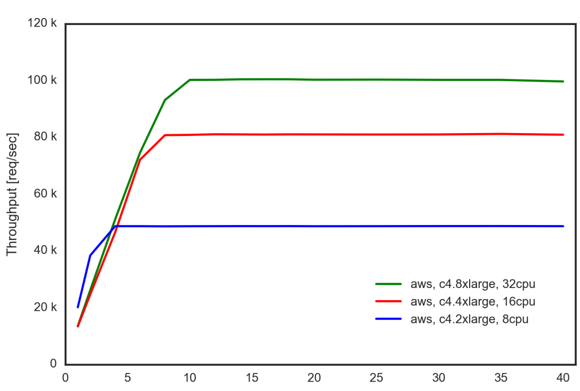
First, we examined ECMP functionality by watching the



(a) Performance of the load balancer container.



(b) GCP

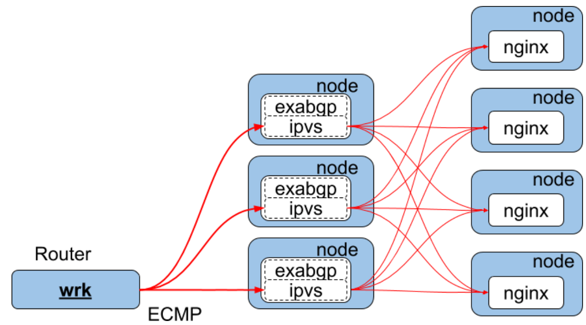


(c) AWS with Node x 6, Client x 1, Load balancer x 1. Custom instance.

Fig. 10: Performance of the proposed load balancer in Cloud environment. (a) GCP: Web:

routing table on the router. Fig. 12 (a) shows the routing table entry on the router when a single load balancer pod exists. From this line, we can tell that packets toward 10.1.1.0/24 are forwarded to 10.0.0.106 where the load balancer pod is running. It also shows that this routing rule is controlled by zebra.

When the number of the load balancer pods is increased to three, we can see the routing table entry in Fig. 12 (b). We have three next hops towards 10.1.1.0/24 each of which



(a)

```
[Hardware Specification]
  CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading)
  Memory: 32GB
  NIC: Broadcom BCM5720 Giga bit
 (Node x 6, Client x 1)

  CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading)
  Memory: 32GB
  NIC: Intel X550
 (LB x 1)

[Node Software]
  OS: Debian 9.5, linux-4.16.8
  Kubernetes v1.5.2
  flannel v0.7.0
  etcd version: 3.0.15

[Container Software]
  Keepalived: v1.3.2 (12/03,2016)
  nginx : 1.11.1(load balancer), 1.13.0(web server)
```

(b)

Fig. 11: (a) Benchmark setup. (b) Benchmark commandline and output example. (c) HW SW spec.

being the node where the load balancer pods are running. The weights of the three next-hops are all 1. The update of the routing entry was almost instant as we increased the number of the load balancers.

Fig. 12 (c) shows the case where we additionaly started new service with two load balancer pods with service addresses in 10.1.2.0/24 range. We could accommodate two different services toward different IP addresses, one with three load balancers and the other with two load balancers on a group of nodes(10.0.0,105,10.0.0,106,10.0.0,107). The update of the routing entry was almost instant as we started the load balancers for the second service.

We also carried out throughput measurement to show our proposed architecture increases as we increase the number of the load balancers.

5.3 Resource Consumption

6. Conclusions

In this paper, we proposed a portable load balancer with

```
10.1.1.0/24 via 10.0.0.106 \
                              dev eth0 proto zebra metric 20
```

(a) Routing table entry with single load balancer pod.

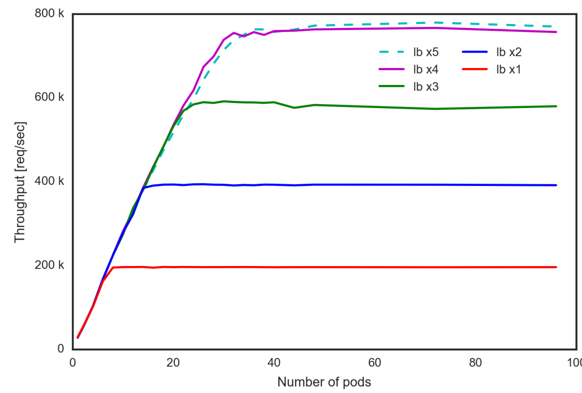
```
10.1.1.0/24 proto zebra metric 20
        nexthop via 10.0.0.105  dev eth0 weight 1
        nexthop via 10.0.0.106  dev eth0 weight 1
        nexthop via 10.0.0.107  dev eth0 weight 1
```

(b) Routing table entry with three load balancer pods.

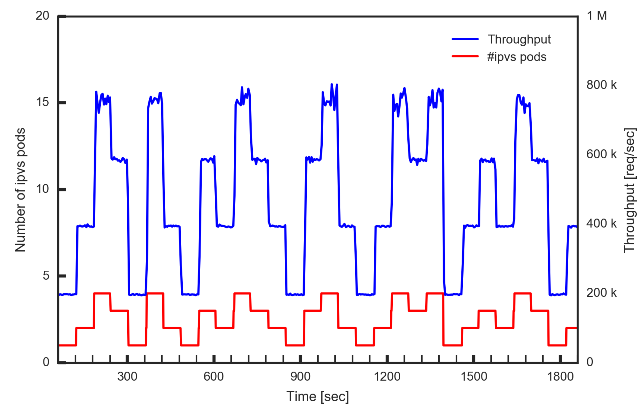
```
10.1.1.0/24 pro to zebra metric 20
        nexthop via 10.0.0.107  dev eth0 weight 1
        nexthop via 10.0.0.105  dev eth0 weight 1
        nexthop via 10.0.0.106  dev eth0 weight 1
10.1.2.0/24 proto zebra metric 20
        nexthop via 10.0.0.107  dev eth0 weight 1
        nexthop via 10.0.0.106  dev eth0 weight 1
```

(c) Routing table entry with two services with two, three load balancer pods respectively.

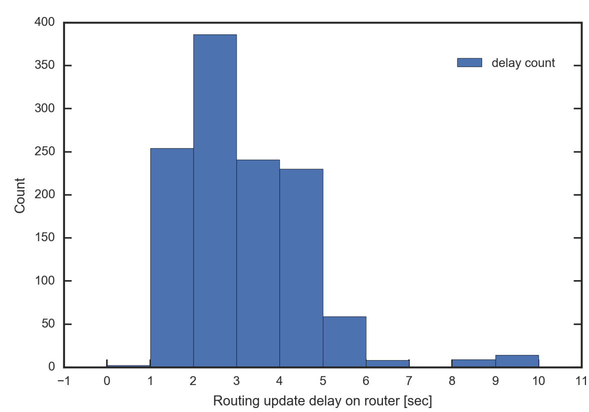
Fig. 12: ECMP routing tables.



(a) Caption 1



(b) Caption 2



(c) Caption 2

Fig. 13: Scalability of portable load balancer with ECMP redundancy.

ECMP redundancy for the Kubernetes cluster systems that is aimed at facilitating migration of container clusters for web services. We implemented an experimental web cluster system with multiple of load balancers and web servers using Kubernetes and OSSs on top of standard Linux boxes to prove the functionality of the proposed architecture. We conducted performance measurements and found that the ipvs based load balancer in container improved the portability of the Kubernetes cluster system while it showed the comparable performance levels as the existing iptables DNAT based load balancer. We also examined that the ECMP redundant feature properly functioned by monitoring the routing table of the upstream router and packet flow as we periodically accessed the web cluster. For future work we plan to run performance measurements for the scalability of our system and also improve performance of a single software load balancer on standard Linux box using Xpress Data Plane(XDP) technology.

References

[1] The Kubernetes Authors, Kubernetes | Production-Grade Container Orchestration, https://kubernetes.io/, accessed July 14, 2017

[2] The Kubernetes Authors, Federation, `https://kubernetes.io/docs/concepts/cluster-administration/federation/`, accessed July 14, 2017

[3] The Kubernetes Authors, Ingress Resources | Kubernetes, `https://kubernetes.io/docs/concepts/services-networking/ingress/`, accessed July 14, 2017

[4] Zhang W., Linux virtual server for scalable network services, Ottawa Linux Symposium, 2000

[5] Michael Pleshakov, NGINX and NGINX Plus Ingress Controllers for Kubernetes Load Balancing, `https://www.nginx.com/blog/nginx-plus-ingress-controller-kubernetes-load-balancing/`, accessed July 14, 2017

[6] NGINX Inc., NGINX Ingress Controller, `https://github.com/nginxinc/kubernetes-ingress`, accessed July 14, 2017

[7] Prashanth B, Mike Danese, Bowei Du, kube-keepalived-vip, `https://github.com/kubernetes/contrib/tree/master/keepalived-vip`, accessed July 14, 2017

[8] Alexandre Cassen, Keepalived for Linux, http://www.keepalived.org/, accessed July 14, 2017
[9] Andrey Sibiryov, GORB Go Routing and Balancing, https://github.com/kobolog/gorb, accessed July 14, 2017
[10] Marttila Tero, Design and Implementation of the clusterf Load Balancer for Docker Clusters, Master's Thesis, Aalto University, October, 2016
[11] Docker Core Engineering, Docker 1.12: Now with Built-in Orchestration, https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/, accessed July 14, 2017
[12] Docker Inc, Use swarm mode routing mesh | Docker Documentation, https://docs.docker.com/engine/swarm/ingress/, accessed July 14, 2017
[13] Martin A. Brown, Guide to IP Layer Network Administration with Linux, http://linux-ip.net/html/index.html, accessed July 14, 2017
[14] CoreOS Inc, etcd | etcd Cluster by CoreOS, https://coreos.com/etcd, accessed July 14, 2017
[15] HashiCorp, Consul by HashiCorp, https://www.consul.io/, accessed July 14, 2017
[16] K. Takahashi, ktaka-ccmp/ipvs-ingress, https://doi.org/10.5281/zenodo.826894, July, 2017
[17] Will Glozer, wrk - a HTTP benchmarking tool, https://github.com/wg/wrk, accessed July 14, 2017
[18] Marmol, V., Jnagal, R., and Hockin, T. (2015). Networking in containers and container clusters. Proceedings of netdev 0.1, February.
[19] Hinden, R. (2004). Virtual router redundancy protocol (VRRP) (No. RFC 3768).
[20] Moy, J. (1997). OSPF version 2 (No. RFC 2178).
[21] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., ... and Hosein, J. D. (2016, March). Maglev: A Fast and Reliable Software Network Load Balancer. In NSDI (pp. 523-535).
[22] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., ... and Kim, C. (2013). Ananta: Cloud scale load balancing. ACM SIGCOMM Computer Communication Review, 43(4), 207-218.
[23] Takahashi, K., Aida, K., Tanjo, T., and Sun, J. (2018, January). A Portable Load Balancer for Kubernetes Cluster. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (pp. 222-231). ACM.
[24] Project Calico, Secure Networking for the Cloud Native Era, https://www.projectcalico.org/, 2018, accessed August 14, 2018
[25] Coreos Inc., coreos/flannel, https://github.com/coreos/flannel, accessed August 14, 2018
[26] Exa-Networks, Exa-Networks/exabgp, https://github.com/Exa-Networks/exabgp, accessed August 14, 2018
[27] Osrg, osrg/gobgp, https://github.com/osrg/gobgp/blob/master/docs/sources/zebra.md, accessed August 14, 2018
[28] ip-sysctl.txt, https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt, accessed August 14, 2018
[29] Walton, D., Retana, A., Chen, E., and Scudder, J. (2016). Advertisement of multiple paths in BGP (No. RFC 7911).
[30] Bates, T., Chen, E., and Chandra, R. (2006). Bgp route reflection: An alternative to full mesh internal bgp (ibgp) (No. RFC 4456).
[31] Thaler, D., and Hopps, C. (2000). Multipath issues in unicast and multicast next-hop selection (No. RFC 2991).

Kimitoshi Takahashi received the B.S. degree from Tokyo University in 1993. During 1993–2002, he stayed at Fujitsu Labs. Limited and worked in the field of semiconductor lithography. Form 1999 to 2000 he stayed at Stanford University as a visiting scholar. In 2002, he founded Cluster Computing Inc. and stayed as CEO since then. Since 2016, he has also been Ph.D candidate at the Graduate University for Advanced Studies.

Kento Aida received Dr. Eng. in electrical engineering from Waseda University in 1997. He became a research associate at Waseda University in 1992. He joined Tokyo Institute of Technology and became a research scientist at the Department of Mathematical and Computing Sciences in 1997, an assistant professor at the Department of Computational Intelligence and Systems Science in 1999, and an associate professor at the Department of Information Processing in 2003, respectively. He is now a professor at National Institute of Informatics from 2007. He was also a researcher at PRESTO in Japan Science and Technology Agency (JST) from 2001 through 2005, a research scholar at the Information and Computer Sciences Department in University of Hawai'i in 2007, and a visiting professor at the Department of Information Processing in Tokyo Institute of Technology from 2007 through 2016.

Tomoya Tanjo

Jingtao Sun received his M.E degrees in computer science from Tokyo University of Technology, Japan in 2013, and received his Ph.D degrees in informatics from the Graduate University for Advanced Studies, Japan in 2016. Since 2016, he joined National Institute of Informatics and became a project researcher at the Information Systems Architecture Research Division. During 2016–2018, he visited Florida University and University of Maryland, Baltimore County as a visiting researcher. His current research insterests include distributed systems, cloud computing, and Internt of Things. He is a member of JPSJ, CCF, IEEE and ACM.

Kazushige Saga

Appendix A: Exabgp configuration on the load balancer container.

exabgp.conf:

```
neighbor 10.0.0.109 {
 description "peer1";
 router-id 172.16.20.2;
 local-address 172.16.20.2;
 local-as 65021;
 peer-as 65021;
 hold-time 1800;
   static {
     route 10.1.1.0/24 next-hop 10.0.0.106;
   }
}
```

Appendix B: Gobgpd configuration on the route reflector.

gobgp.conf:

```
global:
```

```
  config:
    as: 65021
    router-id: 10.0.0.109
    local-address-list:
    - 0.0.0.0 # ipv4 only
  use-multiple-paths:
    config:
      enabled: true

peer-groups:
  - config:
      peer-group-name: k8s
      peer-as: 65021
    afi-safis:
      - config:
          afi-safi-name: ipv4-unicast

dynamic-neighbors:
  - config:
      prefix: 172.16.0.0/16
      peer-group: k8s

neighbors:
  - config:
      neighbor-address: 10.0.0.110
      peer-as: 65021
    route-reflector:
      config:
        route-reflector-client: true
        route-reflector-cluster-id: 10.0.0.109
    add-paths:
      config:
        send-max: 255
        receive: true
```

Appendix C: Gobgpd and zebra configurations on the router.

gobgp.conf:

```
global:
  config:
    as: 65021
    router-id: 10.0.0.110
    local-address-list:
    - 0.0.0.0

  use-multiple-paths:
    config:
      enabled: true

neighbors:
    - config:
        neighbor-address: 10.0.0.109
        peer-as: 65021
      add-paths:
        config:
          receive: true

zebra:
  config:
    enabled: true
    url: unix:/run/quagga/zserv.api
    version: 3
    redistribute-route-type-list:
      - static
```

zebra.conf:

```
hostname Router
log file /var/log/zebra.log
```