PAPER

A Portable Load Balancer with ECMP Redundancy for Container Clusters

Kimitoshi TAKAHASHI[†], *Nonmember and* **Kento AIDA**[††], *Member*

SUMMARY IEICE (The Institute of Electronics, Information and Communication Engineers) provides a LaTeX 2ε class file, named for the IEICE Transactions. This document describes how to use the class file, and also makes some remarks about typesetting a manuscript by using the LaTeX 2ε. The design is based on LaTeX 2ε.

key words: *redundancy, ECMP, load balancer, container, BGP*

1. Introduction

Recently, Linux containers have drawn significant amount of attention because they are lightweight, portable, and repeatable. 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 repeatable 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 variety of purposes. For example disaster recovery, cost performance improvemets, legal compliance, and shortening the geographical distance to customers are the main concerns for web service providers in e-commece, gaming, Financial technology(Fintech) and Internet of Things(IoT) field.

Kubernetes[1], which is one of the 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. 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. The Kubernetes does not provide generic ways to route the traffic from the internet into container cluster and is heavily dependent on external load balancers that are set up on the fly by cloud providers through their application protocol interfaces (APIs). These external 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)[2], [3] 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 external load balancers, we have proposed a containerized software load balancer that is run by Kubernetes as a part of web services consisting of container cluster in the previous work[4]. It enables a user to deploy a web service on different environments without modification easily because the web service itself includes load balancers.

We containerized Linux kernel's Internet Protocol Virtual Server (IPVS)[5] Layer 4 load balancer using an existing Kubernetes ingress[6] framework, as a proof of concept. We also proved that our approach does not significantly deteriorate 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, in our previous work, we have not discussed the redundancy of such load balancers. Routing traffic to one of the load balancers while keeping redundancy in the container environment is a complex issue, because standard Layer 2 rendandacy protocols, e.g. Virtual Router Redundancy Protocol(VRRP) or OSPF[7] that uses multicast, can not be used in many cases. Further more, providing uniform methods independent of various cloud environments

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 National Institute of Infomatics and The Graduate University for Advanced Studies, Chiyoda-ku, Tokyo, Japan.

DOI: 10.1587/trans.E0.??.1

Copyright © 200x The Institute of Electronics, Information and Communication Engineers

and on-premise datacenter is much more difficult.

In this paper, we propose a software load balancer with ECMP redundancy for container cluster environment without a cloud load balancer. We containerize an open source BGP software, exabgp[], and launch it with containerized Linux kernels IPVS load balancer in a single pod using Kubernetes. We set up a redundant load balancer cluster using such pods and evaluate its functionality. We also measure preliminary performance of such a load balancer cluster.

The contributions of this paper are as follows: 1) We propose a portable software load balancer with ECMP redundacy for container cluster environments without cloud load balancers. 2) We implement such a load balancer using existing Open Source Software(OSS). 2) We demonstrate the basic functionality, i.e., redundancy and scalability of such s load balancers. Proving that a user can design and implement a scalable web service using OSS tools by himself will benefit the web service to migrate outside of the major cloud providers.

The rest of the paper is organized as follows. Section 2 highlights 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 **??** will explain existing architecture problems and propose our solutions. In Section **??**, experimental conditions and the parameters that we considered to be important in our experiment will be described in detail. Then, we will show our experimental results and discuss the obtained performance characteristics in Section **??**, 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[8] 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[9], [10] utilizes the ingress[6] capability of Kubernetes, to implement 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[11] project is trying to use Linux kernel's IPVS[5] load balancer capabilities by containerizing the keepalived[12]. The kernel IPVS function is set up in the host OS's net name spaces 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 name spaces 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[13], [14] also uses IPVS for internal load balancing, but it is also considered as part of Docker swarm infrastructure, and thus lacks 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[15] and clusterf[16] 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[17] and HashiCorp's Consul[18]. 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[19] is a software load balancer used in Google Cloud Platform(GCP)[20]. 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 can be solely used in GCP, and the users need open source software load balancer that is runnable even in on-premise data centers. Microsoft's Ananta[21] is another software load balancer implementation using ECMP and windows network stack. Ananta can be solely used in Microsoft's Azure cloud infrastructure[21]. 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.

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

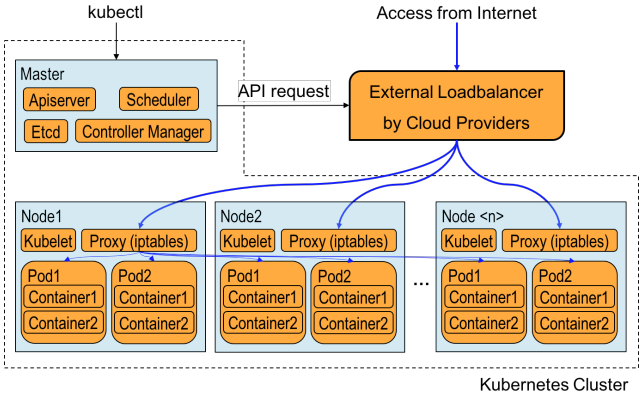

Fig. 1 Conventional architecture of a Kubernetes cluster.

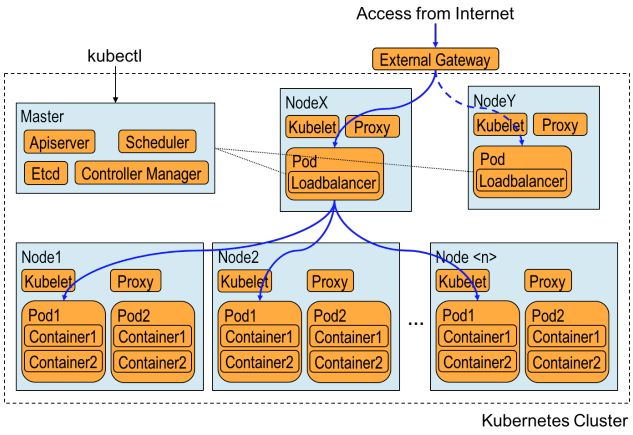

Fig. 2 Kubernetes cluster with proposed load balancer.

3. Proposed Architecture

Here we discuss the overall architecture of the proposed load balancers.

3.1 A containerized Load balancer

3.1.1 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 name space 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 load balancers. The proxy daemon on the nodes will also setup iptables DNAT[2] rules. The Internet traffic will then be evenly distributed by the external 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 external load balancers whose APIs are supported by the Kubernetes daemons is a prerequisite. There are numerous load balancers which is not supported by the Kubernetes. These include the bare metal load balancers for on-premise data centers. In such cases, a user could manually setup the routing table on the gateway so that the traffic would be routed to one of the nodes. Then the traffic would be distributed by the DNAT rules on the node to the designated *pods*. However, this approach would require complicated network configuration and significantly degrade 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 can be used only in limited environments where the external load balancers are supported, and 2) the routes incoming traffic follow are very complex.

3.1.2 Proposed architecture with a potable load balancer

In order to address these problems, we propose a containerized software load balancer that is deployable in any environment even if there are no external load balancers.

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

3.2 Load balancer redundancy

3.2.1 Overlay network

In order to discuss redundancy, the knowledge of the overlay network is essential. We briefly explain an abstract concept of overlay network that is common to existing overlay

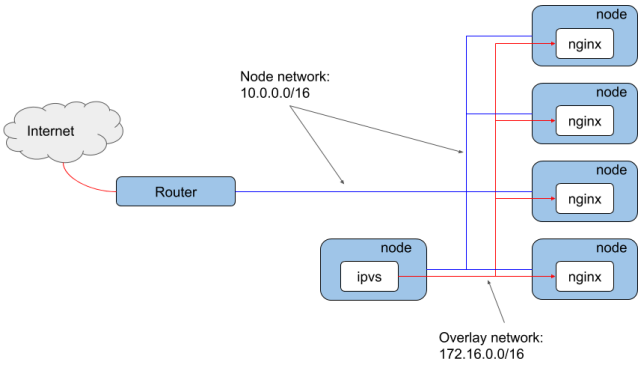

Fig. 3 Network architecture of an exemplified container cluster system. An overlay networks is ....

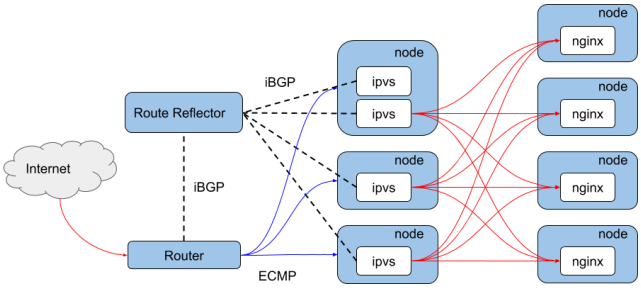

Fig. 4 Network architecture of an exemplified container cluster system. An overlay networks is ....

network including flannel[22] and calico[23].

Fig. 3 shows schematic diagram of network architecture of a container cluster system. An overlay network consists of appropriate routing tables on nodes, and optionally of tunneling setup using ipip or vxlan. The overlay network is the network for containers to communicate with each other. The node network is the network for nodes to communicate with each other. The upstream router usually belongs to the node network. When the ipvs container in the Fig. 3 communicates with the other nodes, the nodes can properly route the packets because they have the routes to 172.16.0.0/16 in their routing table, which is a part of overlay network setups. When a container communicates with the router that has no knowledge of the overlay network, the souce IP address must be translated by Source Network Address Translation(SNAT) rules on each node.

3.2.2 Redundancy with ECMP

The Equal Cost Multi-Path(ECMP) is a functionality a router may support, 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 5 tuples(source IP, destination IP, source port, destination port, protocol). The multiple next hops and their cost are often populated using the Border Gateway Protocol(BGP) protocol.

Fig. 4 shows our proposed redundancy architecture for software load balancers. We propose to use a node with the knowledge of overlay network as a Route Reflector(RR),

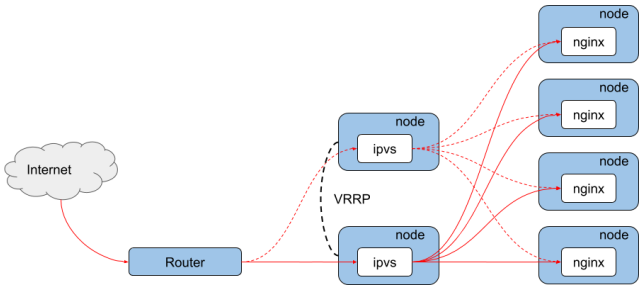

Fig. 5 Network architecture of an exemplified container cluster system. An overlay networks is ....

in other to alleviate the following problems. 1) If we were to setup BGP peering directly between a load balancer and the router, the IP address of the container should be translated using SNAT rules by the nodes, since the router has no knowledge of the overlay network and cannot send out returning packet correctly. However, if we use SNAT, the router only see the node IP address as the peer, and hence the router cannot set up multiple BGP session from a single node. This problem poses limitation that a single node can accommodate only one load balancer at most.

2) Also, if we were to setup BGP peering directly between a load balancer and the router, the router should support so-called dynamic peering(or dynamic neighbor) functions, where one does not have to write exact IP address of peers in the configuration. This function is essential because one can not predict exact IP addresses of the load balancer containers before he launches them. Even if the router supports such functions, the administrator of the upstream router may not like accepting dynamic peering from unknown random IP addresses for security reasons.

By introducing route reflectors we can configure them as we like, e.g., we can make them so that BGP sessions with load balancers are set up dynamically, and multiple BGP sessions from a single node can be established. The upstream router does not need to accept BGP session from containers with random IP addresses, but only from the Router Reflecotr with well known fixed IP address. Although not shown in the Fig. 4, we could also place another Route Reflector for redundancy.

The notable benefit of the ECMP setup is the fact that it is scalable. All load balancer that claims as the next hop is active, i.e., the traffic is forwarded to them depending on the flow tuple. The traffic is distributed by the upstream router. Hence the overall throughput is determined by the router. 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 is worthwhile launching 10 of the software load balancers to fill up maximum throughput of the upstream router.

3.2.3 Redundancy with VRRP

Fig. 5 shows an alternative redundancy setup using the VRRP protocol. In the case of VRRP, the load balancer container

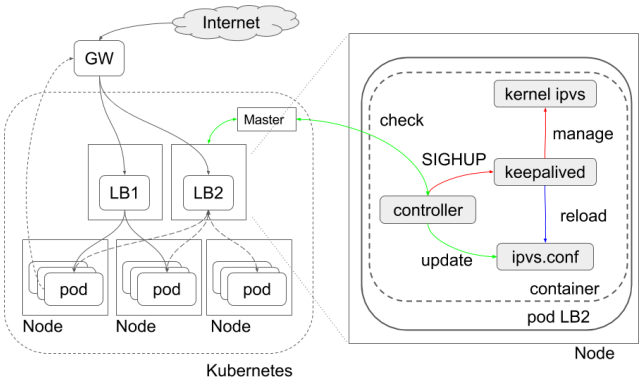

Fig. 6 Implementation

should be able to use the node net namespace for the following two reasons. 1) When failover occurs, the new master sends gratuitous Adress Resolution Packets(ARP) packets to update the ARP cache of the upstream router during the transition. Such gratuitous ARP packets should consist of the virtual IP address of the load balancer and the MAC address of the node where the new master load balancer is running.

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.

VRRP programs also support unicast advertisement by specifying IP addresses of peer load balancers when it starts. However, container cluster management system randomly assign IP addresses of containers when it launches them. 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.

4. Implementation

4.1 IPVS containerization

The proposed load balancer needs to dynamically reconfigure the IPVS balancing rules whenever *pods* are created/deleted. Figure 6 is a schematic diagram to show the dynamic reconfiguration of the IPVS rules. The right part of the figure shows the enlarged view of one of the nodes where the load balancer pod(LB2) is deployed. Two daemon programs, controller and keepalived, run in the container inside the LB2 pod 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 on the master. Whenever *pods* are created or deleted, the controller will automatically regenerate an appropriate ipvs.conf and issue SIGHUP to keepalived. Then, keepalived will reload the ipvs.conf and modify the kernel's IPVS rules accordingly. The actual controller[24] is implemented using the Kubernetes ingress controller[6] framework. By importing existing Golang package, "k8s.io/ingress /core/pkg/ingress", we could simplify the implementation, e.g. 120 lines of code.

Configurations for capabilities were needed in the implementation: 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.

4.2 BGP software containerization

4.3 Proof of concept system architecture

5. Evaluation

5.1 Functionality

5.1.1 Redundancy

5.1.2 Load balancing

5.2 Performance

6. Conclusions

In this paper, we proposed a portable software load balancer that has the following features, 1) runnable as a Linux container, 2) redundancy with ECMP technique, for the container cluster systems.

Our load balancer aims at facilitating migration of container clusters for web services. We implemented a containerized software load balancer that is run by Kubernetes as a part of container cluster, using Linux kernel's IPVS.

To discuss the feasibility of the proposed load balancer, we built a Kubernetes cluster system and conducted performance measurements. Our experimental results indicate that the IPVS based load balancer in container improves the portability of the container cluster while it shows the similar performance levels as the existing iptables DNAT based load balancer.

We also started to the implementation of a novel software load balancer using recently introduced Linux kernel's XDP infrastructure. While it is in a preliminary stage of the development, essential functions and design issues have

been already clarified. They will be presented at the time of the presentation.

7. Future work

We leave the following for the future work: 1) Performance measurement of XDP load balancer 2) Implementation of consistent hashing balancing algorithm 3) Containerization of XDP load balancer These works are ongoing and will be presented when they are available.

References

[1] T.K. Authors, "Kubernetes | production-grade container orchestration," 2017.
[2] Martin A. Brown, "Guide to IP Layer Network Administration with Linux," 2007.
[3] V. Marmol, R. Jnagal, and T. Hockin, "Networking in Containers and Container Clusters," Netdev, 2015.
[4] K. Takahashi, K. Aida, T. Tanjo, and J. Sun, "A portable load balancer for kubernetes cluster," Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp.222–231, ACM, 2018.
[5] W. Zhang, "Linux virtual server for scalable network services," Ottawa Linux Symposium, 2000.
[6] T.K. Authors, "Ingress resources | kubernetes," 2017.
[7] J. Moy, "Ospf version 2," 1997.
[8] T.K. Authors, "Federation," 2017.
[9] M. Pleshakov, "Nginx and nginx plus ingress controllers for kubernetes load balancing," Dec. 2016.
[10] N. Inc., "Nginx ingress controller," 2017.
[11] B.D. Prashanth B, Mike Danese, "kube-keepalived-vip," 2016.
[12] A. Cassen, "Keepalived for linux."
[13] D.C. Engineering, "Docker 1.12: Now with built-in orchestration! - docker blog," 2016.
[14] Docker Inc, "Use swarm mode routing mesh | Docker Documentation," 2017.
[15] A. Sibiryov, "Gorb go routing and balancing," 2015.
[16] T. Marttila, "Design and implementation of the clusterf load balancer for docker clusters," master's thesis, aalto university, 2016-10-27.
[17] I. CoreOS, "etcd | etcd Cluster by CoreOS."
[18] HashiCorp, "Consul by HashiCorp."
[19] 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.," NSDI, pp.523–535, 2016.
[20] A.F. Voellm, "Compute engine load balancing hits 1 million requests per second!."
[21] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D.A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, *et al.*, "Ananta: Cloud scale load balancing," ACM SIGCOMM Computer Communication Review, vol.43, no.4, pp.207–218, 2013.
[22] Coreos, "coreos/flannel," Jul 2018.
[23] "Secure networking for the cloud native era," 2018.
[24] ktaka ccmp, "ktaka-ccmp/ipvs-ingress: Initial release," July 2017.

Kimitoshi Takahashi received the B.A. degree from Tokyo University in 1993. During 1993–2002, he stayed Fujitsu Labs. Limited, after which he cofounded Cluster Computing Inc and stayed there since then.