# A Study on Portable Load Balancer for Container Clusters

Kimitoshi Takahashi
The Graduate University for Advanced Studies
Chiyoda-ku, Tokyo, Japan
ktaka@nii.ac.jp

## 1 INTRODUCTION

(Background) Recently, launching web services on cloud computing infrastructure is getting more popular. Launching web services on the cloud is easier than launching them on on-premise data centers. A web service on the cloud becomes scalable, which means it can accommodate a large amount of traffic from the Internet by increasing the number of web servers, on demand. To distribute high volume traffic from the Internet to thousands of web servers load balancers are often used. Major cloud providers have developed software load balancers[10, 20] as part of their infrastructures, which they claim to have a high-performance level and scalability. In the case of on-premise data centers, one can use proprietary hardware load balancers. The actual implementation and the performance level of those existing load balancers are very different.

As for another issue to address regarding web services, there are also needs to use multiple of cloud providers or on-premise data centers seamlessly, which spread across the world, to prepare for the disaster, to lower the cost or to comply with the legal requirement. Linux container technology[19] facilitates these usages by providing container cluster management system as a middleware, where one can deploy a web service that consists of a cluster of containers without modification even on different infrastructures. However, in reality, it is difficult to do so, because existing load balancers are very different and are not included in the container cluster management system. Therefore, users should always manually adjust their web services to the infrastructure.

In short, there exist several software load balancers that are specific to cloud vendors and on-premise data centers. The differences among them are the major obstacles to provide uniform container cluster platform, which is necessary to realize web service migration across the different cloud providers and on-premise data centers.

To address this problem, we propose a portable and scalable software load balancer that can be used in any environment including cloud providers and in on-premise data centers. We can include this load balancer as a part of a container cluster management system, which acts as a common middleware on which web services run. Users now do not need manual adjustment of their services to the infrastructures. We will implement the proposed software load balancer using following technologies; 1) To make the load balancer usable in any environment, we containerize it using Linux container technology[19]. 2) To make the load balancer scalable, we make it capable of being run in parallel using Equal Cost Multi-Path(ECMP) technique[2]. 3) To make the load balancers performance level meet the need for 10Gbps network speed, we implement the load balancer using eXpress Data Plane(XDP) technology[5]. By providing container cluster management system including the proposed load balancer, we also aim to demonstrate migration capability, where one can migrate a web application across the different cloud providers and on-premise data centers, all over the world, with one push button.

The outcome of our study will benefit users who want to deploy their web services on any cloud provider where no scalable load balancer is provided, to achieve high scalability. Moreover, the result of our study will also benefit users who want to use a group of different cloud providers and on-premise data centers across the globe, as if it were a single computer on which their web services run.

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 3 will explain existing architecture problems and propose our solutions. In Section 4, experimental conditions and results are discussed, which is followed by a summary of our work in Section 5.

## 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.

**Container cluster migration:** Kubernetes developers are trying to add federation[3] capability for handling situations where multiple Kubernetes clusters[1] 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.

**Software load balancer containerization:** As far as load balancer containerization is concerned, the following related work has been identified: Nginx-ingress[14, 21] utilizes the ingress[4] capability of Kubernetes, to implement

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

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[22] project is trying to use Linux kernel's IPVS[25] load balancer capabilities by containerizing the keepalived[6]. 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[9, 11] 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.

**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[24] and clusterf[18] 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[7] and HashiCorp's Consul[12]. 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.

**Cloud load balancers:** Google maglev[10]
Microsoft Ananta[20]

# 3 PROPOSED ARCHITECTURE

## 3.1 Overview

Here we discuss overall architecture of the proposed load balancer. Figure 1 show conventional architecture of the container cluster infrastructure, e.g. Kubernetes. Below the dotted line, there are node linux machines that hosts containers, which is controled by Kubernetes. Avobe the dotted line, there are SWLBs and the Core router. All the equipements and software entities are controled by base cloud infrastructure.

Upon launch of the container clusters, SWLBs are set up by the cloud infrastructure. At the same time iptables DNAT rule's are set up on each of the node linux machines. These SWLBs distribute incoming traffic to every node that hosts containers. The traffic is then distributed again to destination containers using iptables destination network address translation (DNAT)[16, 17] rules in a round-robin manner.

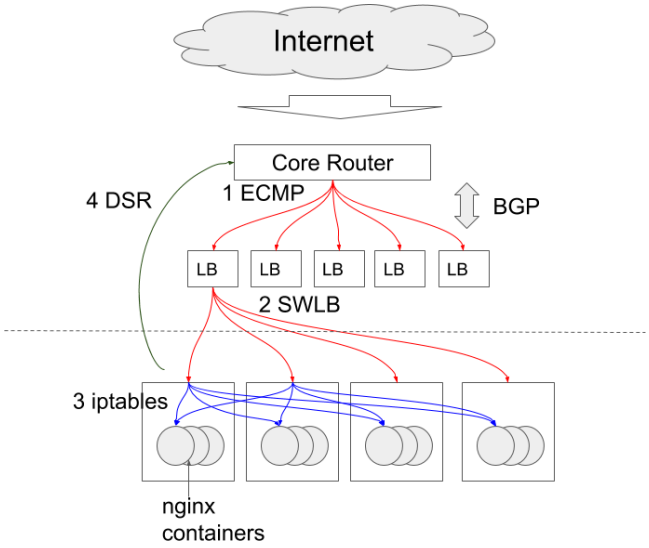


**Figure 1: Conventional architecture. SoftWare load Balancers(SWLB) are set up via cloud API.**

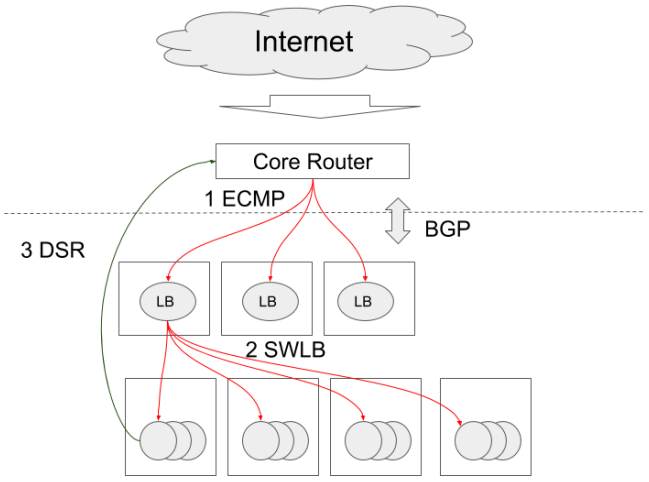


**Figure 2: Proposed architecture. Routing table of the Core Router is updated via BGP.**

Figure 1 shows proposed architecture. Upon launch of the container clusters, SWLBs are also launched as a part of the container cluster. Then, the SWLBs comunicate with the Core router using the Border Gateway Protocol(BGP) to update routing table. The core router evenly distributes the incoming traffic to SWLBs, then the SWLB directly forward the traffic to destination containers.

The big differences between the conventinal architecture and the proposed ones are; a) iptables DNAT load balancing layer does not exist in the latter, b) while the comunication between the Kubernetes and the cloud provider is the

vendor specific API requests to setup the external SLBs in the conventinal architecture, the comunication between the Kubernetes and the cloud provider is standerd BGP protocol in the proposed architecture.

In other words, the proposed architecure better than the conventional one, because it eliminates one extra network hop and cummunicate with extra world using standard protocol. And also because the SWLBs are containerized and are the part of the cluster, a service can deployed in any environment including on-premise datacenters, if only the Core router permits the update of its routing table via BGP. This will greately improve the portability compared to the conventional architecture where the support for vendor specific API is always required.

## 3.2 Containerization

Here we discuss containerization of a software load balancer. 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*[2][25]. 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 manage IPVS balancing rules in the kernel accordingly. It is often used together with IPVS to facilitate ease of use. Although keepalived also supports Virtual Router Redundancy Protocol(VRRP)[13], the authors plans to use ECMP to achive the redundancy. The controller is a daemon that periodically monitors the *pod* information on the master, and performs various actions when such information changes. Kubernetes provides ingress controller framework as the Go Language(Golang) package to implement such controllers. We have implemented a controller program that will feed *pod* state changes to keepalived using this framework.

The proposed load balancer needs to dynamically reconfigure the IPVS balancing rules whenever *pods* are created/deleted. Figure 3 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

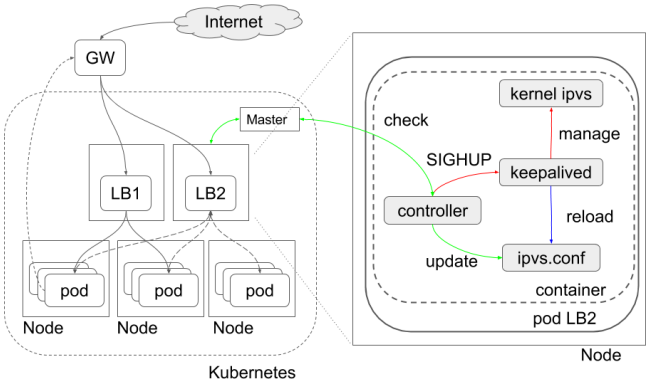


**Figure 3: Implementation**

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[15] is implemented using the Kubernetes ingress controller[4] framework. By importing existing Golang package, "k8s.io/ingress /core/pkg/ingress", we could simplify the implementation, e.g. 120 lines of code.

Even though we could successfully containerised the load balancers and could deploy it as a part of web service, the performance we obtained was not equivalent that of scalable load balancers provided by the GCP. The next subsection will discuss ongoing as to how to improve the performance levels of the open source software load balancer using the novel XDP technology.

## 3.3 XDP loadbalancer

The ipvs is very dependent on the Netfilter and Linux kernel's standard network stack. The IP packet processing of the Linux kernel has been claimed to be inefficient, and thus unable to meet the speed requirement of 10Gbps and above.

Several alternative techniques, DPDK[1], netmaps[23], PF_RING[8] and Maglev[10] to increase the speed of the packet processing have been proposed.

Most bypass the Linux kernel network stack and process the packet in user spaces. While they may improve the performance level in specific applications, they all have some issues regarding compatibility because of the bypass. Often they require dedicated physical NIC, other than the ones used for standard Linux services, e.g., ssh.

Recently, the Linux kernel introduced eXpress Data Plane (XDP)[5] a novel way to improve the traffic manipulation speed, while keeping compatibility with the other functions of the Linux standard networking stack.

[2]The term, *real servers* refers to worker servers that will respond to incoming traffic, in the original literature[25]. We will also use this term in the similar way.

By using the XDP infrastructure, a user can write code that manipulates traffic in the very early stage of accepting the packets from outside the Linux box. It will enable a user to inject a bytecode into the kernel, let the kernel compile it into native machine code and then the code will manipulate the traffic only if it matches the predefined conditions. Traffic that did not match the conditions are pass to the Linux kernel's standard networking stack.

The advantage of XDP can be summarized as follows: 1)The traffic manipulation is very fast since it can tap the very early stage of packet processing flow. 2)Since it only affects the traffic that matches the predefined conditions, irrelevant traffic is processed by the Linux standard networks stack as usual. 3)The XDP let the users manipulate the traffic in the kernel space while keeping the safety.

The XDP code can be written in C code, compiled into bytecodes using clang compiler and loaded into kernel whenever a user has needs. The XDP infrastructure provides built-in protection against dangerous codes. In contrast, while one can always implement the Linux kernel modules that manipulate the traffic, it often crashes the kernel when something wrong happens. Thus, the XDP provide in-kernel traffic manipulation capability while keeping the safety, without affecting the standards network services.

In the course of the study, the author comes to believe that it is critical to provide a software load balancer that is faster than the existing ipvs. Here he discusses the design and prototype implementation of such a load balancer.

(The author started to write a prototype code of XDP load balancer. Here the design and the prototype implementation are presented.)

(Comparison XDP, DPDK, ipvs, iptables.

DPDK: pros: bypass kernel network stack cons: only for Intel NIC, dedicated NIC required XDP: pros: hooks to the NIC driver redirect )

## 4 EXPERIMENTS

bbb

## 5 CONCLUSIONS

In order to realize the smooth migration of the container clusters, providing a uniform environment, i.e., uniform container cluster infrastructure on top of the base infrastructure is very important. Providing a standard load balancer architecture for incoming traffic is critical to achieving that purpose.

The author investigated the general architecture of the load balancer and network configurations. The lateral scalability using the software load balancer and ECMP is critical in order to meet the future demand of the large scale web services.

The author and colleague also investigated containerization of ipvs load balancer to improve the portability of the web services, using Kubernetes. They revealed that this would improve the portability of the service while keeping the performance level of conventional architecture.

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

By realizing smooth migration capability across the different cloud providers and on-premise data centers, users are freed from vendor lock-ins and ultimately obtain the opportunity to deploy global scale web services.

## 6 FUTURE WORK

1. Performance measurement of XDP loadbalancer

2. Balancing algorithm especially consistent hashing (Assessment: packet reordering, persistent ssl conncetion)

3. Containerization of XDP loadbalancer

(Future work, out of scope of the dissertation) Federation of cluster infra. in different Cloud/DC. Data Sync. Global traffic routing

## REFERENCES


[1] [n. d.]. ([n. d.]).

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 63–74.

[3] The Kubernetes Authors. 2017. Federation. (2017). Retrieved July 14, 2017 from https://kubernetes.io/docs/concepts/cluster-administration/federation/

[4] The Kubernetes Authors. 2017. Ingress Resources — Kubernetes. (2017). Retrieved July 14, 2017 from https://kubernetes.io/docs/concepts/services-networking/ingress/

[5] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, Vol. 2.

[6] Alexandre Cassen. [n. d.]. Keepalived for Linux. ([n. d.]). Retrieved July 14, 2017 from http://www.keepalived.org/

[7] Inc CoreOS. [n. d.]. etcd — etcd Cluster by CoreOS. ([n. d.]). https://coreos.com/etcd

[8] Luca Deri et al. 2004. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, Vol. 2004. Amsterdam, Netherlands, 85–93.

[9] Docker Inc. 2017. Use swarm mode routing mesh — Docker Documentation. (2017). https://docs.docker.com/engine/swarm/ingress/

[10] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer.. In *NSDI*. 523–535.

[11] Docker Core Engineering. 2016. Docker 1.12: Now with Built-in Orchestration! - Docker Blog. (2016). Retrieved July 14, 2017 from https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/

[12] HashiCorp. [n. d.]. Consul by HashiCorp. ([n. d.]). https://www.consul.io/

[13] Robert Hinden. 2004. Virtual router redundancy protocol (VRRP). (2004).

[14] NGINX Inc. 2017. NGINX Ingress Controller. (2017). Retrieved July 14, 2017 from https://github.com/nginxinc/kubernetes-ingress

[15] ktaka ccmp. 2017. ktaka-ccmp/ipvs-ingress: Initial Release. (July 2017). https://doi.org/10.5281/zenodo.826894

[16] Victor Marmol, Rohit Jnagal, and Tim Hockin. 2015. Networking in Containers and Container Clusters. *Netdev* (2015).

[17] Martin A. Brown. 2007. Guide to IP Layer Network Administration with Linux. (2007), 5.5. Destination NAT with netfilter (DNAT) pages. http://linux-ip.net/html/index.html

[18] Tero Marttila. 2016-10-27. *Design and Implementation of the clusterf Load Balancer for Docker Clusters*. Master's Thesis, Aalto University. http://urn.fi/URN:NBN:fi:aalto-201611025433

[19] Paul B Menage. 2007. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, Vol. 2. Citeseer, 45–57.

[20] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. 2013. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 207–218.

[21] Michael Pleshakov. 2016. NGINX and NGINX Plus Ingress Controllers for Kubernetes Load Balancing. (Dec. 2016). Retrieved July 14, 2017 from https://www.nginx.com/blog/nginx-plus-ingress-controller-kubernetes-load-balancing/

[22] Bowei Du Prashanth B, Mike Danese. 2016. kube-keepalived-vip. (2016). Retrieved July 14, 2017 from https://github.com/kubernetes/contrib/tree/master/keepalived-vip

[23] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.

[24] Andrey Sibiryov. 2015. GORB Go Routing and Balancing. (2015). Retrieved July 14, 2017 from https://github.com/kobolog/gorb

[25] Wensong Zhang. 2000. Linux virtual server for scalable network services. *Ottawa Linux Symposium* (2000).