コンテナクラスタ
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

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.

Several container cluster management systems, including kuberenates, docoker swarm etc., have been proposed, which enable easy deployment of container clusters. Since these container cluster management systems hide 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, there are difficulty in providing a set of loadbalancers for the incomming traffic of the web services, resulting in difficulty in providing uniform environment for easy migration of web service clusters.

For example in the case of the Kubernetes, a load balancer is not included in a cluster management system, 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)[9, 10] 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.

In order to address this issue, the author believes a rigorous study on providing uniform setups for incoming traffic, thus providing a uniform environment for container clusters, is very important. If we have the uniform environment across the different cloud providers including on-premise data centers, the migration of a web service across the world becomes easy and the user will have a variety of choices as to where to run their web services.

This eliminates cloud vendor lock-ins and daunting task that entails in adjusting service to the different type of infrastructures.

The goals of this study are the followings: 1) To clarify what type of the architecture for incoming traffic is feasible for web service migration. 2) To provide a software load balancer to achieve above architecture. 3) To provide working prototype system for the above and assess the actual feasibility.

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 **??**.

2 RELATED WORK

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

*Other affiliation: *Cluster Computing Inc. Chiyoda-ku, Tokyo, Japan, ktaka@ccmp.jp*

Container cluster migration: Kubernetes developers are trying to add federation[1] 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[8, 13] utilizes the ingress[2] 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[14] project is trying to use Linux kernel's IPVS[16] load balancer capabilities by containerizing the keepalived[3]. 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[5, 6] 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[15] and clusterf[11] 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[4] and HashiCorp's Consul[7]. 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.

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

3 COMPARISON BETWEEN THE DIFFERENT ARCHITECTURE

This section disscuss the different architecture.

In retro spect, either Google type config or

4 CONTAINERISED IPVS SOFTWARE LOAD BALANCER

The author and his collegue proposed a portable load balancer for the Kubernetes cluster systems that is aimed 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, as a proof of concept. In order 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 Kubernetes cluster system while it shows the similar performance levels as the existing iptables DNAT based load balancer. We also clarified that choosing the right operating modes of overlay networks is important for the performance of load balancers. For example, in the case of flannel, only the vxlan and udp backend operation modes could be used in the cloud environment, and the udp backend significantly degraded their performance. Furthermore, we also learned that the distribution of packet processing among multiple CPUs was very important to obtain the maximum performance levels from load balancers.

The limitations of this work that authors aware of include the followings: 1) We have not discussed the load balancer redundancy. 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. VRRP or OSPF[12] that uses multicast, can not be used in many cases. Further more, providing uniform methods independent of various cloud environments and on-premise datacenter is much more difficult. 2) Experiments are conducted only in a 1Gbps network environment. The experimental results indicate the performance of IPVS may be limited by the network bandwidth, 1Gbps, in our experiments. Thus, experiments with the faster network setting, e.g. 10Gigabit ethernet, are needed to investigate the feasibility of the proposed load balancer. 3) We have not yet compared the performance level of proposed load balance with those of cloud provider's load balancers. It shoud be fair to compare the performance of proposed load balancer with those of the combination of the cloud load balancer and the iptables DNAT. The authors leave these issues for future work and they will be discussed elsewhere.

Even though we could successfully containerised the load balancers and could deploy it as a part of web service, the performance we was not equivalent that of scalable load balancers provided by the GCP.

The next section will disscuss how to improve the performance levels of the open source software load balancer.

5 XDP LOADBALANCER

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

6 CONCLUSION

7 FUTURE WORK

REFERENCES

[1] The Kubernetes Authors. 2017. Federation. (2017). Retrieved July 14, 2017 from https://kubernetes.io/docs/concepts/cluster-administration/federation/
[2] The Kubernetes Authors. 2017. Ingress Resources — Kubernetes. (2017). Retrieved July 14, 2017 from https://kubernetes.io/docs/concepts/services-networking/ingress/
[3] Alexandre Cassen. [n. d.]. Keepalived for Linux. ([n. d.]). Retrieved July 14, 2017 from http://www.keepalived.org/
[4] Inc CoreOS. [n. d.]. etcd — etcd Cluster by CoreOS. ([n. d.]). https://coreos.com/etcd
[5] Docker Inc. 2017. Use swarm mode routing mesh — Docker Documentation. (2017). https://docs.docker.com/engine/swarm/ingress/
[6] 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/
[7] HashiCorp. [n. d.]. Consul by HashiCorp. ([n. d.]). https://www.consul.io/
[8] NGINX Inc. 2017. NGINX Ingress Controller. (2017). Retrieved July 14, 2017 from https://github.com/nginxinc/kubernetes-ingress
[9] Victor Marmol, Rohit Jnagal, and Tim Hockin. 2015. Networking in Containers and Container Clusters. *Netdev* (2015).
[10] 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
[11] 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
[12] John Moy. 1997. OSPF version 2. (1997).
[13] 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-bal
[14] 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
[15] Andrey Sibiryov. 2015. GORB Go Routing and Balancing. (2015). Retrieved July 14, 2017 from https://github.com/kobolog/gorb
[16] Wensong Zhang. 2000. Linux virtual server for scalable network services. *Ottawa Linux Symposium* (2000).