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 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.

Several container cluster management systems, including kuberenates, docker swarm etc., have been proposed, which enable easy deployment of container clusters. If these container cluster management systems could hide the differences in the base environments, users would be able to 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 smoothly migrate a web service consisting of a container cluster even to the other side of the world.

However, in actuality, the current container management systems fail to provide uniform environments across the different cloud providers. One of the most prominent differences they fail to hide is the way they route the incoming traffic into the container cluster.

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)[1, 2] 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 provide a uniform environment across the different cloud providers, it is important to have a single way of handling incoming traffic. The author believes a rigorous study on what is the best way to do so is critical.

If we have the uniform environment across the different cloud providers including on-premise data centers, users can smoothly migrate their web services without adjusting them to the base environment. This means that there are no cloud vendor lock-ins and users will gain the freedom as to where to run their web services. Users will also be able to use combined their infrasructure across the world as if it were a single computer.

The impacts the author believes this study will potentially give to the society are the followings: 1) To provide a critical contribution to enable smooth web service migrations. 2) Elimination of the vendor lock-ins. 3) To give the users opportunity to deploy global scale web services.

Therefore in this study, the author clarifies what type of the load balancer architecture is suitable for web service migration. Then he discusses a software load balancer for such architecture. Finally, he provides working prototype system for web service migration.

(This paragraph needs rewrite)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 5, 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 6, which is followed by a summary of our work in Section **??**.

2 RELATED WORK

Kubernetes

Docker swarm

IPVS

Google maglev

Microsoft Athena

Facebook shive

RFC Per flow ECMP

XDP papers

3 LOAD BALANCER AND NETWORK ARCHITECTURE

The typical traditional on-premise data center uses a set of load balancers to distribute the incoming traffic to a cluster of servers thus providing scalability and redundancy of the service. The traffic is routed from the outside of the infrastructure to one of the load balancers using active-backup redundancy protocol OSPF, VRRP or HSRP.

More recent architecture enable active-active redundancy using ECMP. The maglev[ref], athena[ref] and shive[ref] leverage this architecture by using software load balancers, thus providing scalability with less cost than the case where proprietary hardware load balancers are used.

However, since none of these load balancers are open sourced as of the writing of this paper, users can not use either of them, in the arbitrary cloud providers and in their on-premise data centers. So providing a software load balancer that is usable in all of the cloud provider and in on-premise data centers is critical, which is discussed in Section 5 and Section 6.

Here the author discusses the possible network architectures, using such software load balancers.

(Here I explain using Figs....)

It should be noted that even if the load balancer becomes portable and scalable, the actual traffic a data center infrastructure can accommodate is ultimately limited by the incoming network bandwidth and by the core router.

4 PER FLOW ECMP

(Maybe cut this section) Discuss routing traffic from upstream L3 switch. It is important connect to upstream switches using standard protocol.

ECMP VRRP

5 IPVS PORTABLE 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[3] 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 obtained 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.

6 XDP LOADBALANCER

The ipvs is very dependent on the Netfilter and Linux kernel 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 ways, DPDK, netmaps, PF_RING and Maglev 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. This table summarizes, pros and cons of the proposed techniques to improve the speed of the packet processing.

(Table)

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

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)

7 CONCLUSION

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.

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

[2] 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

[3] John Moy. 1997. OSPF version 2. (1997).