A Portable Load Balancer for Kubernetes Cluster

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

Kento Aida
National Institute of Informatics
Chiyoda-ku, Tokyo, Japan
aida@nii.ac.jp

Tomoya Tanjo
National Institute of Informatics
Chiyoda-ku, Tokyo, Japan
tanjo@nii.ac.jp

Jingtao Sun
National Institute of Informatics
Chiyoda-ku, Tokyo, Japan
sun@nii.ac.jp

ABSTRACT

The Linux containers have become very popular these days due to its lightweight nature. Many WEB services are deployed as a cluster of the containers. The Kubernetes is a popular container management system, which enables a user to deploy such WEB services easily. However, since the Kubernetes relies on cloud provider's load balancer, it was difficult to be used in environments where there was no supported load balancers. This was the case for on-premise data centers, or for cloud providers except the major ones. In this paper, we proposed a containerized software load balancer which could be run by the Kubernetes even in these environments. We implemented the load balancer so that linux kernel's ipvs could be used inside a container. Then we compared the performance of the proposed load balancer with existing iptables DNAT and the Nginx. Through the experiment, the choice of the overlay network has been discussed. Also the importance of distributing packet processings to multiple cores have been exemplified.

CCS CONCEPTS

• ;

KEYWORDS

Container, Cloud Computing, Kubernetes, Load Balancer

ACM Reference format:
Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, and Jingtao Sun. 2018. A Portable Load Balancer for Kubernetes Cluster. In *Proceedings of , Tokyo, Japan, (hpcasia2018),* 9 pages.
https://doi.org/

1 INTRODUCTION

Recently the Linux container clusters draw much attention because they are lightweight, portable and repeatable. Container clusters are generally more lightweight than Virtual Machine clusters, because the containers share the kernel with host OS, still having separate execution environment. They are generally portable because the process execution environments are archived into the tar files. Whenever one attempts to run a container, the exact same file systems are restored from the archives, even on totally different data centers. This means a container cluster can provide the repeatable and portable execution environment. For WEB services, the Linux containers are attractive because of the same reasons. Furthermore, it is expected that WEB services consisting of container clusters are capable of being easily migrated for Disaster Recovery purposes or for other business requirements.

The Kubernetes[4] is one of the popular container cluster management systems, which enable easy deployment of container clusters. By using the Kubernetes as middle-ware, we can easily deploy a WEB service on different cloud providers or on on-premise data centers, without adjusting the container cluster configuration to the base environment. This allows a user to easily migrate a WEB service consisting of a container cluster even to the different side of earth. The typical scenario is that the user starts the container cluster in a new location and changes the network configuration so that the traffic is routed to the new location. Then, he can stop the old container cluster eventually.

However, this scenario only works when the user migrates a container cluster among major cloud providers including GCP, AWS and Azure. The Kubernetes does not include a load balancer, and it is heavily dependent on external load balancers offered by the major cloud providers, which is set up on the fly through its API. The external load balancer will distribute incoming traffic to servers that host containers. The traffic is then distributed again to destination containers using iptables DNAT[17, 18] rules with 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 this case, the user needs to manually configure the static route setting for inbound traffic with an ad-hoc manner. Since the Kubernetes fails to provide an uniform environment from a view point of the container cluster among the different environments, migrating container clusters among them will always be a burden.

In order to solve this problem by eliminating the dependency on external load balancer in the Kubernetes, we propose a containerized software load balancer. The proposed load balancer runs as a part of WEB services consisting of container cluster. It enables the user to easily deploy a WEB service on different environment without modification, because the load balancer is included in the

*Other affiliation: Cluster Computing Inc. Japan, ktaka@ccmp.jp

hpcasia2018, , Tokyo, Japan

ACM ISBN .
https://doi.org/

WEB service itself. We implemented the proposed load balancer by creating containers for Linux kernel's ipvs[25] layer 4 load balancer using the existing ingress[3] framework of the Kubernetes. We also investigate the feasibility of the proposed load balancer by comparing the performance with the native iptables DNAT load balancer and the Nginx layer 7 load balancer. The results indicated that the proposed load balancer could improve the portability of container clusters without performance degradation compared with the existing load balancer. The performance of the proposed load balancer may be affected by the network configurations of overlay network and distributed packet processing. We also evaluate how the network configurations affects the performance and discusses the best setting that derive the best performance.

The contribution of this paper are followings: 1) We propose a portable software load balancer, which is runnable on any cloud provider or on on-premise data center, in a container cluster. 2) We discuss feasibility of the proposed load balancer by comparing the performance. 3) We also discuss about usable overlay network configurations and clarify importance of a technique to draw the best performance from such load balancers.

The rest of the paper is organized as follows. The section 2 highlights related work specifically those deal with container cluster migration, software load balancer containerization or a load balancer related tools in the context of the container technology. The section 3 will explain the problem of the existing architecture and propose our solution. In the section 4, experimental conditions and the important parameters we considered in the experiment will be described in detail. Then we will show the result of experiment and discuss the obtained performance characteristics in the section 5, which is followed by the conclusion in the section 6.

2 RELATED WORK

This section highlights related work especially those deal with container cluster migration, software load balancer containerization or load balancer tools in the context of the container technology.

Container cluster migration: Developers of the Kubernetes are trying to add federation[2] capability where multiple Kubernetes clusters are deployed on multiple cloud providers or on-premise data centers and are managed via the federation apiserver server. However, how each of Kubernetes clusters is run on different type of cloud providers or in on-premise data centers, especially when the load balancers of such environments are not supported by the Kubernetes, seems out of the scope of that project. The main scope of this paper is to make the Kubernetes usable on environments without supported load balancers by providing a containerized software load balancer.

Software load balancer containerization: As far as the containerization of a load balancers are concerned, there are following related works: Nginx-ingress[15, 20] utilizes the ingress[3] capability of the Kubernetes, and implements containerized Nginx proxy as a load balancer. The Nginx itself is famous as a high performance web server program, which also has the functionality of layer 7 load balancer. The Nginx is capable of handling TLS encryption and also capable of URI based switching. However the flip side of these is that it is much slower than layer 4 switching. We compared

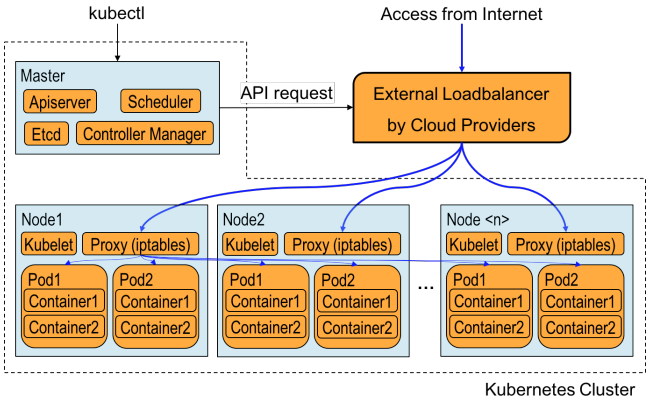


Figure 1: Conventional Architecture of A Kubernetes cluster.

the performance between the Nginx as a load balancer and the proposed load balancer in this paper. The kube-keepalived-vip[21] project is trying to use the Linux kernel's load balancer capability, called ipvs[25], by containerizing the keepalived[6] and ipvs. The keepalived container is using the daemonsets[1] framework of the Kubernetes. This work is very similar to our proposal in the sense that it uses keepalived and ipvs in container. However it differs from our proposal, in the sense that the load balancers are run on on every node as a part of the Kubernetes infrastructure. Our proposal is to provide a portable load balancer that is configurable and deploy-able at user's will, by making it run as a part of the WEB service itself not as a port of the Kubernetes cluster infrastructure. The swarm mode of the docker[10, 11] also uses ipvs for internal load balancing, however it is also considered as built-in load balancer functionality of the docker, and is different from our proposal.

Load balancer tools in the context of container: There are several other projects where an effort to utilize ipvs in the context of container environment. The gorb[22] and the clusterf[19] are daemons which setups ipvs rules in the kernel inside docker container. They utilize information about running container stored in key-value storages like the etcd[8] and the consul[13]. These could be used to implement a containerized load balancer in our proposal. However, we did not use them, since the Kubernetes ingress framework already provided the methods to retrieve the information about running container through the standard API.

3 LOAD BALANCERS IN KUBERNETES CLUSTER

3.1 Problems of Conventional Architecture

The Kubernetes container management system has an issue when it is used in outside of recommended cloud providers e.g. GCP or AWS. Figure 1 shows an exemplified Kubernetes cluster. The Kubernetes cluster typically consists of a master and nodes. 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*,

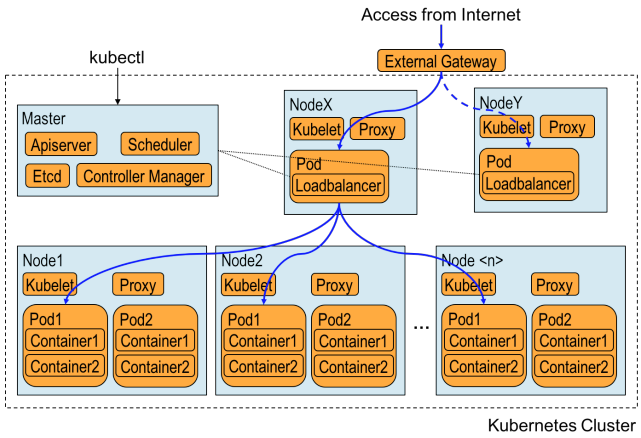


Figure 2: A Kubernetes cluster with proposed load balancer.

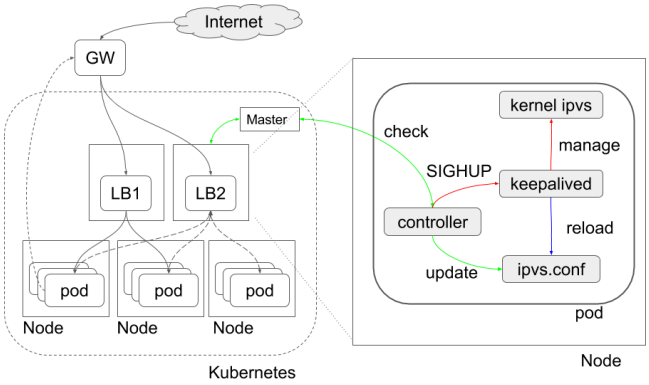


Figure 3: Implementation

depending the PodSpec information obtained from the apiserver on the master. A *pod* is a group of containers sharing net name space and cgroups, and it is the basic execution unit in the Kubernetes cluster.

When a service is created, the master will schedule *pods* and kubelets on the nodes will create them accordingly. At the same time, the masters will send out request to cloud providers API endpoints to set up external load balancers. The proxy daemon on the nodes will also setup iptables DNAT[18] rules. The traffic from the internet will be distributed by the external load balancer to nodes evenly, then it will be distributed again by the DNAT rules on the node serves to designated *pods*. The returning packets will follow the exact same route as the incoming ones.

The problems of this architecture are the followings: 1) Having external load balancers whose APIs are supported by the Kubernetes daemons is prerequisite. There are many load balancers which is not supported by the Kubernetes, especially bare metal load balancers for on-premise data centers. In such cases, one could manually setup the routing table on the gateway so that the traffic will be routed to destination nodes. Then the traffic will be distributed by the DNAT rules on the node to designated *pods*. However, this approach requires complicated network configuration and significantly degrades the portability of container clusters. 2) Distributing the traffic twice, firstly on the external load balancers and secondary on each node, will complicate the administration of packet routing. Imagine the situation, where the DNAT table on one of the nodes malfunctions. In such case, one will only observe occasional timeouts. It is very difficult to find out which node malfunctions.

So in short, 1) the Kubernetes can be used only in limited environments where the external load balancers are supported, 2) the route incoming traffic follows are very complex.

In order to address these problems, we propose a containerized software load balancer, which is deploy-able on any environment even if there are no external load balancers.

3.2 Proposed Architecture

Figure 2 shows the architecture of the proposed load balancer for the Kubernetes cluster. The proposed load balancer has following characteristics: 1) Each load balancer itself is run as a *pod* by the Kubernetes. 2) Configuration of the load balancers are dynamically updated depending on the information of running *pods*. The proposed load balancer solved the the problems of conventional architecture as follows: The load balancer can run on any environment, even on the environment without external load balancers supported by the Kubernetes including on-premise data centers, since the load balancer itself is containerized. The incoming traffic is directly distributed to destination *pods* by the proposed load balancer. It makes the administration, e.g. finding malfunctions, easier.

We designed the proposed load balancer using three components, ipvs, keepalived and ingress controller. The components are implemented as docker images. The ipvs is a layer 4 load balancer, which is included in the Linux kernel 2.6.0 released in 2003 or later, to distribute incoming TCP traffic to *real servers*[1] [25]. For example, ipvs distributes incoming HTTP traffic with one destination IP address to multiple HTTP servers (e.g. Apache HTTP or nginx) running on multiple servers to improve the performance of WEB service. The keepalived is a management program to perform health checking of *real servers* and manage the ipvs balancing rules in the kernel accordingly. It is often used with the ipvs to provide the ease of use. The keepalived also provides VRRP[14] redundancy, but that is beyond the scope of this paper. The ingress controller is a daemon to periodically monitor the *pod* information on the master, and does some actions upon change of those information. The Kubernetes provides the framework as the Golang package to implement such controller. We implemented a controller program which will feed the *pod* state change to the keepalived using this framework.

3.3 Implementation

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 rules. Two daemon programs, controller and keepalived, run in the container illustrated in the right part of the figure. The keepalived manages Linux kernel's ipvs rules depending on the configuration file, ipvs.conf. The keepalived is also capable of health-checking of

[1]This term refers to worker servers that will respond to incoming traffic, in the original literature[25]. We will also use it in the similar way.

```
virtual_server fwmark 1 {
  delay_loop 5
  lb_algo lc
  lb_kind NAT
  protocol TCP
  real_server 172.16.21.2 80 {
    uthreshold 20000
    TCP_CHECK {
      connect_timeout 5
      connect_port 80
    }
  }
  real_server 172.16.80.2 80 {
    uthreshold 20000
    TCP_CHECK {
      connect_timeout 5
      connect_port 80
    }
  }
}
```

Figure 4: An example of ipvs.conf

```
# kubectl exec -it ipvs-controller-4117154712-kv633 -- ipvsadm -L
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port Forward Weight ActiveConn InActConn
FWM  1 lc
  -> 172.16.21.2:80      Masq    1      0          0
  -> 172.16.80.2:80      Masq    1      0          0
```

Figure 5: An example of ipvs balancing rules

real server, which is represented as a combination of IP address and port number of the target *pods*. If the health-check to a *real server* fails, the keepalived will remove that *real server* from the ipvs rules.

The controller monitors information of running *pods* of a service in the Kubernetes cluster by consulting the apiserver running on the master. Whenever *pods* are created or deleted, it will automatically regenerate appropriate ipvs.conf and issue SIGHUP to the keepalived. Then the keepalived will reload the ipvs.conf and modify the kernel's ipvs rules accordingly. The actual controller[16] is implemented using ingress controller[3] framework of the Kubernetes. By importing existing Golang package, "k8s.io/ingress /core/pkg/ingress", we can simplify the implementation, e.g. 120 lines of code.

Configurations for capabilities are 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 the keepalived to manipulate kerne's ipvs rules. For the former case, we also need to mount the "/lib/module" of the node's file system on the container's file system.

Figure 4 and Figure 5 show an example of the ipvs.conf file generated by the controller and the corresponding ipvs's load balancing rules, respectively. We can see that the packet with `fwmark=1`[5] is distributed to the `172.16.21.2:80` and `172.16.80.2:80` using the masquerade mode(Masq) and the least connection(lc)[25] balancing algorithm.

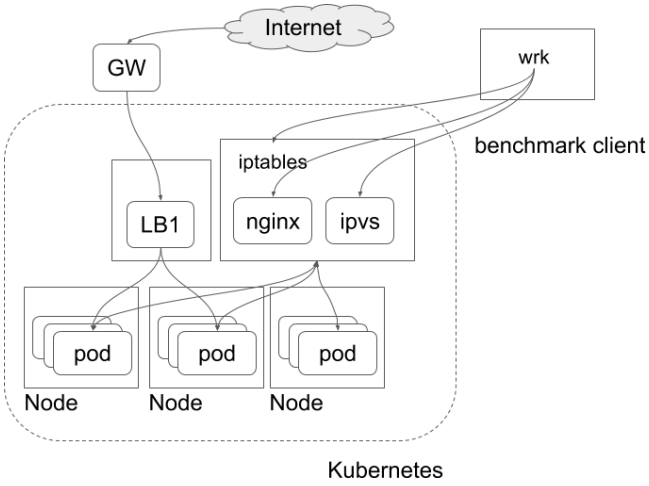


Figure 6: Benchmark setup

4 PERFORMANCE MEASUREMENT

We discuss the feasibility of the proposed load balancer by comparing the performance with the existing iptables DNAT and the Nginx based load balancer. We conducted the performance measurement using the benchmark program called, wrk[12].

We also investigated the performance under two network configurations: First one is an overlay network setting[17, 23] that is often used to build the Kubernetes cluster. The flannel[9] is one of popular overlay network technologies. We compared the performances of three backends settings[7], *i.e.* operating modes of the flannel, to find the best setting. The other one is the setting for distributed packet processing. It is known that distributing interrupt handling from the NIC and the following IP protocol processing among multiple cores impact the network performance. In order to derive the best performance from a load balancer, we also investigated how this setting would affect the performance of load balancers.

4.1 Benchmark method

Figure 6 illustrates a schematic diagram of the experimental setting. Multiple *pods* are deployed on multiple nodes in the Kubernetes cluster. In each *pod*, a Nginx web server that returns IP address of the *pod* are running. We set up the ipvs, iptables DNAT and Nginx load balancers on one of the nodes.

We measured the throughput, request/sec, of the WEB service running on the Kubernetes cluster as follows: The HTTP GET requests are sent out by the wrk on the client machine toward the nodes. The destination IP address and port number is chosen depending on the type of the load balancer on which the measurement is performed. The load balancer on the node then distribute the requests to the *pods*. Each *pod* will return HTTP responses to the load balancer and the load balancer will then return the response to the client. The number of responses received by the wrk on the client is counted, then the performance in terms of the request/sec is obtained.

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

Result sample:
 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
```

Figure 7: An example output of benchmark by wrk

Figure 7 shows an example of the command-line for the wrk and the corresponding output. The command-line in Figure 7 will generate 40 threads of wrk program and lets those threads send out in total of 800 concurrent HTTP requests during 30 seconds. The example output shows information including per thread statistics, error counts, Request/sec and Transfer/sec[MByte/sec].

Figure 8 shows hardware and software configuration used in our experiments. We configured the Nginx HTTP server to return a small HTTP content, the IP address of the *pod*, to make relatively severe condition for load balancers. The size of the characters of an IP address is only up to 15 bytes. If we had chosen the size of the HTTP response so that most of the IP packet resulted in MTU, the performance would have been limited by the bandwidth of the Ethernet. However, the performance is limited by the load balancer's ability, in the experimental setting with small HTTP responses.

The hardware we used had eight physical CPU cores and a network card(NIC) with 4 rx-queues.

4.2 Overlay network

We used the flannel to build the Kubernetes cluster used in our experiment. The flannel has three types of backend(*i.e.* operating mode), called host-gw, vxlan and udp[7].

In the host-gw mode, the flanneld on a node simply configures the routing table based on the IP address assignment information of the overlay network stored in the etcd. When a *pod* on a node sends out an IP packet to *pods* on the different node, the former node consults the routing table and learn that the IP packet should be sent out to the latter. Then the former node forms Ethernet frames having the destination MAC address of the latter node without changing the IP header and send out the IP packet.

In the case of the vxlan mode, the flanneld creates the Linux kernel's vxlan device, flannel.1. The flanneld also configures the routing table appropriately based on the information stored in the etcd. When *pods* on different nodes need to communicate, the packet is routed to flannel.1. The vxlan functionality of the Linux kernel searches the MAC address of flannel.1 device on the destination node and then forms an Ethernet frame toward the MAC address. The vxlan then encapsulates the Ethernet frame in a UDP/IP packet with the vxlan header and send out the IP packet.

```
Physical Machine Specification:
  CPU: Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz
  # of Physical Cores: 8
  Hyper Threading: enabled
  Memory: 32GB
  NIC: Broadcom BCM5720 Gigabit Ethernet PCIe

Number of Physical Machines:
  Master: 1
  Node: 7
  Client: 1

Node Software version:
  OS: Debian 8.7 (Jessie)
  Kernel: 3.16.0-4-amd64 #1 SMP Debian 3.16.39-1 (2016-12-30)
  Kubernetes v1.5.2
  flannel v0.7.0
  etcd version: 3.0.15

Load balancer Software version:
  Keepalived: v1.3.2 (12/03,2016)
  Nginx: nginx/1.11.1

Worker Pod Software version:
  nginx : nginx/1.13.0
```

Figure 8: Hardware and software configuration

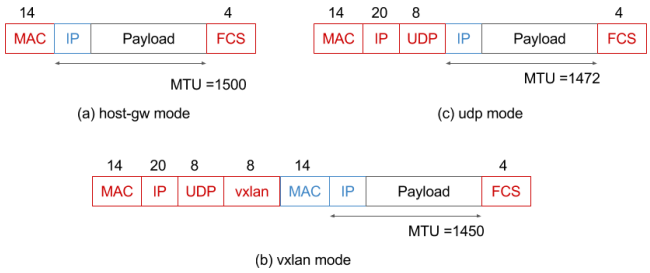


Figure 9: frame diagram

In the case of udp mode, the flanneld creates the tun device, flannel0, and configures the routing table. The flannel0 device is connected to flanneld daemon itself. The IP packet routed to flannel0 is encapsulated by the flanneld, and sent out to the appropriate node. The encapsulation is done for IP packets.

Figure 9 shows the formats of Ethernet frames in the flannel overlay network with three backends. The MTU sizes in the backends, assuming the MTU size without encapsulation is 1500byte, are also presented. Since packets are not encapsulated in the host-gw mode, the MTU size remains 1500bytes. Additional 50bytes of header is used in the vxlan mode, resulting in MTU size of 1450bytes. In the case of the udp mode, only 28bytes of header is used for encapsulation, which results in MTU size of 1472bytes.

The performance of the load balancers might be influenced by the overhead of encapsulation. Thus, the host-gw mode, where there is no overhead due to encapsulation, results in the best performances as shown in Section 5. However, the host-gw mode has a significant drawback that it does not work correctly in some cloud platforms. In other words, a gateway cannot find the proper destination of a

flannel backend	On-premise	GCP	AWS
host-gw	OK	NG	(OK)
vxlan	OK	OK	OK
udp	OK	OK	OK

Table 1: Viable flannel backends

```
81: eth0-tx-0
82: eth0-rx-1
83: eth0-rx-2
84: eth0-rx-3
85: eth0-rx-4
# obtained from /proc/interrupts
```

Figure 10: RX/TX queues of the hardware

packet and drop the packet, since the host-gw mode simply sends out a packet without encapsulation.

We investigated which backend of the flannel is usable on AWS, GCP and on-premise data centers. The results are summarized in Table 1. In the case of GCP, an IP address of `/32` is assigned to every VM host. Every communication between VMs goes through GCP's gateway. As for AWS, VMs within the same subnet communicate directly, while VMs in different subnets communicate via the AWS's gateway. The gateways do not have knowledge of the flannel overlay network and drop the packets; thus, it prohibits the use of the host-gw mode of the flannel on those cloud providers.

4.3 Distributed packet handling

Recently the performance of CPUs are improved significantly due to the development of multi-core CPUs. One of the top of the line server processors from Intel even has up to 28 cores in a single CPU. In order to enjoy the benefit of such many cores in communication performance, it is known to be necessary to distribute the handling of interrupts from the NIC and the IP protocol processing to the available physical cores. The Receive Side Scaling (rss)[24] is the technology to distribute handling of the interrupt from NIC queues to multiple CPU cores. Then the Receive Packet Steering (rps)[24] distributes the IP protocol processing to multiple CPU cores by issuing inter core software interrupt.

The performance of a load balancer may be affected by these technologies. Therefore, we compared how the performance of a load balancer changes depending on the rss and rps settings. The following shows how the rss and rps are enabled and disabled in our experiment. The Network Interface Card (NIC) used in the experiment is Broadcom BCM5720, which has four rx-queues and one tx-queue. Figure 10 shows the IRQ number assignment to those NIC queues.

When packets arrive, they are distributed to these rx-queues depending on the flow each packet belongs to. Each receive queue has a separate IRQ associated with it. The NIC triggers this to notify a CPU when new packets arrive on the given queue. Then, the notified CPU handles the interrupt and performs the protocol processing. According to the [24], the CPU cores allowed to be notified is controlled by setting hexadecimal value corresponding to the bit maps indicating the allowed CPU cores in "/proc/irq/$irq_number /smp_affinity". For example, in order to route the interrupt for eth0-rx-1 to CPU0, we should set "/proc/irq/82/smp_affinity" to binary number `0001`, which is 1 in hexadecimal value. And in order to route the interrupt for eth0-rx-2 to CPU1, we should set "/proc/irq/83/smp_affinity" to binary number `0010`, which is 2 in hexadecimal value.

We refer the setting to distribute interrupts from four rx-queues to CPU0, CPU1, CPU2 and CPU3 as `rss = on`. It is configured as the following setting:

```
rss=on
echo 1 > /proc/irq/82/smp_affinity
echo 2 > /proc/irq/83/smp_affinity
echo 4 > /proc/irq/84/smp_affinity
echo 8 > /proc/irq/85/smp_affinity
```

On the other hand, the `rss = off` means that interrupt from any of rx-queue is routed to CPU0. It is configured as the following setting:

```
rss=off
echo 1 > /proc/irq/82/smp_affinity
echo 1 > /proc/irq/83/smp_affinity
echo 1 > /proc/irq/84/smp_affinity
echo 1 > /proc/irq/85/smp_affinity
```

The rps distributes protocol processing of the packet by placing the packet on the desired CPU's backlog queue and wakes up the CPU using inter-processor interrupts. We have used the following settings to enable the rps:

```
rps=on
echo fefe > /sys/class/net/eth0/queues/rx-0/rps_cpus
echo fefe > /sys/class/net/eth0/queues/rx-1/rps_cpus
echo fefe > /sys/class/net/eth0/queues/rx-2/rps_cpus
echo fefe > /sys/class/net/eth0/queues/rx-3/rps_cpus
```

The hexadecimal value "fefe" represented as "1111 1110 1111 1110" in binary, thus this setting will allow distributing protocol processing to all of the CPUs except for CPU0 and CPU8. In this paper, we will refer this setting as `rps = on`. On the other hand, the `rps = off` means that no CPU is allowed for rps. Here, the IP protocol processing is performed on the CPU the initial hardware interrupt is received. It is configured as the following settings:

```
rps=off
echo 0 > /sys/class/net/eth0/queues/rx-0/rps_cpus
echo 0 > /sys/class/net/eth0/queues/rx-1/rps_cpus
echo 0 > /sys/class/net/eth0/queues/rx-2/rps_cpus
echo 0 > /sys/class/net/eth0/queues/rx-3/rps_cpus
```

The rps is especially effective when the NIC does not have multiple receive queues or when the number of the queues is much smaller than the number of CPU cores. That was the case of our experiment, where we had a NIC with only four rx-queues, while there was a CPU with eight physical cores.

5 RESULT AND DISCUSSION

Figure 11 shows the performance of the ipvs load balancer, that is, the throughput (Request/sec) of the nginx web server *pods* in our experiments. As for the overlay network, we measured the performance for three backend modes of the flannel, host-gw(Figure 11(a)), vxlan(Figure 11(b)) and udp(Figure 11(c)). We compared the performances with the following settings for rss and rps:

```
(rss, rps) = (off, off)
           = (on , off)
           = (off, on )
```

Except for the udp cases, we can see the trend that the throughput linearly increases as the number of *pods* increases and then it eventually saturates. The performance saturation indicates the maximum performance of the ipvs load balancer. What limits the maximum performance depends on the flannel backend modes and on the (rss, rps) settings. From the results in Figures 11(a,b), if we turn off distributed packet processing, *i.e.* when "(rss, rps) = (off, off)", the performance significantly degrades. In this case the bottleneck of the performance is mainly due to packet processing in a single core.

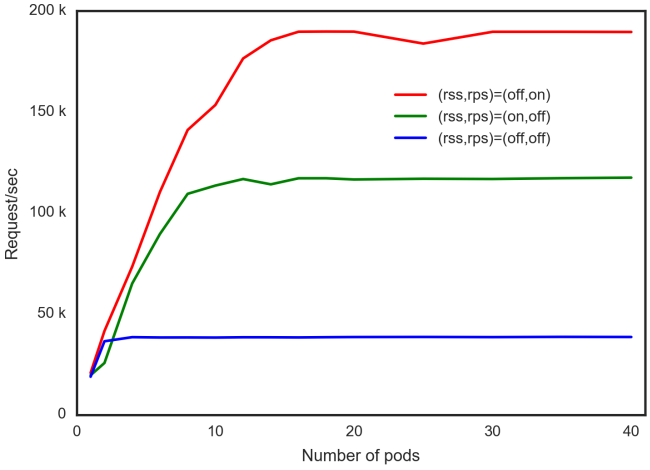
If we compare the results for the cases when "(rss, rps) = (on, off)" and "(rss, rps) = (off, on)", the latter is better than the former. It is understandable, since in the case of "rps = on", eight physical cores are used whereas in the case of "rss = on" only four cores are used on the hardware used in our experiment. The performance bottleneck of the case when "rss = on" is considered to be due to the fact that the packet processing is done on only four CPU cores. Now we are investigating the reason for the performance saturation for the case when "rps = on", and we leave the further investigation for future work.

If we compare the performances among the flannel backend modes, the host-gw mode where no encapsulation is conducted shows the highest performance, followed by the vxlan mode where the Linux kernel encapsulate the Ethernet frame. The udp mode where flanneld itself encapsulate the IP packet shows the significantly lower performances.

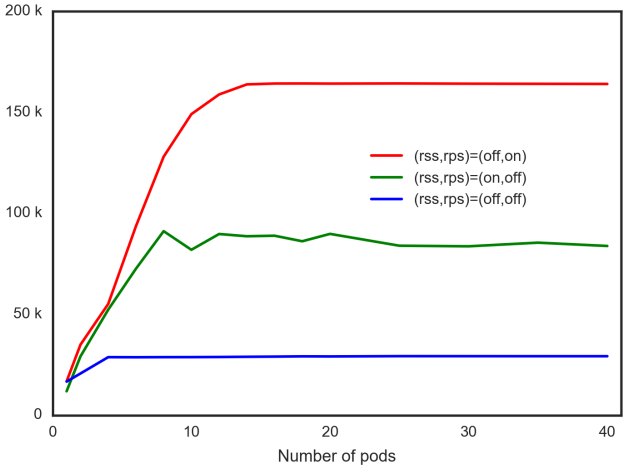
Figure 12 shows the performance of the load balancer functionality of the iptables DNAT. The performance value increases as the number of the *pod* increases linearly, then at some point comes to the saturation, as we see in the case with the ipvs results.

If we compare the results for different packet handling settings, the highest performance is obtained for the case when "(rss, rps) = (off on)", followed by the case when "(rss, rps) = (on, off)". The performance result for the case when "(rss, rps) = (off, off)" resulted in the poorest performance as the case for the ipvs. As for the flannel backend modes, the host-gw shows the highest performance followed bye the vxlan. The udp backend mode totally degrades the performance.

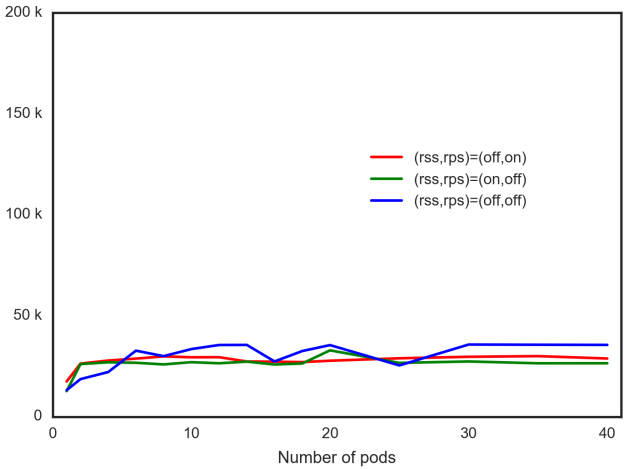
Figure 13 compares the performance measurement results among three load balancer types, namely, ipvs, iptables DNAT and nginx with the condition of "(rss, rps) = (off on)". The proposed ipvs load balancer exhibits almost equivalent performance as the existing iptables DNAT based load balancing functionality. The Nginx based load balancer shows no performance improvement even though



(a) host-gw

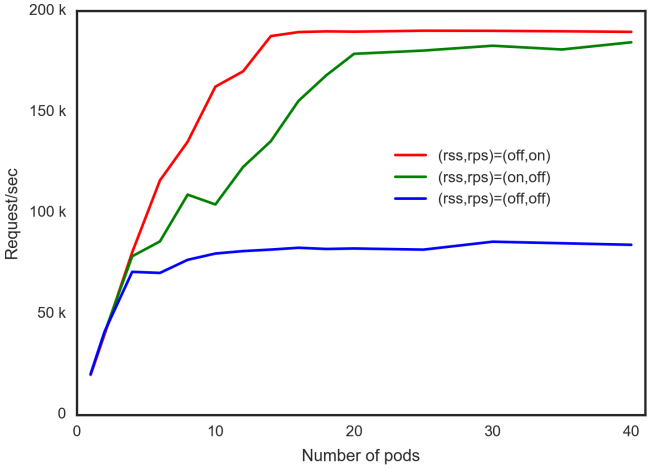


(b) vxlan

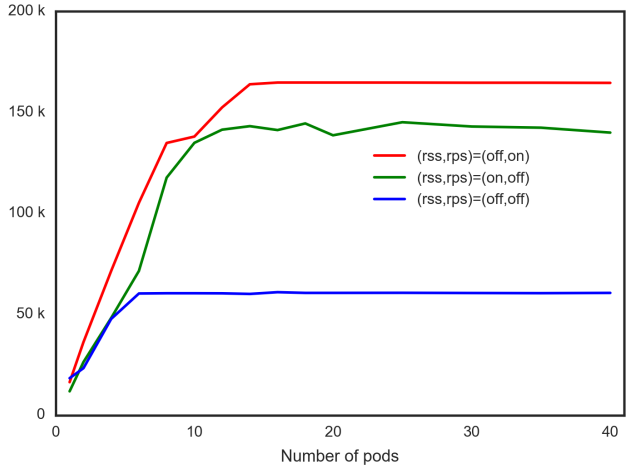


(c) udp

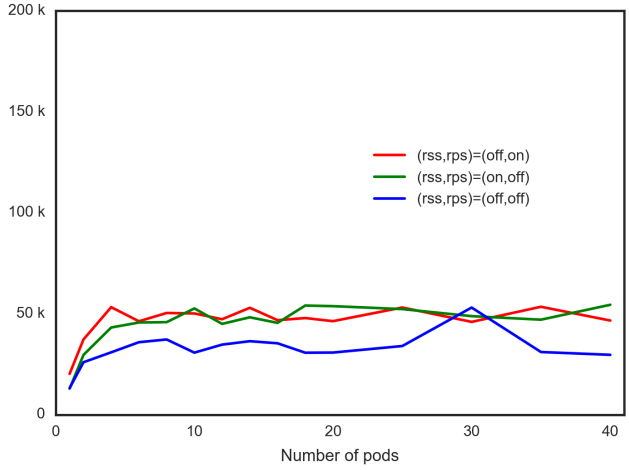
Figure 11: ipvs results



(a) host-gw

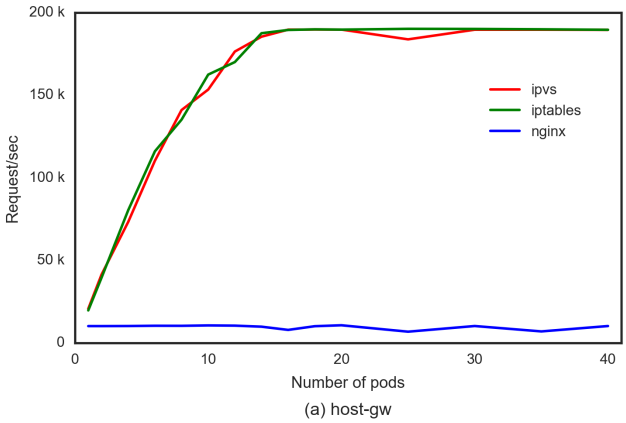


(b) vxlan

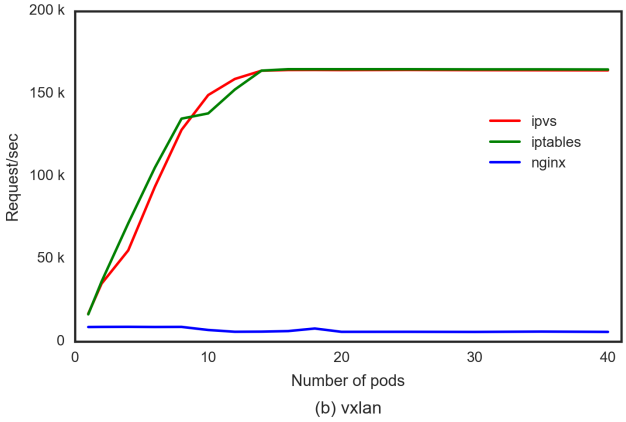


(c) udp

Figure 12: iptables results



(a) host-gw



(b) vxlan

Figure 13: ipvs and iptables comparison

the number of the Nginx web server *pods* is increased. It is understandable because the performance of the Nginx as a load balancer is expected to be similar to the performance as a WEB server.

6 CONCLUSIONS

We proposed a portable load balancer for the Kubernetes cluster system, which aims to enable migration of a container cluster for WEB services. In order to discuss the feasibility of the proposed load balancer, we built The Kubernetes cluster system, and conducted performance measurement. The experimental result indicates that the proposed ipvs based load balancer improves the portability of the Kubernetes cluster system while it shows the similar performance as the existing iptables DNAT based load balancer functionality. We also learned that only the vxlan and the udp backend operation modes could be used in the cloud environment and that 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 from load balancers.

ACKNOWLEDGMENTS

The authors would like to thank lots of people.

REFERENCES

[1] The Kubernetes Authors. 2017. Daemon Sets | Kubernetes. (2017). Retrieved July 14, 2017 from https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

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

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

[4] The Kubernetes Authors. 2017. Kubernetes | Production-Grade Container Orchestration. (2017). Retrieved July 14, 2017 from https://kubernetes.io/

[5] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, and Jasper Spaans. 2002. Linux Advanced Routing & Traffic Control HOWTO. (2002), 11. Netfilter & iproute – marking packets pages. http://www.tldp.org/HOWTO/Adv-Routing-HOWTO/index.html

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

[7] Inc CoreOS. [n. d.]. Backend. ([n. d.]). https://github.com/coreos/flannel/blob/master/Documentation/backends.md

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

[9] Inc CoreOS. [n. d.]. flannel. ([n. d.]). https://github.com/coreos/flannel

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

[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] Will Glozer. 2012. wrk - a HTTP benchmarking tool. (2012). Retrieved July 14, 2017 from https://github.com/wg/wrk

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

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

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

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

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

[18] 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

[19] 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

[20] 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/

[21] 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

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

[23] Alan Sill. 2016. Standards Underlying Cloud Networking. *IEEE Cloud Computing* 3, 3 (2016), 76–80. https://doi.org/10.1109/MCC.2016.55

[24] Tom Herbert and Willem de Bruijn. [n. d.]. Scaling in the Linux Networking Stack. ([n. d.]). https://www.kernel.org/doc/Documentation/networking/scaling.txt

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