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 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 cluster are capable of being easily migrated for Disaster Recovery purposes or for other business requirements. The migration in this context means that when one wants to move a WEB service consisting of a container cluster to the different side of the earth, he starts a container cluster in a new location and routes the traffic to there, then stops the old container cluster eventually. This is the typical scenario of WEB service migration.

The Kubernetes[4] is one of the popular container cluster management systems which enable easy deployment of the container clusters. By using the Kubernetes as a middle-ware, we expect to be able to provide a uniform environment by hiding differences the different infrastructures have. If so, we can deploy a WEB service on the different cloud providers or the on-premise data centers, without adjusting them to environment every time. The Kubernetes becomes suitable infrastructure for the WEB service migration.

However, this scenario will not work if the Kubernetes fails to provide a uniform environment. We find that the Kubernetes is heavily dependent on external load balancers which is set up by cloud providers on the fly through its API. The external load balancer will distribute the incoming traffic to every servers which hosts containers. The traffic is then distributed again to the actual containers using iptables DNAT[17, 18] rules with the round-robin manner. This mechanism is only known to work on major cloud providers including GCP, AWS and Azure. There are many load balancers which could not be set up by the Kubernetes, due to the lack of support by the Kubernetes or due to the lack of the API which a program can utilize. This is especially true for bare metal load balancers which is used in on-premise data centers. On such environments a static routing for inbound traffic could be set up manually with ad hoc manner. This results in different environments, from a view point of a container cluster. We have to always adjust a container cluster to run on different environments. Thus, migrating container clusters among those environments would always be a burden, even if we used the Kubernetes as our container infrastructure.

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

There are two possible solution to make the Kubernetes be able to provide uniform environment even if it is used in variety of cloud providers or in on-premise data centers. The first is to make all of the existing load balancers supported by the Kubernetes or replace the existing load balancers with the ones supported by the Kubernetes. The second is to make the Kubernetes independent of the external load balancers. In this paper we took the second approach as it seemed easier to realize. We won't exclude the possibility of the first approach, but that should be discussed elsewhere.

In order to eliminate the dependency on external load balancer, we propose a containerized software load balancer which is run by the Kubernetes as a part of container clusters. By doing this, we can deploy a WEB service on different environments without modification. This will ease the WEB service migration to the other side of the world. We will containerize Linux kernel's ipvs[26] layer 4 load balancer using existing ingress[3] framework of the Kubernetes. We also investigate the feasibility of such load balancer by comparing the performance with iptables DNAT load balancer functionality and Nginx layer 7 load balancer. Also discussed will be appropriate overlay network settings which enable such load balancers to work properly, and techniques to derive the most performance out of such load balancers.

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 centers, as a part of container cluster. 2) We discuss feasibility of such a load balancer by comparing the performance. 3) We also discuss about usable overlay network 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 container. 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 container.

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 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 where there is no supported load balancers by providing a containerized 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, 21] 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. The upper bound of the performance of the Nginx as a load balancer will not be much better than that as a single web server serving lightweight web pages. Therefore it is meaningless to use the Nginx as a load balancer in such cases. The kube-keepalived-vip[22] project is trying to use the Linux kernel's load balancer capability, called ipvs[26] by containerizing the keepalived[6] and ipvs. The keepalived container is using the daemonsets[1] framework of the Kubernetes. In this case the container is run on every node. This work is very similar to our proposal in the sense that it uses keepalived and ipvs in container. However it does not provide per container cluster load balancer which is portable among different cloud providers. Our proposal is to provide a load balancer that is configurable and deploy-able at user's will, as a part of container cluster 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[23] 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 has the similar functionality to a combination of keepalived and the Kubernetes ingress controller in our proposal. These may be usable in our environment in the future, but that is beyond the scope of this paper.

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 master servers and node servers. On the master servers daemons that control the Kubernetes cluster are typically deployed. These daemons include, apiserver, scheduler, controller-manager and etcd. On the node servers, the kubelet daemon will run pods, depending the PodSpec information obtained from the apiserver on the master servers. A pod is a group of containers which share the net name space and cgroups, and 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 daemons

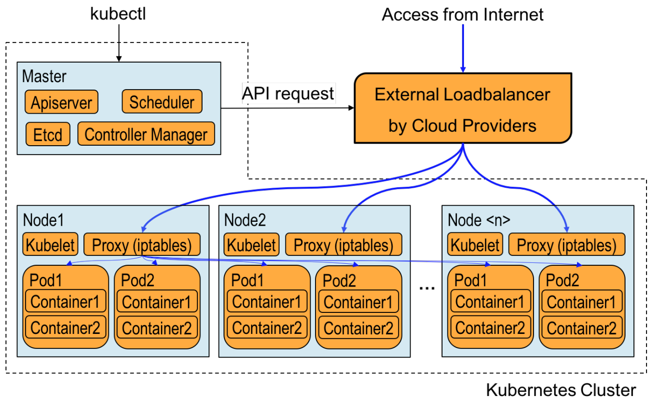


Figure 1: Conventional Architecture of A Kubernetes cluster.

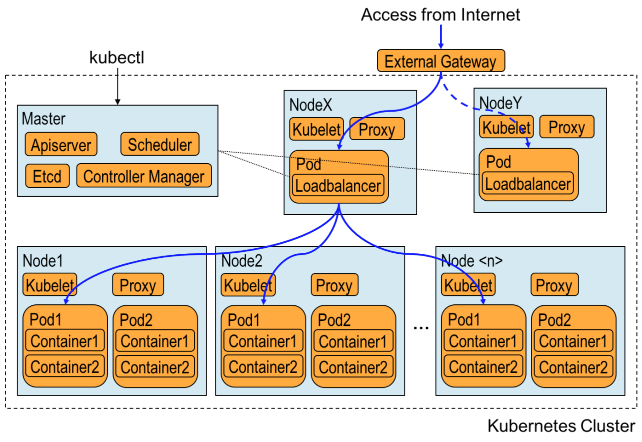


Figure 2: A Kubernetes cluster with proposed load balancer.

on the node servers will also setup iptables DNAT[18] rules, which distributes the inbound traffic to the existing pods. The traffic from the internet will be distributed by the external load balancer to node servers 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 the prerequisite. There are many cloud providers where the API of load balancers is not supported by Kubernetes. There are also many load balancers which dose not even have APIs that is usable by 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 one of the node servers. Then the traffic will be distributed by the DNAT rules on the node to designated pods. However this is far from convenient. 2) Distributing the traffic twice, firstly on the external load balancers and secondary on each node, will complicate the route which a packet follows. Imagine the situation, where the DNAT table on one of the node servers malfunctions. In such case, one will only observe occasional timeouts, which is not enough to find out the malfunctioning node.

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 is no external load balancers.

3.2 Proposed Architecture

Figure 2 shows the proposed architecture of the Kubernetes cluster, which has following characteristics: 1) Each load balancer itself is run as a pod by the Kubernetes. 2) The configuration of the load balancers are dynamically updated depending on the information about running pods.

By having these characteristics, we can resolve the problems of conventional architecture. Since the load balancer itself is containerized, it can be run on any environment where there is no supported external load balancers including on-premise data centers. And because it distribute traffic only once on the load balancer, the incoming traffic follows simple route. It becomes relatively easy to find malfunctions.

In order to realize such a load balancer we have created docker container image using ipvs, keepalived and ingress controller.

The Linux kernel has been shipped with ipvs, which is the layer 4 load balancer capability, since the release of kernel 2.6.0 in 2003. The ipvs has the capability of distributing incoming TCP traffic to many real serves where actual server programs reside. One typical example is distribute the HTTP traffic destined for a virtual server with single IP, to many of it's subordinate real servers where HTTP server program like the Apache HTTP server or the nginx actually runs. In this way we can balance the load of a single server to its subordinate real servers.

The keepalived is a management program which performs the health checking of the real servers and manage the ipvs balancing rules in the kernel accordingly. Because it provides the ease of use, it is often used with the ipvs. The keepalived also provides VRRP[14] redundancy. We can utilize it, but that is beyond the scope of this paper.

The ingress controller is a daemon which periodically monitors the pod information on the master, and does some actions upon change of those information. The Kubernetes provides the framework as a Golang package to implement such controller. We have implemented a controller program which will feed the pod state change to the keepalived using this framework.

These are the main idea of our proposal, the following section explains the implementation in detail.

3.3 Implementation

Figure 3 show the schematic diagram of a container that functions as a load balancer. Inside the container on a node, two daemons, controller and keepalived are run. The keepalived is a program which manages Linux kernel's ipvs rules depending on the configuration file, ipvs.conf. The keepalived is also capable of health-checking the liveliness of real-server, which is a combination of IP address and port number of the target pods. If the health-check to a real

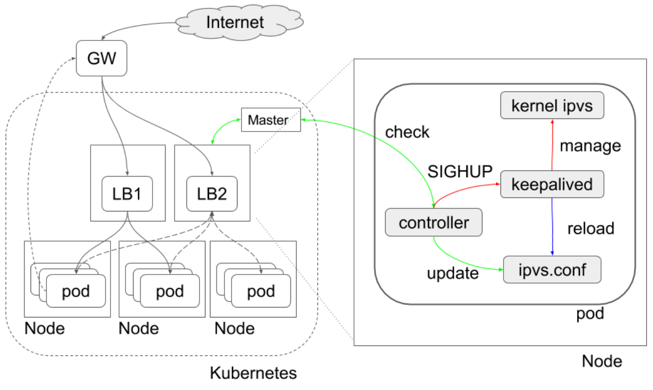


Figure 3: Implementation

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 master server. 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] has been implemented using ingress controller[3] framework of the Kubernetes. By importing existing Golang package, "k8s.io/ingress /core/pkg/ingress", we needed only 120 lines of code.

In this way, the ipvs's balancing rules inside Linux kernel are maintained so that it can distribute the incoming traffic only to the living pods.

In order to let the container function as a load balancer we needed let the kernel load appropriate ipvs related kernel modules, including ip_vs.ko. In order to do so, we needed to add the CAP_SYS_MODULE capability to the container, and also needed to mount the "/lib/module" of the node's file system on the container's file system. In order to let the keepalived manipulate kernel's ipvs rules, we needed to add CAP_NET_ADMIN capability to the container. There is an explanation in [20] about some of the capabilities that is usually dropped inside docker containers.

Figure 4 shows an example of ipvs.conf file generated by the controller. Shown in Figure 5 is the corresponding ipvs's load balancing rules. 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)[26] balancing algorithm.

4 PERFORMANCE MEASUREMENT

We discuss the feasibility of the proposed load balancer by comparing the performance with existing iptables DNAT and Nginx based load balancer. We carried out performance measurement of the proposed load balancer using benchmark program called, wrk[12]. We also measured performance of iptables's DNAT load balancing functionality and Nginx layer 7 load balancer for comparison.

When creating Kubernetes cluster, an overlay network[17, 24] is often used. Among available overlay networks, the flannel[9] is a popular one. We compared each of the three backends[7], *i.e.* operating modes of the flannel in the benchmark of the load balancers.

```
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

It is known that distributing the handling of interrupts from the NIC and the following IP protocol processing, among multiple cores will improve the performance of a Linux based network appliances including load balancers. We also investigated how this would affect the performance of load balancers.

The following subsections explain these further in detail.

4.1 Benchmark method

Using the wrk, we measured performance of the load balancers. Figure 6 shows schematic diagram of the experimental setups. Having deployed pods running Nginx web servers which returns IP address of the pod itself, we set up the ipvs, iptables DNAT and Nginx load balancers on one of the node servers.

The HTTP GET requests are sent out by the wrk on the client machine toward the node servers. 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 server 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.

Figure 7 shows an example of the command-line for the wrk and the corresponding output. The command-line in Figure 7 will

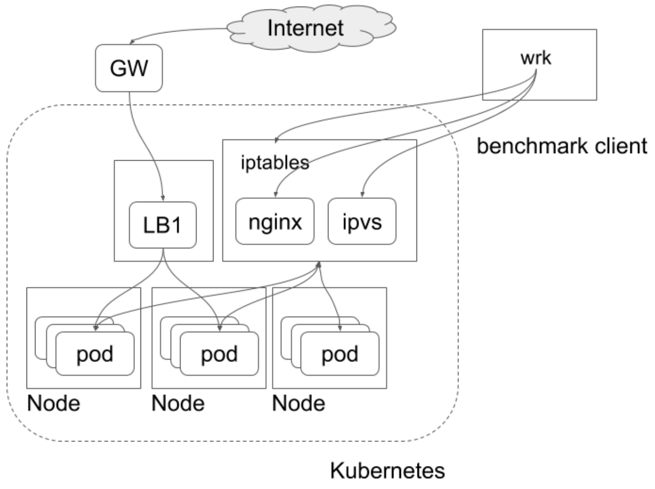


Figure 6: Benchmark setup

```
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

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]. We considered the "Request/sec" to be the performance, and compared them among several conditions.

Figure 8 shows hardware and software configuration used in our experiments. We used Nginx as the target HTTP server which returns own IP address as HTTP content. The size of the characters of an IP address is only up to 15 bytes, which is relatively severe condition for a load balancer. If we chose the size of the HTTP response so that most of the IP packet result in MTU, the benchmark results would be limited by the bandwidth of the Ethernet. Since we used smaller HTTP responses, we could perform the severer benchmark against load balancers than otherwise. Each of the hardwares we used had a network card(NIC) with 4 rx-queues. It had 8 physical CPU cores, which will become 16 logical core when the Hyper Threading is enabled.

```
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

4.2 Overlay network

When building a Kubernetes cluster, one has a choice in usable overlay networks. We used the flannel to build the Kubernetes cluster used in our experiment. The flannel has 3 types of backend *i.e.* operating mode, called host-gw, vxlan and udp[7].

In the host-gw mode, the flanneld daemon program on a node server simply set up the routing table based on IP address assignment information of the overlay network which is stored in etcd. When a pod on a node sends out a IP packet to another pods on a different node server, the former node servers will consult the routing table and learn that that IP packet should be sent out to the latter node. Then the former node will form Ethernet frames having destination MAC address of the latter node without changing the IP header and send out them.

In the case of the vxlan mode, the flanneld will create Linux kernel's vxlan device, flannel.1. The flanneld also setup 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 will find out MAC address of flannel.1 device on the opponent node server then form a Ethernet frame toward that MAC address. The vxlan will then encapsulate the Ethernet frame in a UDP/IP packet with vxlan header. The IP packet is sent out eventually.

In the case of udp mode, the flanneld will create a tun device, flannel0 and setup the routing table. The flannel0 device is connected to flanneld daemon itself. An IP packet routed to flannel0 is encapsulated bye the flanneld, and eventually sent out to the appropriate node server. The encapsulation is done for IP packets.

Figure 9 shows schematic diagrams of frame format for the three backend of the flannel overlay network. Also shown are the MTU

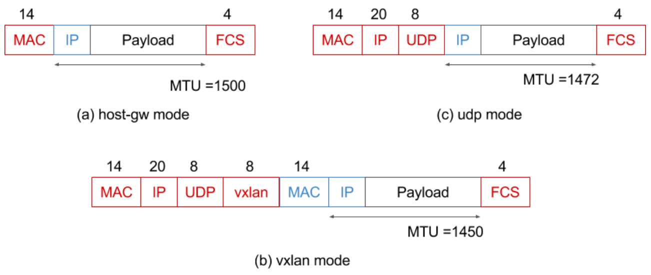


Figure 9: frame diagram

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

Table 1: Viable flannel backends

sizes of each operation mode, assuming the MTU without encapsulation is 1500byte. Since packets are not encapsulated in the host-gw mode, the MTU remains 1500bytes. Additional 50bytes of header is used in vxlan mode, resulting in MTU of 1450bytes. In the case of udp mode only 28bytes of header is used for encapsulation, which results in MTU of 1472bytes.

Since each of the three backend mode of the flannel uses different mechanism to send out packets to pods on different node servers, performance of the load balancers are expected to be influenced by the overhead of encapsulation processing. We assumed the host-gw mode, where there is no overhead due to encapsulation, will result in the best performances. That was the case as shown in Section 5. In the experiment, we compared the performance of load balancers when different backend mode of the flannel is used.

There is one issue to mention about the gw-mode of the flannel. Since it simply sends out a packet without encapsulation, if there is a gateway between node servers, the gateway will drop the packet because it does not know where to send the packet to.

This is known to happen in the case of cloud providers. We have 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, if VMs are within a same subnet, they communicate directly. However a communication is via AWS's gateway if the VMs reside in different subnets. The gateways do not have knowledge of the flannel overlay network, thus prohibit the use of the host-gw mode of the flannel on those environment.

4.3 Distributed packet handling

Recently the performance of a CPU is 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, 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

```
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

Receive Side Scaling (rss)[25] is the technology which distributes the handling of the interrupt from the NIC queues to multiple CPU cores. Then the Receive Packet Steering (rps)[25] will distributes the IP protocol processing to multiple CPU cores by issuing inter core software interrupt.

We expected that the performance of a load balancer was affected by these technology. 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 4 rx-queues and 1 tx-queue. Figure 10 shows the IRQ number assignment to those NIC queues.

When packets arrive, they will be 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. The notified CPU will handle the interrupt and then perform the protocol processing. According to the [25], which CPU core is allowed to be notified is controlled by setting hexadecimal value corresponding to the bit maps indicating 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 can distribute interrupts from 4 rx-queues to CPU0, CPU1, CPU2 and CPU3, by the following settings:

```
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
```

In this paper, we will refer this setting as `rss = on`. On the other hand the `rss = off` means the following settings:

```
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
```

In this case, interrupt from any of rx-queue is routed to CPU0.

The rps will distribute protocol processing of the packet by placing the packet on the desired CPU's backlog queue and wake up that 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 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
```

In this case no CPU is allowed for rps. The IP protocol processing is performed on the CPUs the initial hardware interrupt is received.

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 4 rx-queues, while there was a CPU with 8 physical cores.

5 RESULT AND DISCUSSION

Figure 11 shows benchmark results for ipvs load balancer. All of them shows the performance in terms of Request/sec as a function of the number of nginx web server pods. As for the overlay network, we measured the performance for 3 backend mode of the flannel, host-gw(Figure 11(a)), vxlan(Figure 11(b)) and udp(Figure 11(c)).

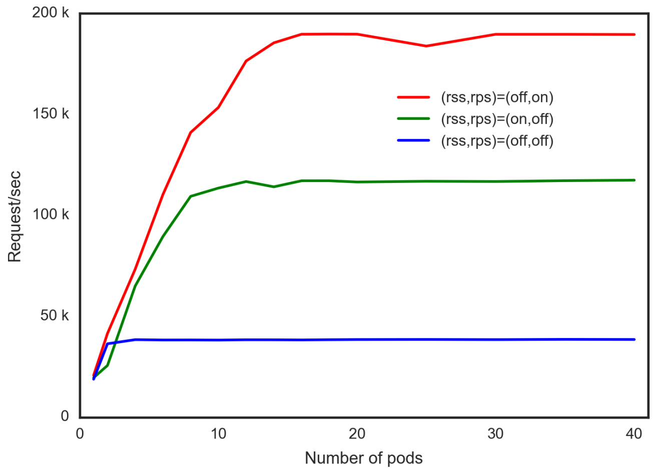
The rss and rps settings compared are the followings:

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

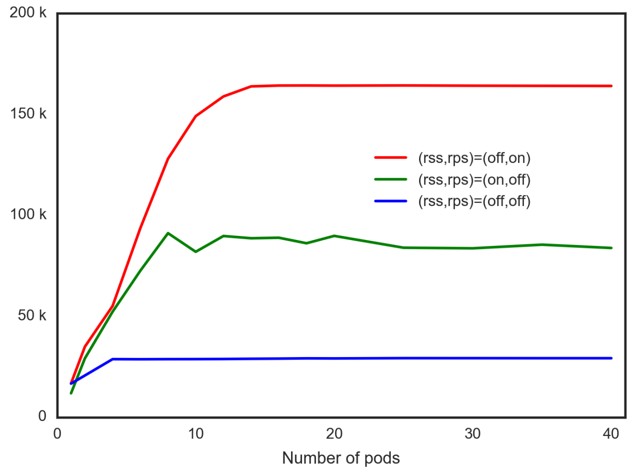
Except for the udp cases, there seems to exist general tendency, where as the pod number increases the resulting performance increases linearly to some point, then eventually saturate.

Those saturated performance indicate the maximum performance of ipvs load balancer. What limits the maximum performance depends on the flannel backend type 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 single core packet processing.

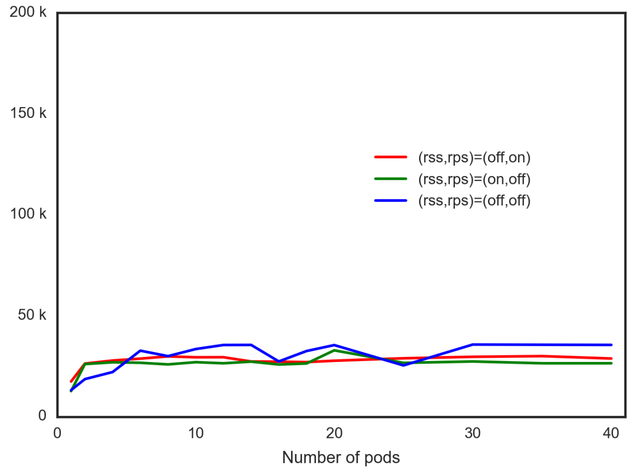
If we compare the results for the cases when "(rss, rps) = (on, off)" and "(rss, rps) = (off, on)", the latter is better than the former. It is understandable, since in the case of "rps = on", 8 physical cores are used whereas in the case of "rss = on" only 4 cores are used on the hardware used in our experiment. The performance bottleneck of the case when "rss = on" is considered to be due to the fact that



(a) host-gw

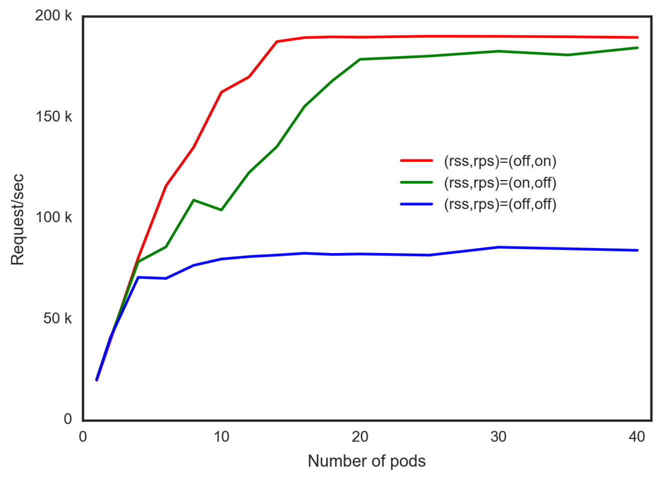


(b) vxlan

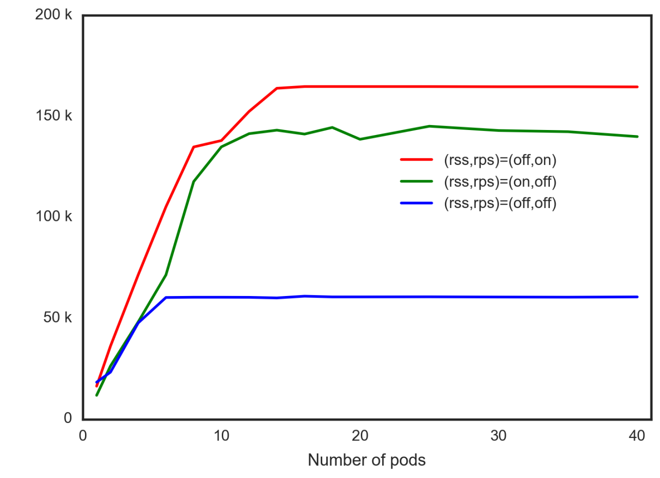


(c) udp

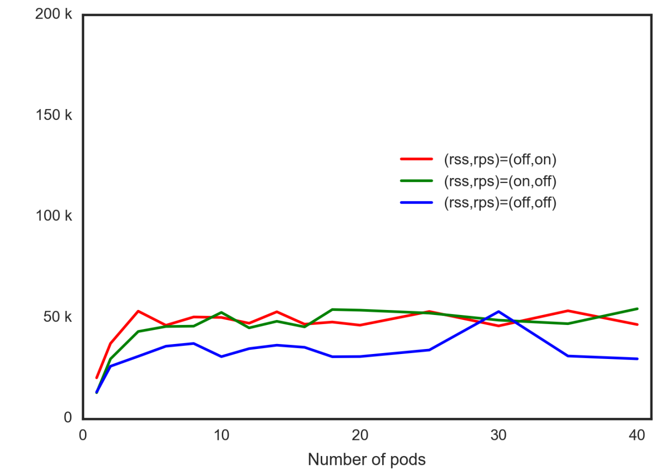
Figure 11: ipvs results



(a) host-gw

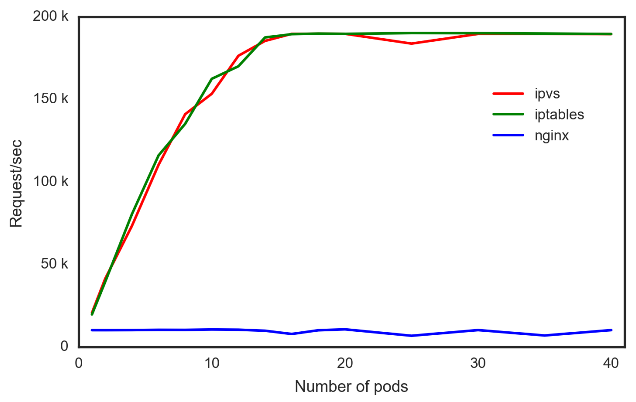


(b) vxlan

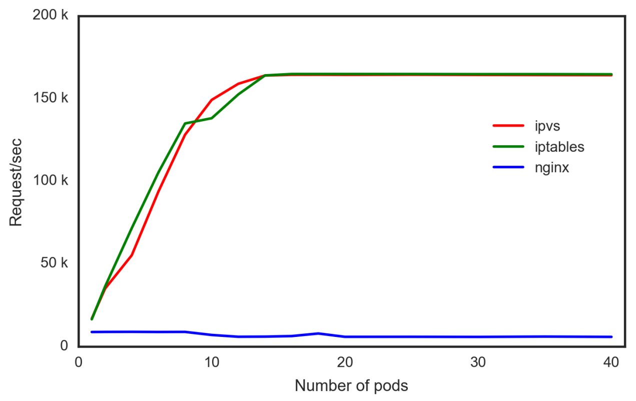


(c) udp

Figure 12: iptables results



(a) host-gw



(b) vxlan

Figure 13: ipvs and iptables comparison

the packet processing is done on only 4 CPU cores. What limit the performance for the case when “rps = on” is not clear yet.

If we compare the performance among the type of the flannel backend, the host-gw mode where no encapsulation is conducted has 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 has the significantly lower performances.

Figure 12 show the benchmark results for 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 was the case with 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 was the case for the ipvs. As for the type of the flannel backend, the host-gw has the highest performance followed bye the vxlan. The udp backend totally degrade the performance.

Figure 13 compares the performance measurement results among the load balancer type, namely, ipvs, iptables DNAT and nginx with the condition of "(rss, rps) = (off on)". The proposed ipvs load balancer has almost equivalent performance as the existing iptables DNAT based load balancing functionality. The nginx based load balancer significantly under performs compared with two other methods. For the nginx load balancer the performance never increases even though the number of the nginx web servers pod is increased. This is understandable because the nginx as the web server in our experiment only returns it's IP address, which is nearly the lightest workload. The performance of the nginx as a load balancer can not be much better than that of nginx as a web server with lightest workload. Therefor we can expect that the nginx as a load balancer limit the performance to it's own maximum even though the web server pods are added to the cluster.

6 CONCLUSIONS

We proposed a portable load balancer for Kubernetes cluster system, which aims to enable migration of a container cluster for WEB services. In order to discuss the feasibility of such a load balancer, we built Kubernetes cluster system, and carried out performance measurement. The experimental result indicated that the proposed ipvs based load balancer had almost similar performance as the existing iptables DNAT based load balancer functionality. We have also clarified that the choice of the overlay network is very important to draw the best performance from the load balancers. We could achieve the best performance without encapsulation, however it may not work on cloud providers. The vxlan mode of the flannel out performed the udp mode. Further more it was also clarified that the distribution of packet processing among multiple CPU cores is very important draw the best 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] Amith Raj MP, Ashok Kumar, Sahithya J Pai, and Ashika Gopal. 2016. Enhancing security of Docker using Linux hardening techniques. In *Applied and Theoretical Computing and Communication Technology (iCATccT), 2016 2nd International Conference on*. IEEE, 94–99.

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

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

[25] 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

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