A Study on Portable Load Balancer for Container Clusters

by

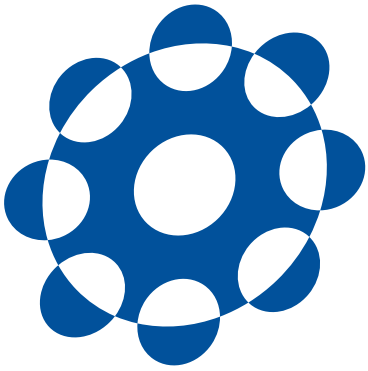
Kimitoshi Takahashi

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy



The Graduate University for Advanced Studies (SOKENDAI)

March 2019

Committee

AAA BBB (Chair)	National Institute of Informatics / Sokendai
CCC DDD	EEE University
FFF GGG	National Institute of Informatics / Sokendai

Abstract

Linux container technology and clusters of the containers are expected to make web services consisting of multiple web servers and a load balancer portable, and thus realize easy migration of web services across the different cloud providers and on-premise datacenters. This prevents service to be locked-in a single cloud provider or a single location and enables users to meet their business needs, e.g., preparing for a natural disaster. However existing container management systems lack the generic implementation to route the traffic from the internet into the web service consisting of container clusters. For example, Kubernetes, which is one of the most popular container management systems, is heavily dependent on cloud load balancers. If users use unsupported cloud providers or on-premise datacenters, it is up to users to route the traffic into their cluster while keeping the redundancy and scalability. This means that users could easily be locked-in the major cloud providers including GCP, AWS, and Azure. In this paper, we propose an architecture for a group of containerized load balancers with ECMP redundancy. We containerize Linux ipvs and exabgp, and then implement an experimental system using standard Linux boxes and open source software. We also reveal that our proposed system properly route the traffics with redundancy. Our proposed load balancers are usable even if the infrastructure does not have supported load balancers by Kubernetes and thus free users from lock-ins.

Contents

Chapter 1

Introduction

1.1 Motivation of the research

Nowadays, a great number of people in the world can not spend a day without using smartphones or personal computers(PCs) to retrieve information for work or for dailylife from the internet. For example, people use those devices to look up weather, news, emails, social media and sometimes to play games. These services are often called web services, where information is deliverd using Hyper Text Transfer Protocols(HTTP) or Hypertext Transfer Protocol Secure (HTTPS) from servers at the other end of the internet. Web services are provided by various organizations, including commercial companies, government, non-profitable organizations, schools, etc. A client program on PCs or smartphone sends out requests to servers and the servers respond with data that is requested, using HTTP or HTTPS.

Servers for web services are usually computers located in a data center. Servers also refer to the server programs that are runing on those computers. Multiple servers cooperate to fulfill the need of the clients. The author call a group of those servers a web cluster or a web service cluster. Fig. xxx shows schematic diagram of an example of a web cluster. There are several servers that work together to respond to requests from clients. There are also load balancers that distribut requests to multiple web servers. Those servers are often purchased by web service providers and located in server housing facilities called data centers.

The emergence of Cloud Computing made it easier for service providers to deploy web services than before. Cloud providers utilize virtual machines(VM) where multiple VMs share a single physical server. They charge their customers with finer granularities in pay-as-you-go bases. From the point of view of cloude users (i.e. web service providers), this lowers initial cost spent on infrastructures for their services. It also shortens the time to start a service from the inception of the project, hence gives the users agility. And at the time when the computing resources for a service is excessive, users can easily decrease the VMs.

More recently, Linux containers[1] have come to draw a significant amount of attention because they are lightweight, portable, and reproducible. Linux containers are generally more lightweight than virtual machines(VMs), because the containers share the kernel with the host operating system (OS), even though they maintain separate execution environments. Linux containers can be run on top of Linux OS, therefore they can be run in most of the cloud infrastructures and on-premise data centers. 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 reproducible 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 a cluster of containers would be capable of being migrated easily for a variety of purposes. For example disaster recovery, cost performance optimizations, meeting 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.

The motivation of this research is to develop tools and infrastructures that will enable users to deploy their services across the world seamlessly, regardless of the cloud providers or data centers they use. Also, the author aims to realize the future where users can choose whatever infrastructure they like without sacrificing

advanced features that are provided only by limited cloud providers.

1.2 Lock-in problem

It is desirable if users can migrate their services to multiple of cloud providers or on-premise data centers seamlessly, which spread across the world. Container cluster management systems facilitate these usages by functioning as middlewares, which hide the differences among cloud providers and on-premise data centers.

Kubernetes[2], which is one of the most popular container cluster management systems, enables easy deployment of container clusters. Kubernetes are initially developed by engineers inside Google, to facilitate container cluster deployment for web services. Kubernetes allows users to deploy a cluster of containers each of which depends on each other, with ease of launching a single program. It also allows users to increase or decrease the number of containers dynamically.

Since Kubernetes is expected to hide the differences in the base environments, it is expected that users can easily deploy a web service on different cloud providers or on on-premise data centers, without adjusting the container cluster configurations to the new environment. This allows a user to easily migrate a web service consisting of a container cluster even to the other side of the world with the following senario; A user starts the container cluster in the new location, route the traffic there, then stop the old container cluster at his or her convenience. This is a typical web service migration scenario.

However, this scenario only works when the user migrates a container cluster among major cloud providers including Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure. This is because Kubernetes partially hides differences in base environments. Kubernetes does not provide generic ways to route the traffic from the internet into container cluster running in the Kubernetes and expects the base infrastructure route traffic to every node that might host container. In other words, Kubernetes is heavily dependent on cloud load balancers, which is external load balancers that are set up on the fly by cloud providers through their application protocol interfaces (APIs). Once the traffic reaches the nodes, Kubernetes handles it nicely, but this is a problem since not every cloud provider or on-premise data center has load balancers capable of being utilized by Kubernetes.

Other container cluster management systems, e.g. docker swarm, etc, also lack a generic way to route the trafic into the container cluster. One of the aims of this study is to seek a generic way to route the traffic into container clusters, and thereby facilitating web service migrations.

Load balancers are often used to distribute high volume traffic from the Internet to thousands of web servers . Major cloud providers have developed software load balancers[3, 4] as a part of their infrastructures. They claim that their load balancers have a high-performance level and scalability. Once cloud load balancers distribute incoming traffic to every server that hosts containers, the traffic is then distributed again to destination containers using the iptables destination network address translation(DNAT)[5, 6] rules in a round-robin manner.

In the case of on-premise data centers, there are variety of proprietary hardware load balancers. The actual implementation and the performance level of those existing load balancers are very different and are most likely not supported by Kubernetes. In those environments, the user needs to manually configure the static route for inbound traffic in an ad-hoc manner. Since the Kubernetes fails to provide a uniform environment from a container cluster viewpoint, migrating container clusters among the different environments will always require a daunting tasks. The other aims of this study is to provide a load balancer that woks well with Kubernetes for environments lacking support by Kubernetes, and thereby facilitating web service migrations.

1.3 Contribution

In order to achieve these aims, the author proposes a portable and scalable software load balancer that can be used with Kubernetes in any environment including cloud providers and in on-premise data centers. Users now do not need to manually adjust their services to the infrastructures. As a proof of concept the author implements the proposed software load balancer using following technologies; 1) To make the load balancer usable in any environment, we containerize ipvs[7] using Linux container technology[1]. 2) To make the load balancer redundant and scalable, we make it capable of updating the routing table of upstream router with Equal Cost Multi-Path(ECMP) routes[8] using a standard protocol, Border Gateway Protocol(BGP). In order to make the load balancer's performance level to meet the need for 10Gbps network speed, a software load balancer that better performs than ipvs is required. The author also extends the research into implementing the novel load balancer using eXpress Data Plane(XDP) technology[9] to enhance the performance level.

The author implements containerized Linux kernel's Internet Protocol Virtual Server (ipvs)[7] Layer 4 load balancer using a Kubernetes ingress[10] framework, as a proof of concept. Then extend it to support Equal Cost Multi-Path(ECMP)[?] redundancy by running a Border Gateway Protocol(BGP) agent container together with ipvs container. Functionality and performances are evaluated for each of them.

Although major cloud providers do not currently provide BGP peering service for their users, the authors expect our proposed load balancer will be able to run, once this approach is proven to be beneficial and they start BGP peering services. Therefore we focus our discussions on verifying that our proposed load balancer architecture is feasible, at least in on-premise data centers. For the cloud environment without BGP peering service, single instance of ipvs load balancer can still be run with redundancy. The liveness of the load balancer is constantly checked by one of the Kubernetes agents, and if anything that stop the load balancer happens, Kubernetes will restart the load balancer container. The routing table of the cloud provider can be updated by newly started ipvs container immediately.

The authors limit the focus of this study on providing a portable load balancer for Kubernetes to prove the concept of proposed architecture. However, the same concept can be easily applied to other container management systems, which should be discussed in future work.

The contributions of this paper are as follows: Although there have been studies regarding redundant software load balancers especially from the major cloud providers[3, 4], their load balancers are only usable within their respective cloud infrastructures. This paper aims to provide a redundant software load balancer architecture for those environments that do not have load balancers supported by Kuberenetes. The understanding obtained from detailed analysis of the evaluation also helps both the research community and web service industry, because there is not enough of them. Moreover, since proposed load balancer architecture uses nothing but existing Open Source Software(OSS) and standard Linux boxes, users can build a cluster of redundant load balancers in their environment.

The outcome of our study will benefit users who want to deploy their web services on any cloud provider where no scalable load balancer is provided, to achieve high scalability. Moreover, the result of our study will potentially benefit users who want to use a group of different cloud providers and on-premise data centers across the globe seamlessly. In other words, users will become being able to deploy a complex web service on aggregated computing resources on the earth, as if they were starting a single process on a single computer.

1.4 Outline

The rest of the paper is organized as follows. Section 2.1 highlights related 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 **??** discusses problems of the existing architecture and proposes our solutions. In Section **??**, we explain experimental system in detail. Then, we show our experimental results and discuss obtained characteristics in Section **??**, which is followed by a summary of our work in Section 9.1.

Chapter 2

Background

2.1 Related Work

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

Container cluster migration: Kubernetes developers are trying to add federation[11] capability for handling situations where multiple Kubernetes clusters[1] are deployed on multiple cloud providers or on-premise data centers, and are managed via the Kubernetes federation API server (federation-apiserver). However, how each Kubernetes cluster is run on different types of cloud providers and/or on-premise data centers, especially when the load balancers of such environments are not supported by Kubernetes, seems beyond the scope of that project. The main scope of this paper is to make Kubernetes usable in environments without supported load balancers by providing a containerized software load balancer.

Software load balancer containerization: As far as load balancer containerization is concerned, the following related work has been identified: Nginx-ingress[12, 13] utilizes the ingress[10] capability of Kubernetes, to implement a containerized Nginx proxy as a load balancer. Nginx itself is famous as a high-performance web server program that also has the functionality of a Layer-7 load balancer. Nginx is capable of handling Transport Layer Security(TLS) encryption, as well as Uniform Resource Identifier(URI) based switching. However, the flip side of Nginx is that it is much slower than Layer-4 switching. We compared the performance between Nginx as a load balancer and our proposed load balancer in this paper. Meanwhile, the kube-keepalived-vip[14] project is trying to use Linux kernel's ipvs[7] load balancer capabilities by containerizing the keepalived[15]. The kernel ipvs function is set up in the host OS's net namespaces and is shared among multiple web services, as if it is part of the Kubernetes cluster infrastructure. Our approach differs in that the ipvs rules are set up in container's net namespaces and function as a part of the web service container cluster itself. The load balancers are configurable one by one, and are movable with the cluster once the migration is needed. The kube-keepalived-vip's approach lacks flexibility and portability whereas ours provide them. The swarm mode of the Docker[16, 17] also uses ipvs for internal load balancing, but it is also considered as part of Docker swarm infrastructure, and thus lacks the portability that our proposal aims to provide.

Load balancer tools in the container context: There are several other projects where efforts have been made to utilize ipvs in the context of container environment. For example, GORB[18] and clusterf[19] are daemons that setup ipvs rules in the kernel inside the Docker container. They utilize running container information stored in key-value storages like Core OS etcd[20] and HashiCorp's Consul[21]. Although these were usable to implement a containerized load balancer in our proposal, we did not use them, since

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

Kubernetes ingress framework already provided the methods to retrieve running container information through standard API.

Cloud load balancers: As far as the cloud load balancers are concerned, two articles have been identified. Google's Maglev[3] is a software load balancer used in Google Cloud Platform(GCP). Maglev uses modern technologies including per flow ECMP and kernel bypass for user space packet processing. Maglev serves as the GCP's load balancer that is used by the Kubernetes. Maglev is not a product that users can use outside of GCP nor is an open source software, while the users need open source software load balancer that is runnable even in on-premise data centers. Microsoft's Ananta[4] is another software load balancer implementation using ECMP and windows network stack. Ananta can be solely used in Microsoft's Azure cloud infrastructure[4]. The proposed load balancer by the author is different in that it is aimed to be used in every cloud provider and on-premise data centers.

This chaper provides background information that are important in this research. First two of the most popular overlay networks used in Kubernetes are explaind in detail. Then the author explain how to utilize multicore CPUs for packet proccessing in Linux.

2.2 Overlay Network

2.2.1 Flannel

We used flannel to build the Kubernetes cluster used in our experiment. Flannel has three types of backend, *i.e.*, operating modes, named host-gw, vxlan, and udp[22].

In the host-gw mode, the flanneld installed on a node simply configures the routing table based on the IP address assignment information of the overlay network, which is 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 containing the destination MAC address of the latter node without changing the IP header, and send them out.

In the case of the vxlan mode, flanneld creates the Linux kernel's vxlan device, flannel.1. Flanneld will 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 identify the MAC address of flannel.1 device on the destination node, then form an Ethernet frame toward the MAC address. The vxlan then encapsulates the Ethernet frame in a UDP/IP packet with a vxlan header, after which the IP packet is eventually sent out.

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

Figure 2.1 shows the schematic diagrams of frame formats for three backends modes of the flannel overlay network. The MTU sizes in the backends, assuming the MTU size without encapsulation is 1500 bytes, are also presented. Since packets are not encapsulated in the host-gw mode, the MTU size remains 1500 bytes. An additional 50 bytes of header is used in the vxlan mode, thereby resulting in an MTU size of 1450 bytes. In the case of the udp mode, only 28 bytes of header are used for encapsulation, which results in an MTU size of 1472 bytes.

Performance of the load balancers can be influenced by the overhead of encapsulation. Thus, the host-gw mode, where there is no overhead due to encapsulation, results in the best performance levels as shown in Section **??**. However, the host-gw mode has a significant drawback that prohibit it to work correctly in cloud

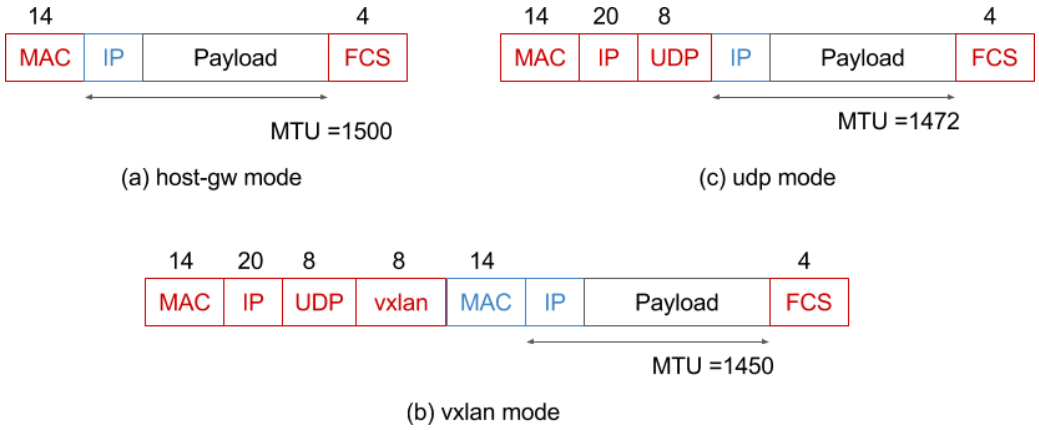


Figure 2.1: frame diagram

mode	On-premise	GCP	AWS
host-gw	OK	NG	NG
vxlan	OK	OK	OK
udp	OK	OK	OK

Table 2.1: Viable flannel backend modes. In cloud environment tunnelig using vxlan or udp is needed.

platforms. Since the host-gw mode simply sends out a packet without encapsulation, if there is a cloud gateway between nodes, the gateway cannot identify the proper destination, thus drop the packet.

We conducted an investigation to determine which of the flannel backend mode would be usable on AWS, GCP, and on-premise data centers. The results are summarized in Table 2.1. In the case of GCP, an IP address of /32 is assigned to every VM host and every communication between VMs goes through GCP's gateway. As for AWS, the VMs within the same subnet communicate directly, while the VMs in different subnets communicate via the AWS's gateway. Since the gateways do not have knowledge of the flannel overlay network, they drop the packets; thereby, they prohibit the use of the flannel host-gw mode in those cloud providers.

In our experiment, we compared the performance of load balancers when different flannel backend modes were used.

host-gw

vxlan

udp

2.2.2 Calico

no-tunnel

ipip

```
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 2.2: RX/TX queues of the hardware

2.3 Multicore Packet Proccessing

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 now includes up to 28 cores in a single CPU. In order to enjoy the benefits of multi-core CPUs in communication performance, it is necessary to distribute the handling of interrupts from the NIC and the IP protocol processing to the available physical cores.

2.3.1 rss

Receive Side Scaling (RSS)[23] is a technology to distribute handling of the interrupt from NIC queues to multiple CPU cores. Subsequently, Receive Packet Steering (RPS)[23] distributes the IP protocol processing to multiple CPU cores by issuing inter core software interrupts.

Since load balancer performance levels could be affected by these technologies, we conducted an experiment to determine how load balancer performance level change depending on the RSS and RPS settings. The following shows how RSS and RPS are enabled and disabled in our experiment. The NIC used in our experiment is Broadcom BCM5720, which has four rx-queues and one tx-queue. Figure 2.2 shows the interrupt request (IRQ) number assignments 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 [23], the CPU cores allowed to be notified is controlled by setting a 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. Further, 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, `RSS = off` means that an interrupt from any 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
```

2.3.2 rps

The RPS distributes IP protocol processing 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
```

Since the hexadecimal value "fefe" represented as "1111 1110 1111 1110" in binary, 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, `RPS = off` means that no CPU is allowed for RPS. Here, the IP protocol processing is performed on the CPUs 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 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.

2.3.3 rps

2.3.4 Others

rfs

xps

xfs

2.4 Other parameters

2.4.1 tcp congestion mode

2.5 Cloud Load Balancers

2.5.1 Maglev

2.5.2 Ananta

2.5.3 GCP Load Balancer

GCP experimental data.

2.6 ipvs

2.7 Summary

Chapter 3

Load Balancer Architecture

This chapter provides discussion of load balancer suitable for container clusters. First we discuss problems of conventions architecture in Section 3.1. Then we discuss architectual choices and propose the best one in Section 3.2. After that we discuss the how to inplement a portable load balancer in Section 3.2.1. Finally we discuss the routig and redundancy architecture in Section 3.2.2.

3.1 Problems of Kuberenetes

Problems commonly occur when the Kubernetes container management system is used outside of recommended cloud providers(such as GCP or AWS). Figure 3.1 shows an exemplified Kubernetes cluster. A Kubernetes cluster typically consists of a master and nodes. They can be physical servers or VMs. 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*, depending the PodSpec information obtained from the apiserver on the master. A *pod* is a group of containers that share same net name space and cgroups, and is the basic execution unit in a Kubernetes cluster.

When a service is created, the master schedules where to run *pods* and kubelets on the nodes launch them accordingly. At the same time, the master sends out requests to cloud provider API endpoints, asking them to set up external cloud load balancers. The proxy daemon on the nodes also setup iptables DNAT[5] rules. The Internet traffic will then be evenly distributed by the cloud load balancer to nodes, after which it will be distributed again by the DNAT rules on the nodes to the designated *pods*. The returning packets follows the exact same route as the incoming ones.

This architecture has the followings problems: 1) There must exist cloud load balancers whose APIs are supported by the Kubernetes daemons. There are numerous load balancers which is not supported by the Kubernetes. These include the bare metal load balancers for on-premise data centers. 2) Distributing the traffic twice, first on the external load balancers and second on each node, complicates the administration of packet routing. Imagine a situation in which the DNAT table on one of the nodes malfunctions. In such a case, only occasional timeouts would be observed, which would make it very difficult to find out which node was malfunctioning.

Regarding the first problem, if there is no load balancer that is not supoorted by Kubernetes, users might be able to set up the routing manually depending on the infrastructure. The traffic would be routed to a node then distributed by the DNAT rules on the node to the designated *pods*. However, this approach significantly degrades the portability of container clusters.

In short, 1) Kubernetes can be used only in limited environments where the external load balancers are supported, and 2) the routes incoming traffic follow are very complex. In order to address these problems, we propose a containerized software load balancer that is deployable in any environment even if there are no external load balancers.

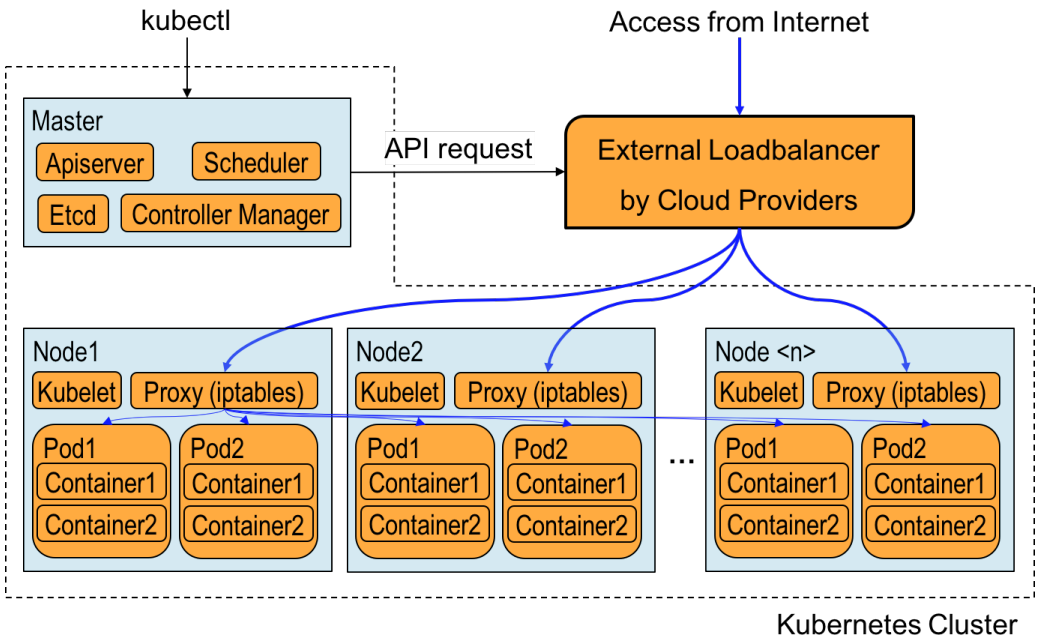


Figure 3.1: Conventional architecture of a Kubernetes cluster.

3.2 Proposed Architecture

This section discusses the load balancer architecture. How they are implemented and how redundancy is realized.

The problems of Kubernetes architecture in Figure 3.1 have been the followings; 1) There are environments with load balancers whose APIs are not supported by Kubernetes. 2) Incoming traffic is distributed twice, once at the load balancer and once at every node.

The author proposes a load balancer architecture, where a cluster of load balancers are deployed as a cluster of containers.

Figure 3.2 shows the proposed laod balancer architecture for Kubernetes, which has the following characteristics: 1) Each load balancer itself is run as a *pod* by Kubernetes. 2) Balancing tables are dynamically updated based on information about running *pods*. 3) There exist multiple load balancers for redundancy. 4) The routes to load balancers in the upstream router are updated dynamically. The proposed load balancer can resolve the conventional architecture problems, as follows: Since the load balancer itself is containerized, the load balancer can run in any environment including on-premise data centers, even without external load balancers that is supported by Kubernetes. Load balancers can share the server pool with web containers. The incoming traffic is directly distributed to designated *pods* by the load balancer. It makes the administration, e.g. finding malfunctions, easier than the conventional architecture.

There are several other possible ways to solve these problems. a) Make Kubernetes support all of the existing load balancer hardware that could be used in on-premise data centers. b) Force users to buy new hardware load balancer that is supported by Kubernetes. c) Provide software load balancers that function similarly as the cloud load balancers.

The a) is impossible. The b) does not improve the usability of the container cluster since the specific hardware is always needed. The c) seems viable solution since they can be realized using commodity hardware and the author thinks it is worth while investigating in the future.

However, there is a reason why the author chose the proposed architecture as the best candidate. The requirements for the load balancer are exactly the benefits a container cluster can provide. These are

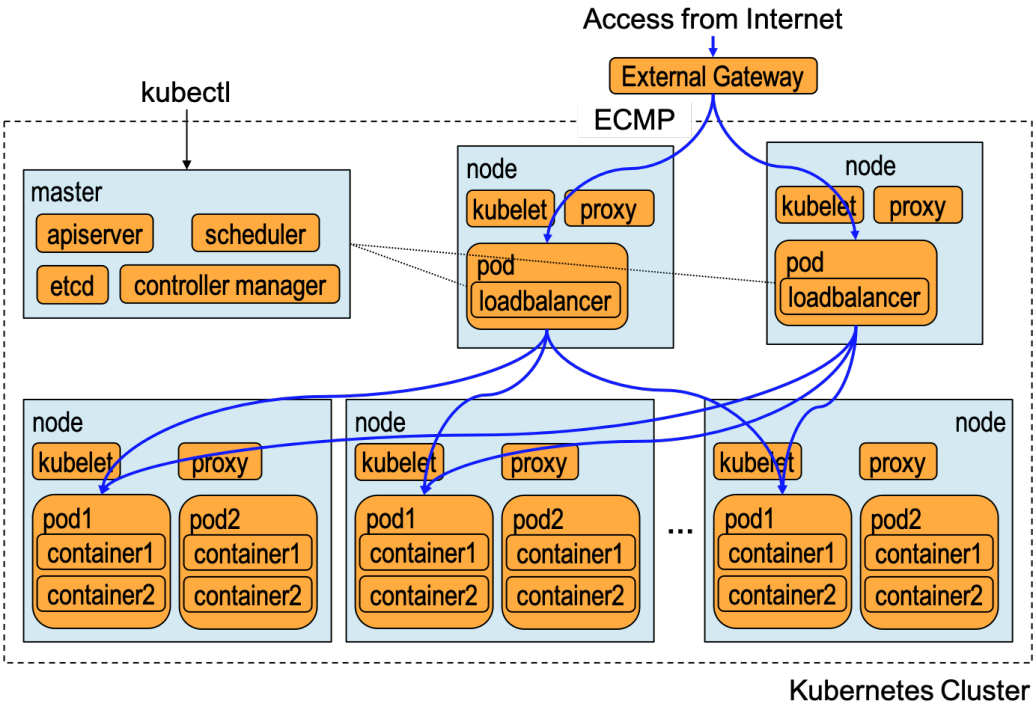


Figure 3.2: Kubernetes cluster with proposed load balancer.

portability, elasticity(scalability) and redundancy. Load balancers should exist any environment and behave the same manner everywhere. The load balancer should be able to change the performance depending on the demands. The load balancer should never fail; thus multiple instances should always be running. It is also beneficial if the load balancer can share the same server pool with web servers since this will ease the capacity planning.

3.2.1 Portable Load Balancer

In order to demonstrate a software load balancer that is runnable in any environment, the ipvs is containerized. In addition to that the proposed load balancer uses two other components, keepalived, and a controller. These components are placed in a single Docker container image. The ipvs is a Layer-4 load balancer capability, which is included in the Linux kernel 2.6.0 released in 2003 or later, to distribute incoming Transmission Control Protocol(TCP) traffic to *real servers*[1] [7]. For example, ipvs distributes incoming Hypertext Transfer Protocol(HTTP) traffic destined for a single destination IP address, to multiple HTTP servers(e.g. Apache HTTP or nginx) running on multiple nodes in order to improve the performance of web services. Keepalived is a management program that performs health checking for *real servers* and manages ipvs balancing rules in the kernel accordingly. It is often used together with ipvs to facilitate ease of use. The controller is a daemon that periodically monitors the *pod* information on the master, and it performs various actions when such information changes. Kubernetes provides ingress controller framework as the Go Language(Golang) package to implement the controllers. We implement a controller program that feeds *pod* state changes to keepalived using this framework.

[1]The term, *real servers* refers to worker servers that will respond to incoming traffic, in the original literature[7]. We will also use this term in the similar way.

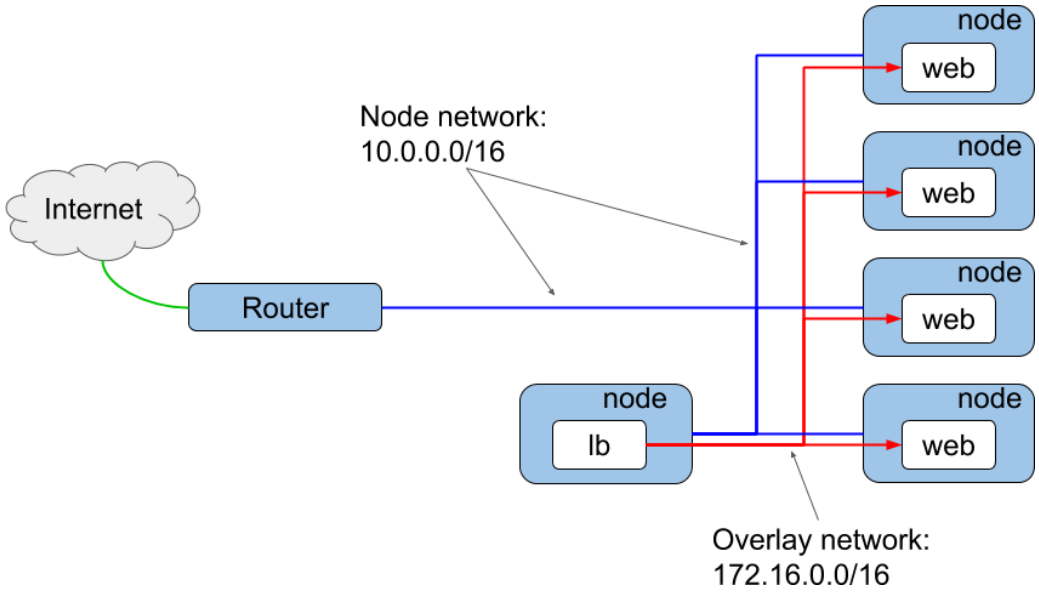


Figure 3.3: The network architecture of an exemplified container cluster system.
A load balancer(lb) pod(the white box with "lb") and web pods are running on nodes(the blue boxes). The traffic from the internet are forwarded to the lb pod by the upstream router using the node network, and the distributed to web pods using the overlay network.

3.2.2 Routing and Redundancy

While containerizing ipvs makes it runnable in any environment, it is essential to discuss how to route the traffic to the ipvs container. We propose redundant architecture using ECMP for load balancer containers usable especially in on-premise data centers. We first explain overlay network briefly in **??** as background, then present the proposed architecture with ECMP redundancy in **??**. We also present an alternative architecture using VRRP as a comparison in **??**, which we think is not as good as the architecture using ECMP.

Overlay Network

In order to discuss load balancer for container cluster, the knowledge of the overlay network is essential. We briefly explain an abstract concept of overlay network in this subsection.

Fig. 3.3 shows schematic diagram of network architecture of a container cluster system. Suppose we have a physical network(node network) with IP address range of 10.0.0.0/16 and an overlay network with IP address range of 172.16.0.0/16. The node network is the network for nodes to communicate with each other. The overlay network is the network setups for containers to communicate with each other. An overlay network typically consists of appropriate routing tables on nodes, and optionally of tunneling setup using ipip or vxlan. The upstream router usually belongs to the node network. When a container in the Fig. 3.3 communicates with any of the nodes, it can use its IP address in 172.16.0.0/16 IP range as a source IP, since every node has proper routing table for the overlay network. When a container communicates with the upstream router that does not have routing information regarding the overlay network, the source IP address must be translated by Source Network Address Translation(SNAT) rules on the node the container resides.

The SNAT caused a problem when we tried to co-host multiple load balancer containers for different services on a single node, and let them connect the upstream router directly. This was due to the fact that the BGP agent used in our experiment only used the source IP address of the connection to distinguish the BGP peer. The agent behaved as though different BGP connections from different containers belonged to a single BGP session because the source IP addresses were identical due to the SNAT.

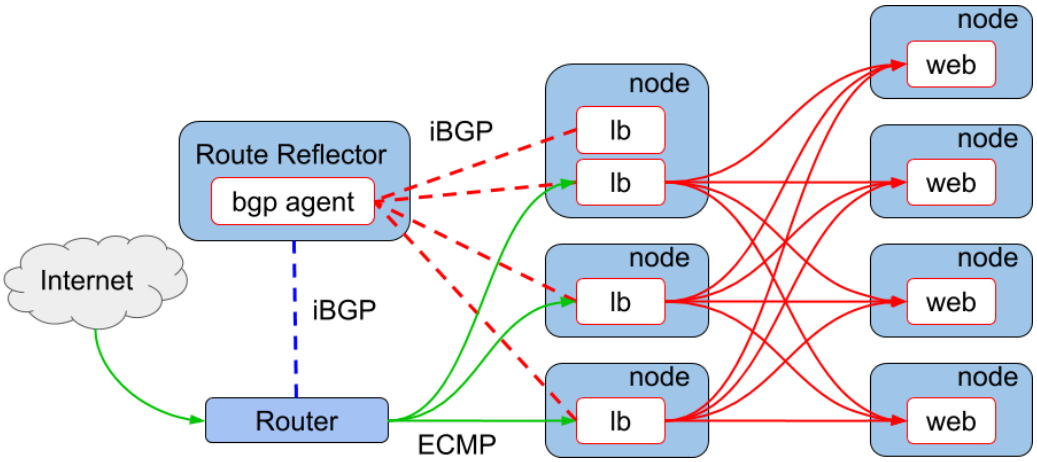


Figure 3.4: The proposed architecture of load balancer redundancy with ECMP.
The traffic from the internet is distributed by the upstream router to multiple of lb pods using hash-based ECMP and then distributed by the lb pods to web pods using Linux kernel's ipvs. The ECMP routing table on the upstream router is populated using iBGP.

There many overlay network implementations. The author investigated two of the popular ones to see how it works.

ECMP

Fig. 3.4 shows our proposed redundancy architecture with ECMP for software load balancer containers. The ECMP is a functionality a router often supports, where the router has multiple next hops with equal cost(priority) to a destination, and generally distribute the traffic depending on the hash of the flow five tuples(source IP, destination IP, source port, destination port, protocol). The multiple next hops and their cost are often populated using the BGP protocol. The notable benefit of the ECMP setup is the fact that it is scalable. All the load balancers that claims as the next hop is active, i.e., all of them are utilized to increase the performance level. Since the traffic from the internet is distributed by the upstream router, the overall throughput is determined by the router after all. However, in practice, there are a lot of cases where this architecture is beneficial. For example, if a software load balancer is capable of handling 1 Gbps equivalent of traffic and the upstream router is capable of handling 10 Gbps, it still is worthwhile launching 10 of the software load balancer containers to fill up maximum throughput of the upstream router.

We place a node with the knowledge of the overlay network as a route reflector, to deal with the complexity due to the SNAT. A route reflector is a network component for BGP to reduce the number of peerings by aggregating the routing information[?]. In our proposed architecture we use it as a delegater for load balancer containers towards the upstream router.

By using the route reflector, we can have the following benefits. 1) Each node can accommodate multiple load balancer containers. This was not possible when we tried to directly connect load balancers and the router through SNAT. 2) The router does not need to allow peering connections from random IP addresses that may be used by load balancer containers. Now, the router only need to have the reflector information as the BGP peer definition.

Since we use standard Linux boxes for route reflectors, we can configure them as we like; a) We can make them belong to overlay network so that multiple BGP sessions from a single node can be established. b) We can use a BGP agent that supports dynamic neighbor (or dynamic peer), where one only needs to define the IP range as a peer group and does away with specifying every possible IP that load balancers may use.

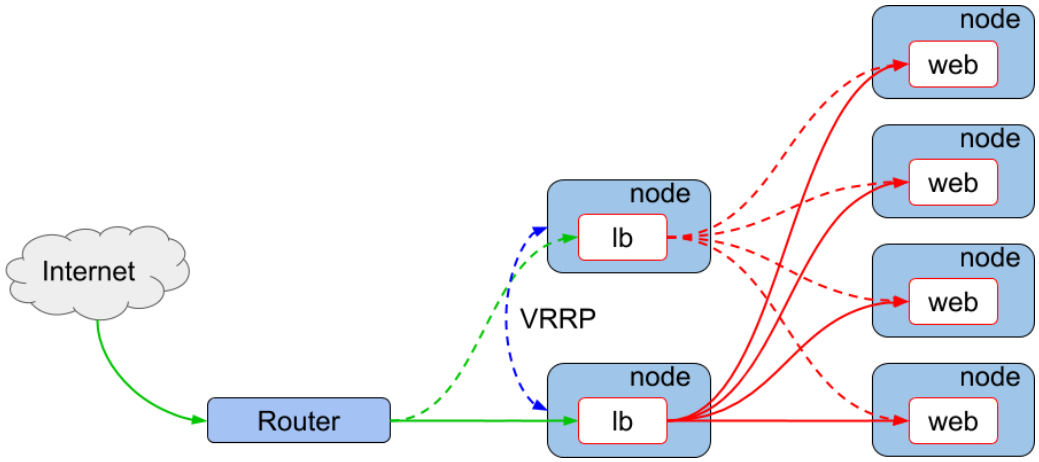


Figure 3.5: An alternative redundant load balancer architecture using VRRP.
The traffic from the internet is forwarded by the upstream router to a active lb node and then distributed by the lb pods to web pods using Linux kernel's ipvs. The active lb pod is selected using VRRP protocol.

The upstream router does not need to accept BGP sessions from containers with random IP addresses, but only from the router reflector with well known fixed IP address. This may be preferable in terms of security especially when a different organization administers the upstream router. Although not shown in the Fig. 3.4, we could also place another route reflector for redundancy purpose.

VRRP

Fig. 3.5 shows an alternative redundancy setup using the VRRP protocol that was first considered by the authors, but did not turn out to be preferable. In the case of VRRP, the load balancer container needs to run in the node net namespace for the following two reasons. 1) When fail over occurs, the new master sends gratuitous Address Resolution Packets(ARP) packets to update the ARP cache of the upstream router and Forwarding Data Base(FDB) of layer 2 swicthes during the transition. Such gratuitous ARP packets should consist of the virtual IP address shared by the load balancers and the MAC address of the node where the new master load balancer is running. Programs that send out gratuitous ARP with node MAC address should be in the node net namespace. 2) Furthermore, the active load balancer sends out periodic advertisement using UDP multicast packet to inform existence of itself. The load balancer in backup state stays calm unless the VRRP advertisement stops for a specified duration of time. The UDP multicast is often unsupported in overlay network used by container cluster environment, and hence the load balancer needs to be able to use the node net namespace. Running containers in the node net namespace loses the whole point of containerization, i.e., they share the node network without separation. This requires the users' additional efforts to avoid conflict in VRRP configuration for multiple services.

VRRP programs also support unicast advertisement by specifying IP addresses of peer load balancers before it starts. However, container cluster management system randomly assign IP addresses of containers when it launches them, and it is impossible to know peer IPs in advance. Therefore the unicast mode is not feasible in container cluster environment.

The other drawback compared with the ECMP case is that the redundancy of VRRP is provided in Active-Backup manner. This means that a single software load balancer limits the overall performance of the entire container cluster. Therefore we believe the ECMP redundancy is better than VRRP in our use cases.

3.3 Summary

In this chapter the followings have been discussed....

Chapter 4

Implementation

This chapter presents implementation of the proof of the concept system for the proposed load balancer architecture in detail. First overall architecure is explained in Section **??**. Then ipvs containerization is explained in detail in Section **??**. Finally implementation of BGP software container is explained in Section **??**.

4.1 Proof of concept system architecture

Fig. 4.1 shows the schematic diagram of proof of concept container cluster system with our proposed redundant software load balancers. All the nodes and route reflector are configured using Debian 9.5 with self compiled linux-4.16.12 kernel. The upstream router also used conventional linux box using the same OS as the nodes and route reflector. For the Linux kernel to support hash based ECMP routing table we needed to use kernel version 4.12 or later. We also needed to enable kernel config option CONFIG_IP_ROUTE_MULTIPATH[**?**] when compiling, and set the kernel parameter fib_multipath_hash_policy=1 at run time. In the actual production environment, proprietary hardware with the highest throughput is often deployed, but we could still test some of the required advanced functions by using a Linux box.

Each load balancer pod consists of an exabgp container and an ipvs container. The ipvs container is responsible for distributing the traffic toward the IP address that a service uses, to web server(nginx) pods. The ipvs container monitors the availability of web server pods and manages the load balancing rule appropriately. The exabgp container is responsible for advertising the route toward the IP address that a service uses, to the route reflector. The route reflector aggregates the routing information advertised by load balancer pods and advertise them to the upstream router.

The exabgp is used in the load balancer pods because of the simplicity in setting as static route advertiser. On the other hand, gobgp is used in the router and the route reflector, because exabgp did not seem to support add-path[**?**] needed for multi-path advertisement and Forwarding Information Base(FIB) manipulation[**?**]. The gobgp supports the add-path, and the FIB manipulation through zebra[**?**]. The configurations for the router is summarised in B.3.

The route reflector also uses a Linux box with gobgp and overlay network setup. The requirements for the BGP agent on the route reflector are dynamic-neighbours and add-paths features. The configurations for the route reflector is summarised in B.2.

4.2 Ipvs container

The proposed load balancer needs to dynamically reconfigure the IPVS balancing rules whenever *pods* are created/deleted. Figure 4.2 is a schematic diagram to show the dynamic reconfiguration of the IPVS rules. The right part of the figure shows the enlarged view of one of the nodes where the load balancer pod(LB2) is deployed. Two daemon programs, controller and keepalived, run in the container inside the LB2 pod are illustrated. The keepalived manages Linux kernel's IPVS rules depending on the ipvs.conf configuration file. It is also capable of health-checking the life of *real server*, which is represented as a combination of the IP addresses and port numbers of the target *pods*. If the health check to a *real server* fails, keepalived will remove

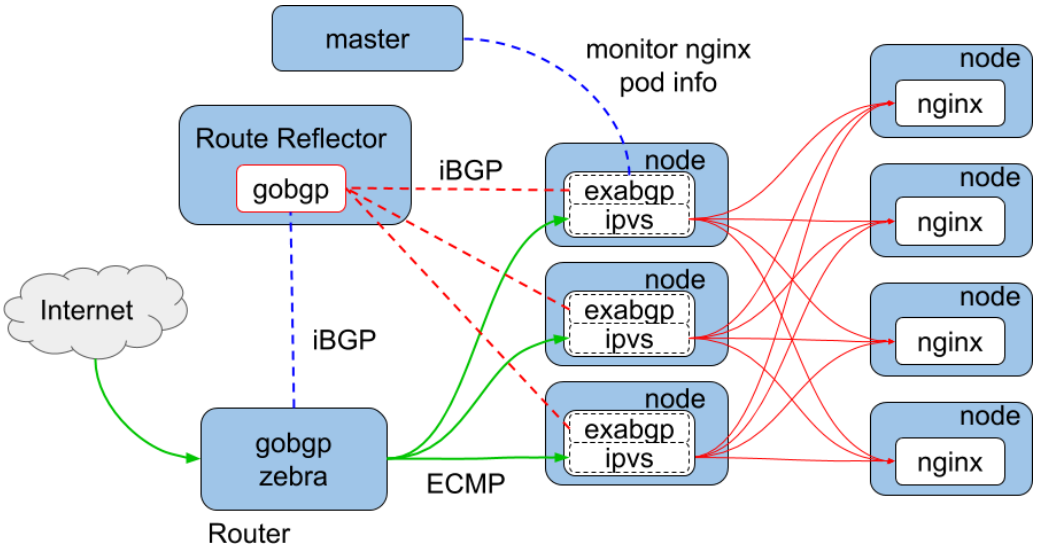


Figure 4.1: An experimental container cluster with proposed redundant software balancers.
The master and nodes are configured as Kubernetes's master and nodes on top of conventional Linux boxes, respectively. The route reflector and the upstream router are also conventional Linux boxes.

that *real server* from the IPVS rules.

The controller monitors information concerning the running *pods* of a service in the Kubernetes cluster by consulting the apiserver running on the master. Whenever *pods* are created or deleted, the controller will automatically regenerate an appropriate ipvs.conf and issue SIGHUP to keepalived. Then, keepalived will reload the ipvs.conf and modify the kernel's IPVS rules accordingly. The actual controller[24] is implemented using the Kubernetes ingress controller[10] framework. By importing existing Golang package, "k8s.io/ingress /core/pkg/ingress", we could simplify the implementation, e.g. 120 lines of code.

Configurations for capabilities were 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 keepalived to manipulate the kernel's IPVS rules. For the former case, we also needed to mount the "/lib/module" of the node's file system on the container's file system.

Figure 4.3 and Figure 4.4 show an example of an ipvs.conf file generated by the controller and the corresponding IPVS load balancing rules, respectively. Here, we can see that the packet with `fwmark=1`[25] is distributed to `172.16.21.2:80` and `172.16.80.2:80` using the masquerade mode(Masq) and the least connection(lc)[7] balancing algorithm.

4.3 BGP software container

In order to implement the ECMP redundancy, we also containerized exabgp using Docker. Fig.4.5 (a) shows a schematic diagram of the network path realized by the exabgp container. We used exabgp as the BGP advertiser as mentioned earlier. The traffic from the Internet is forwarded by ECMP routing table on the router to the node, then routed to ipvs container.

Fig.4.5 (??) summarises some key settings required for the exabgp container. In BGP announcements the node IP address, 10.0.0.106 is used as the next-hop for the IP range 10.1.1.0/24. Then on the node, in order to route the packets toward 10.1.1.0/24 to the ipvs container, a routing rule to the dev docker0 is created in the node net namespace. A routing rule to accept the packets toward those IPs as local is also required in the

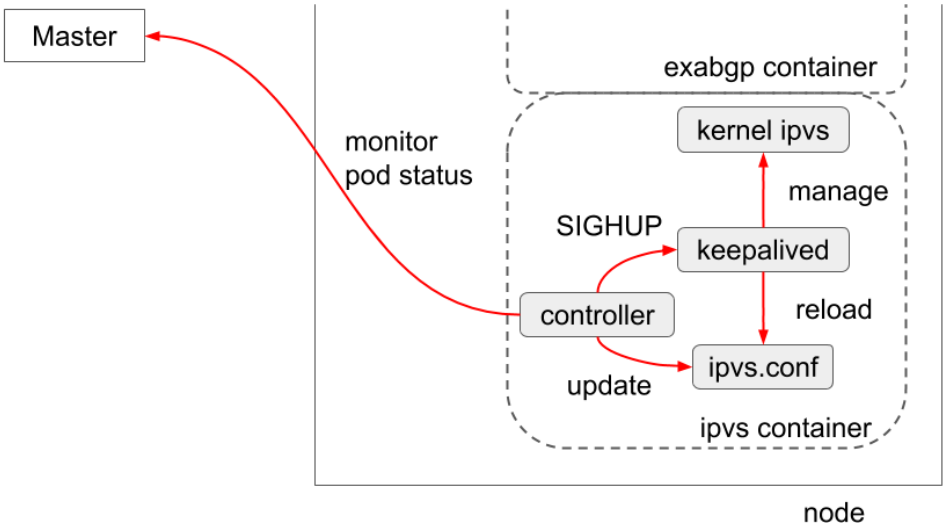


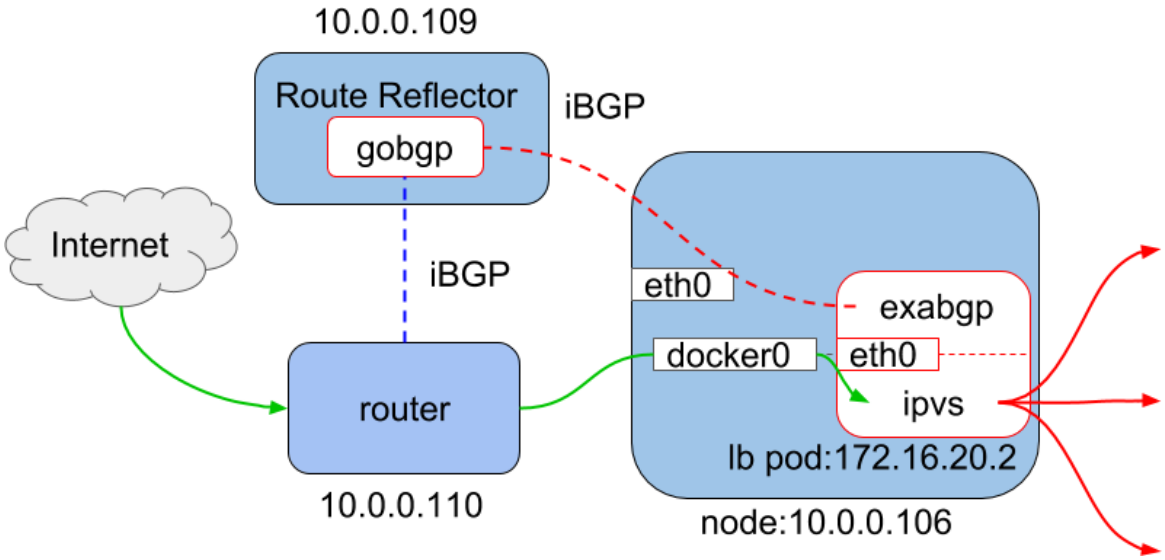
Figure 4.2: Implementation

```
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 4.3: 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 4.4: Example of IPVS balancing rules



(a)

[BGP announcement]
route 10.1.1.0/24 next-hop 10.0.0.106
[Routing in node net namespace]
ip netns exec node ip route replace 10.1.1.0/24 dev docker0
[Accept as local]
ip route add local 10.1.1.0/24 dev eth0

(b)

Figure 4.5: (a) Network path by the exabgp container. (b) Required settings in the exabgp container.

container net namespace. A configuration of exabgp is shown in B.1.

4.4 ingress controller

4.5 choice bgp software

4.6 Summary

Chapter 5

Evaluation of a portable load balancer

This chapter discusses portability and performance level of a single ipvs load balancer in 1 Gbps environments. First the author investigated general characteristics of a single load balancer using physical servers in on-premise data center and compared performance level with existing iptables DNAT and nginx as a load balancer. Then the author also carried out the performance measurement in GCP and AWS to show that the containerized ipvs load balancer is runnable and has the same characteristics in the cloud environment. The following sections explain these in further detail.

5.1 Throughput measurement for ipvs-nat Load balancer

5.1.1 Benchmark method

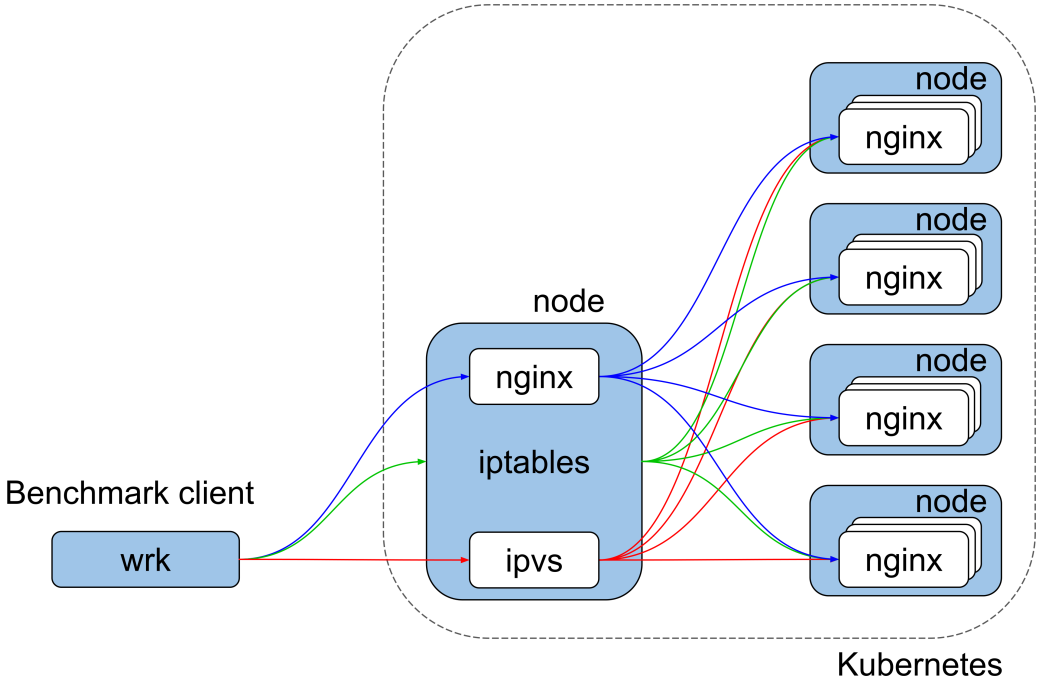
A set of throughput measurement was carried out using an HTTP benchmark program, wrk[26]. Figure 5.1(a) illustrates a schematic diagram of the experimental setup. Multiple *pods* are deployed on multiple nodes in the Kubernetes cluster. In each *pod*, an nginx web server pod that returns the IP address of the *pod* are running. The author set up the ipvs, iptables DNAT, and nginx load balancers on one of the nodes. All the nodes and the benchmark client are connected to a 1Gbps network switch as in Figure 5.1(b).

The throughput, Request/sec, is measured cluster as follows: The HTTP GET requests are sent out by the wrk on the client machine toward the nodes, using destination IP addresses and port numbers that are chosen based on the type of the load balancer on which the measurement is performed. The load balancer on the node then distributes the requests to the *pods*. Each *pod* returns HTTP responses to the load balancer, after which the load balancer returns them to the client. Based on the number of responses received by wrk on the client, load balancer performance, in terms of Request/sec can be obtained.

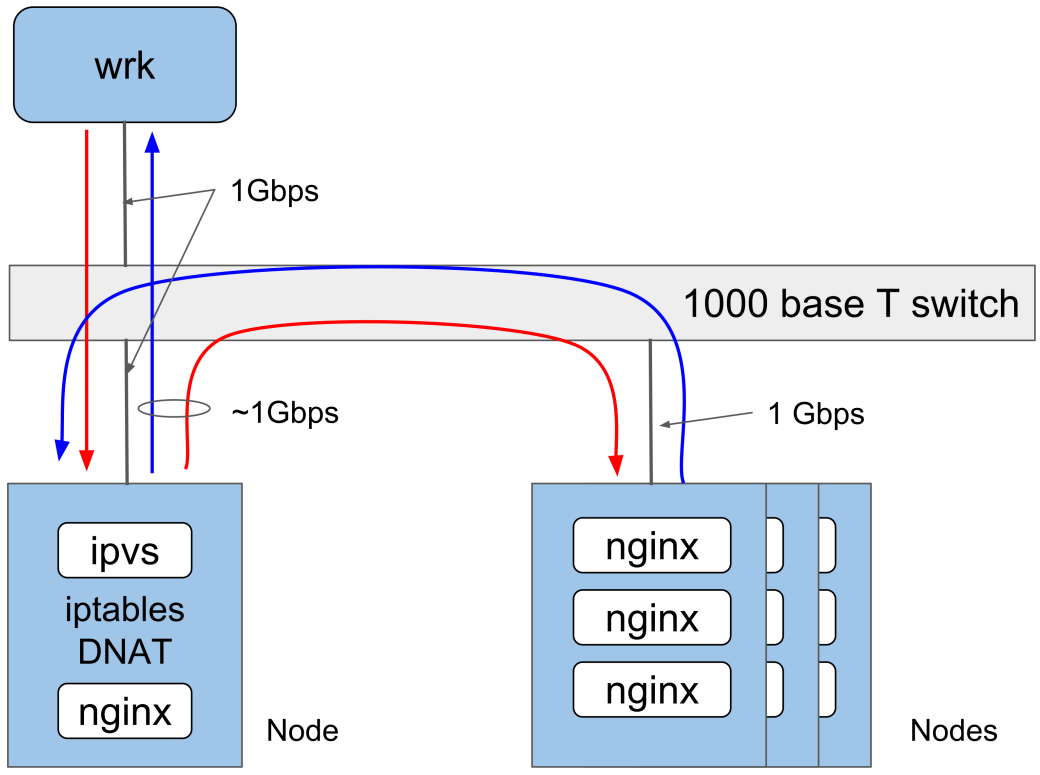
Table 5.1 shows an example of the command-line for wrk and the corresponding output. The command-line in Table 5.1 will generate 40 wrk program threads and allow those threads to send out a total of 800 concurrent HTTP requests over the period of 30 seconds. The output example shows the information including per thread statistics, error counts, Request/sec and Transfer/sec.

Table 5.2 shows hardware and software configuration used in the experiments. All of the nginx web server pods are configured to return the IP address of the *pod*, in order to make them return a small HTTP content. This makes a relatively severe condition for load balancers. The size of the character string making up an IP address is limited to 15 bytes. If the author had chosen the HTTP response size so that most of the IP packet resulted in maximum transmission unit(MTU), the performance would have been dominantly limited by the Ethernet bandwidth.

For this experiment a total of eight servers are used; six servers for nodes, one for the load balancer and one for the benchmark client, with all having the same hardware specifications. The software versions used for Kubernetes, web server and load balancer *pods* are also summarized in the Table 5.2. The hardware we used had eight physical CPU cores and a 1Gbps NIC with 4 rx-queues.



(a) Logical configuration.



(b) Physical configuration.

Figure 5.1: Benchmark setup.

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

[Output example]
 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
```

Table 5.1

[Hardware Specification]
CPU: Xeon E5-2450 2.10GHz (with 8 core, Hyper Threading)
Memory: 32GB
NIC: Broadcom BCM5720 Giga bit
(Node x 6, LB x 1, Client x 1)

[Node Software]
OS: Debian 8.7, linux-3.16.0-4-amd64
Kubernetes v1.10.6
flannel v0.7.0
etcd version: 3.0.15

[Container Software]
Keepalived: v1.3.2 (12/03,2016)
nginx : 1.11.1(load balancer), 1.13.0(web server)

Table 5.2

5.1.2 Effect of multicore proccessing

Figure 5.2 shows a result of throughput experiment with different multicore proccessing settings. The following three RSS and RPS settings were compared:

```
(RSS, RPS) = (off, off)
           = (on , off)
           = (off, on )
```

The case with "(RSS, RPS) = (off, off)" means that multicore packet processing is completely disabled, i.e., all the incoming packets are processed by a single core. The "(RSS, RPS) = (on, off)" means that the interrupt handling and the following IP protocol processing are performed on four of the CPU cores by assigning four rx-queues to those cores. In this case four of the eight CPU cores are utilized. The "(RSS, RPS) = (off, on)" means that a single core handles all of the interrupts from the NIC then the following IP processings are performed on the other cores. In this case, all of the eight CPU cores are utilized.

We can see a general trend in which the throughput linearly increases as the number of nginx *pod*s increases and then it eventually saturates. The saturated throughput levels indicate the maximum performance

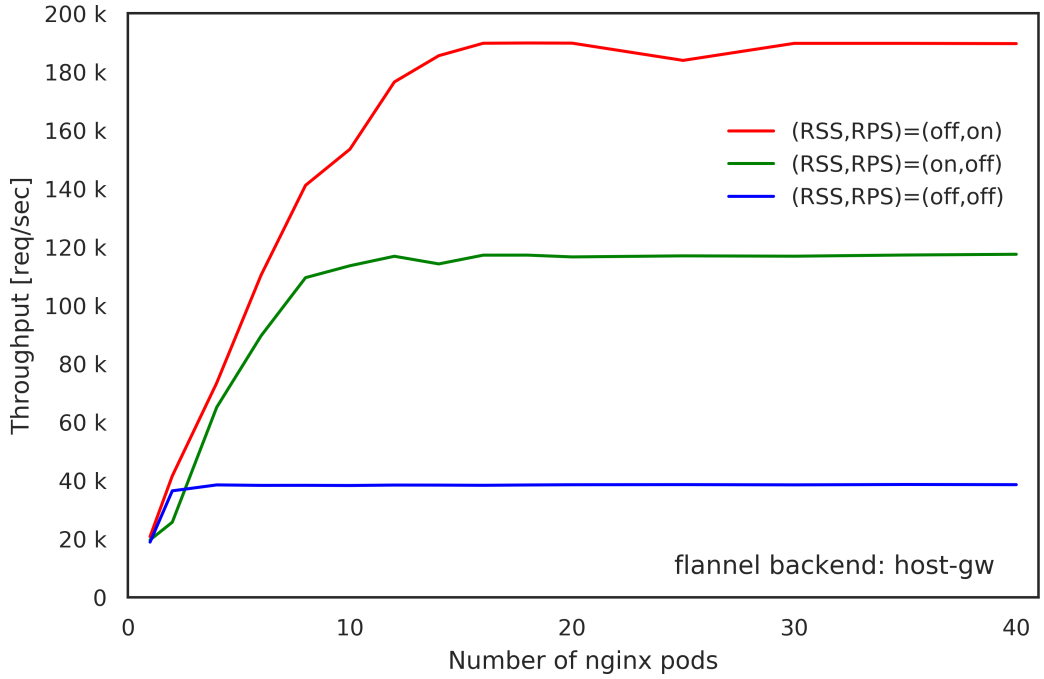


Figure 5.2: Effect of multicore proccessing on ipvs throughput.

level of the ipvs load balancer. The maximum performance levels depend on the (RSS, RPS) settings. From the results in this figure, it can be seen that if we turn off multicore packet processing, *i.e.*, when "(RSS, RPS) = (off, off)", performance degrades significantly.

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 clear that the case that utilizes all of the CPU cores better performs than the case with only four CPU cores utilized.

At first, it was not clear what caused the performance limit for the case when "(RSS, RPS) = (off, on)", the author thought it was due to the insufficient CPU performance. However, that was not the case in the conditions of the experiment; it turned out to be due to the 1Gbps bandwidth. A packet level analysis using tcpdump[27] revealed that 665.36 bytes of extra HTTP headers, TCP/IP headers and ethernet frame headers are needed for each request in the case of the wrk benchmark program(Appendix C). This results in the upper limit of 184,267 [req/sec] when the date size of HTTP response body is 13 byte, which agrees well with the performance limit for the case when "(RSS, RPS) = (off, on)" in Figure 5.2. Figure 5.3 shows the theoretical upper limit of the performance level for 1Gbps ethernet together with actual benchmark results for the range of larger data sizes, and they agree very well. Therefore it can be said that when "RPS = on", ipvs performance is limited by 1Gbps bandwidth. The author regarded that "(RSS, RPS) = (off, on)" is the best setting in our experimental conditions, and used this setting throughout this thesis unless explicitly stated otherwise.

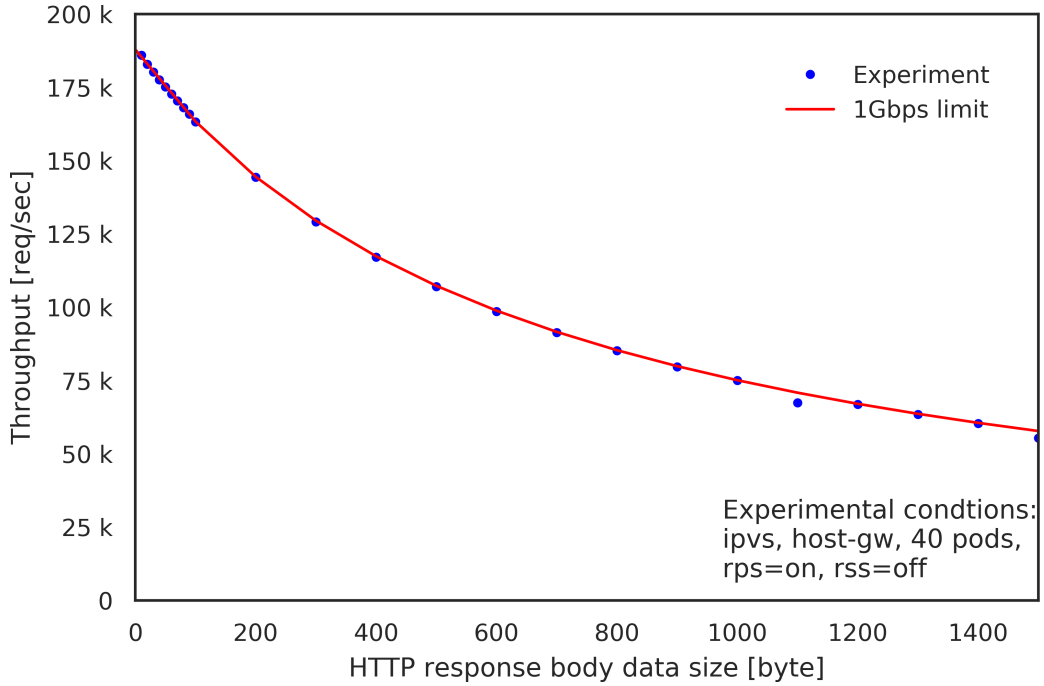


Figure 5.3: Performance limit due to 1Gbps bandwidth

5.1.3 Effect of overlay network

Figure 5.4 shows the ipvs throughput results for different overlay network settings. As for the overlay network, the author used the flannel and measured the performance levels for flannel's three backend modes, host-gw, vxlan and udp. Except for the udp backend mode case, we can see the trend in which the throughput linearly increases as the number of nginx *pod* increases and then it eventually saturates. The saturated throughput levels indicate the maximum performance levels of the ipvs load balancer. If we compare the performance levels among the flannel backend modes types, the host-gw mode where no encapsulation is conducted shows the highest performance level, followed by the vxlan mode where the Linux kernel encapsulate the Ethernet frame. The udp mode where flanneld itself encapsulate the IP packet shows significantly lower performances levels. The author considers the host-gw mode is the best, the vxlan tunnel the second best and the udp tunnel mode unusable. As is shown here, overlay network settings greately affect the performance level. The author used host-gw mode for most of the experiments conducted in on-premise data centers and vxlan mode for the experiments conducted in cloud environments.

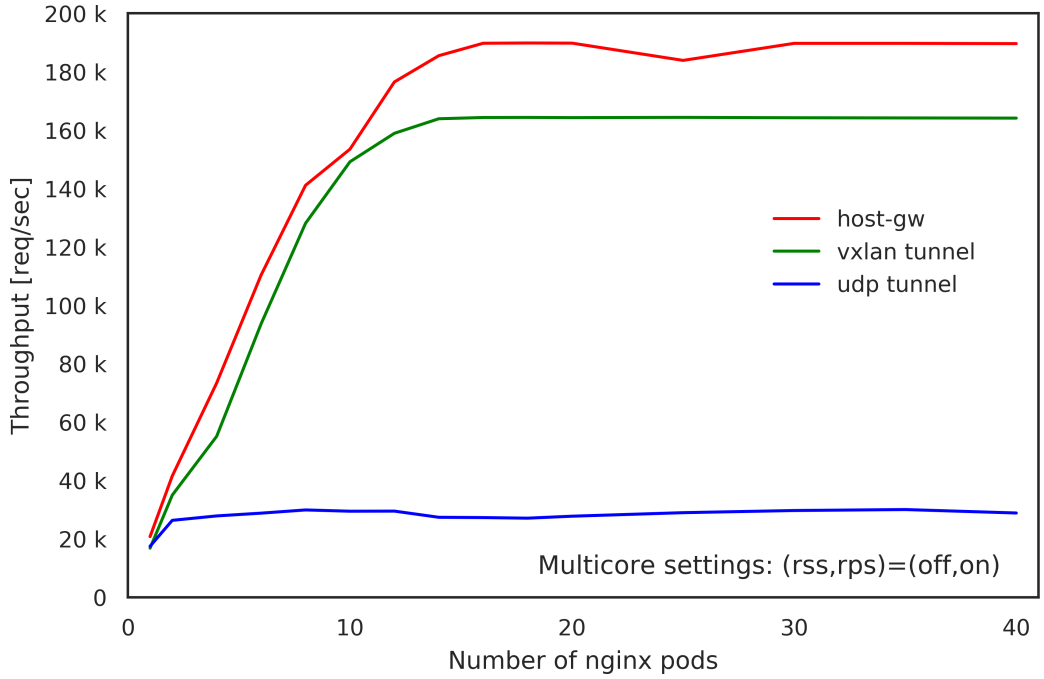


Figure 5.4: Effect of flannel backend modes on ipvs throughput.

5.1.4 Comparison of different load balancer

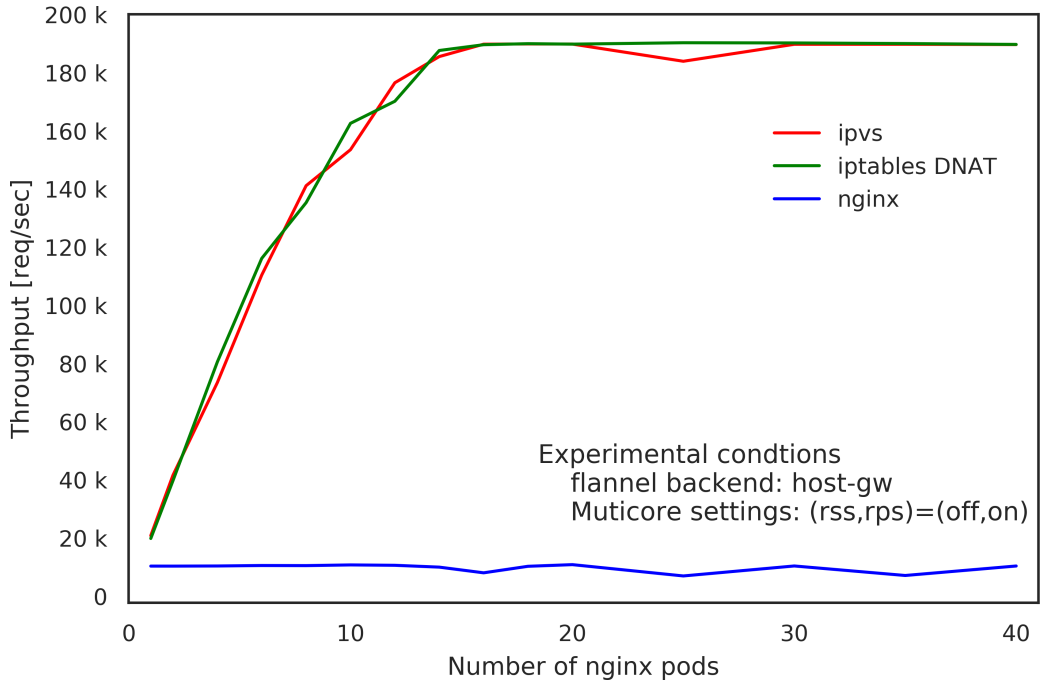


Figure 5.5: Throughput comparison between ipvs, iptables DNAT and nginx.

Figure 5.5 compares the performance measurement results for different load balancer ipvs, iptables DNAT, and nginx. The proposed ipvs load balancer exhibits almost equivalent performance levels as the iptables DNAT based load balancer. The nginx based load balancer shows no performance improvement even though the number of the nginx web server *pods* is increased. It is understandable because the performance of the single nginx as a load balancer is expected to be similar to the performance as a web server.

Figure 5.6 compares Cumulative Distribution Function(CDF) of the load balancer latency at the two constant loads, 160K[req/sec] and 180K[req/sec] for ipvs and iptables DNAT. We can see that the latencies

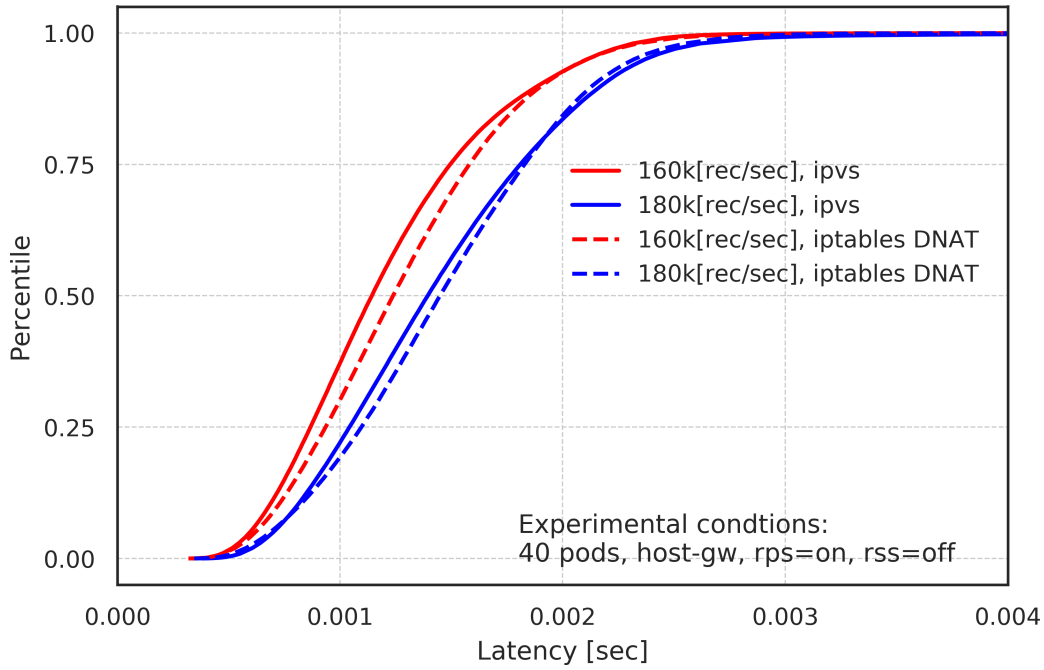


Figure 5.6: Latency cumulative distribution function.

are a little bit smaller for ipvs. For example, the median values at 160K[req/sec] load for ipvs and iptables DNAT are, 1.14 msec and 1.24 msec, respectively. Also, at 160K[req/sec], they are 1.39 msec and 1.45 msec, respectively. These may not be considered a significant difference; however, we can at least say that our proposed load balancer is as good as iptables DNAT. So, to conclude this section, the containerized ipvs load balancer showed equivalent performance levels with the iptables DNAT load-balancing function that is used in Kubernetes cluster.

5.2 L3DSR using ipvs tun

The performance levels of ipvs and iptables DNAT have been limited by 1 Gbps bandwidth. This can be alleviated in the case of ipvs by using so-called Layer 3 Direct Server Return(l3dsr) setup. Figure 5.7 shows the schematic diagram illustrating packet flow for the HTTP request packet(the red arrows) and response packet(the blue arrow).

The ipvs has the mode called ipvs-tun. When the ipvs-tun send out the packets to real servers, it encapsulates the original packet in ipip tunneling packet that is destined to real servers. The real server receives the packet on a tunl0 device and decapsulates the ipip packet, revealing the original packet. Since the source IP address of the original packet is maintained, the returning packets are sent directly toward the benchmark client. In this scheme, the returning packets do not consume the bandwidth nor the CPU power of the load balancer node.

The iptables DNAT does not have the functions that enable L3DSR settings. Therefore this one of the benefits of the ipvs load balancer.

The author carried out throughput measurement using the physical setup shown in Figure 5.7. Figure 5.8 shows the throughput of the ipvs-tun, conventional ipvs (after here the author call it ipvs-nat) and iptables DNAT. As can be seen in the figure, while the performance levels for ipvs-nat and iptables DNAT exactly match, the performance levels for ipvs-tun is greatly improved, e.g., 1.5 times larger saturated throughput than for ipvs-nat and iptables DNAT cases.

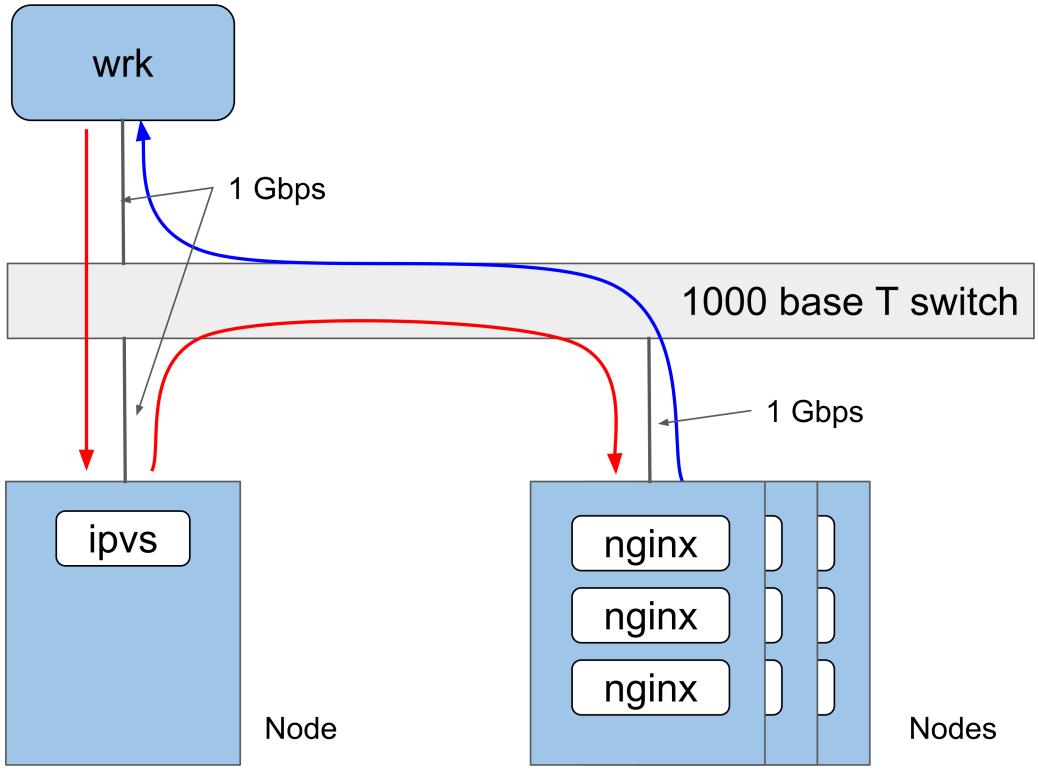


Figure 5.7: Physical configuration for L3DSR experiment.

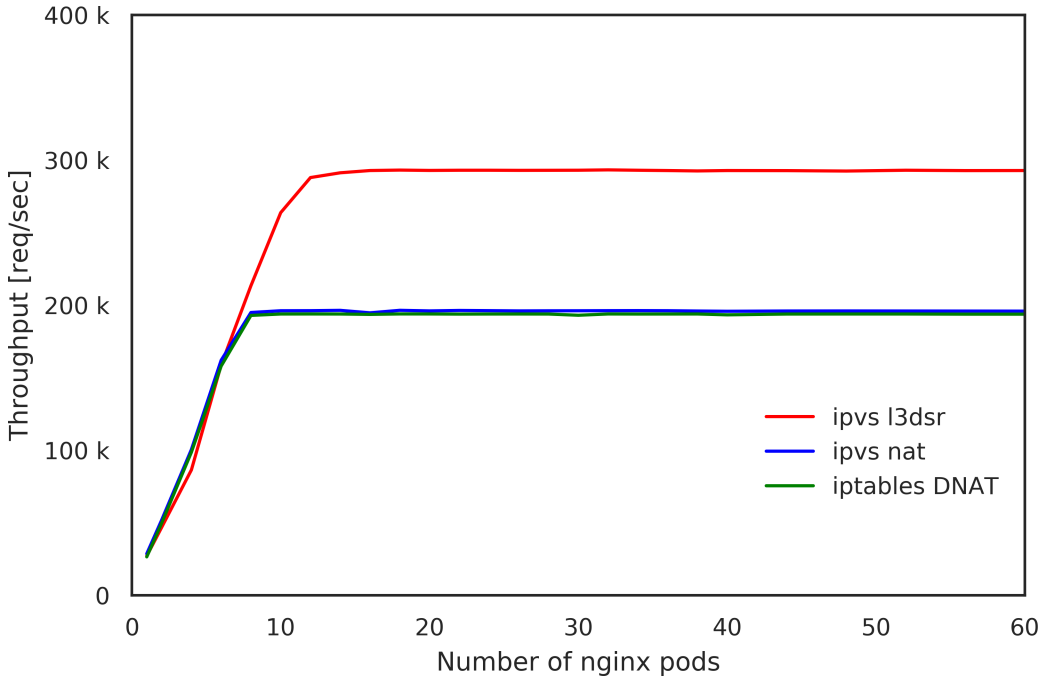


Figure 5.8: Throughput of ipvs l3dsr @1Gbps.

5.3 Cloud experiment [Add experimental conditions for GCP and AWS]

So far, it has been shown that the proposed ipvs-nat load balancer in a container has equivalent throughput, and the proposed ipvs-tun load balancer in a container has even better throughputs. In this section, the author shows that the proposed load balancer is portable by showing that it can be run in cloud environments, and also shows that it has the same behavior as in on-premise data centers.

Figure 5.9 and Figure 5.10 show the load balancer performance levels that are measured in GCP and AWS, respectively. For both environments, the author measured throughput with several conditions of CPU counts, since the machine specifications can be easily changed in the cases of cloud environments. Both results show similar characteristics as the experiment in an on-premise data center in Figure 5.2, where throughput increases linearly to a certain saturation level that is determined by utilized CPU core count. In other words, it indicates that the proposed load balancer can be run in cloud environments and also functions properly.

It seems that CPU counts determine the load balancer's throughput saturation levels. The actual throughput numbers are smaller than those of the load balancers in on-premise data centers. This may be because the physical servers in on-premise data center outperform the VMs in a cloud environment, or because network bandwidth is smaller and is limited based on the type of instances. A detailed analysis is further required in the future to clarify which factor limits the throughput in the case of the cloud environment. Nonetheless, we can say that the proposed ipvs load balancers can be run in both GCP and AWS, and the behavior is the same with the load balancers in on-premise data centers.

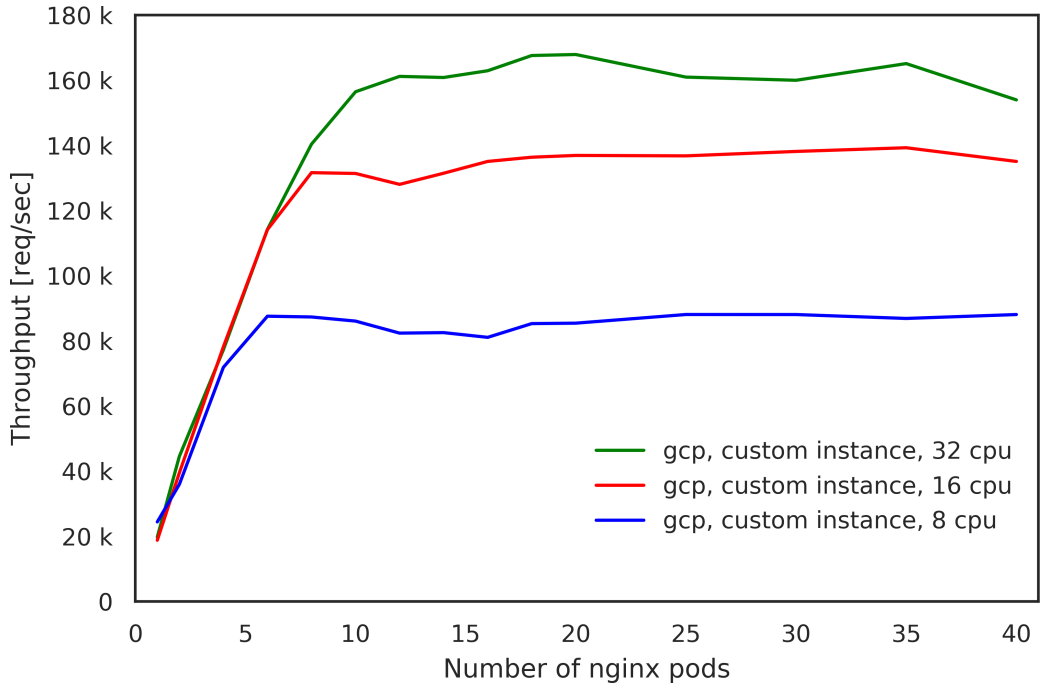


Figure 5.9: GCP

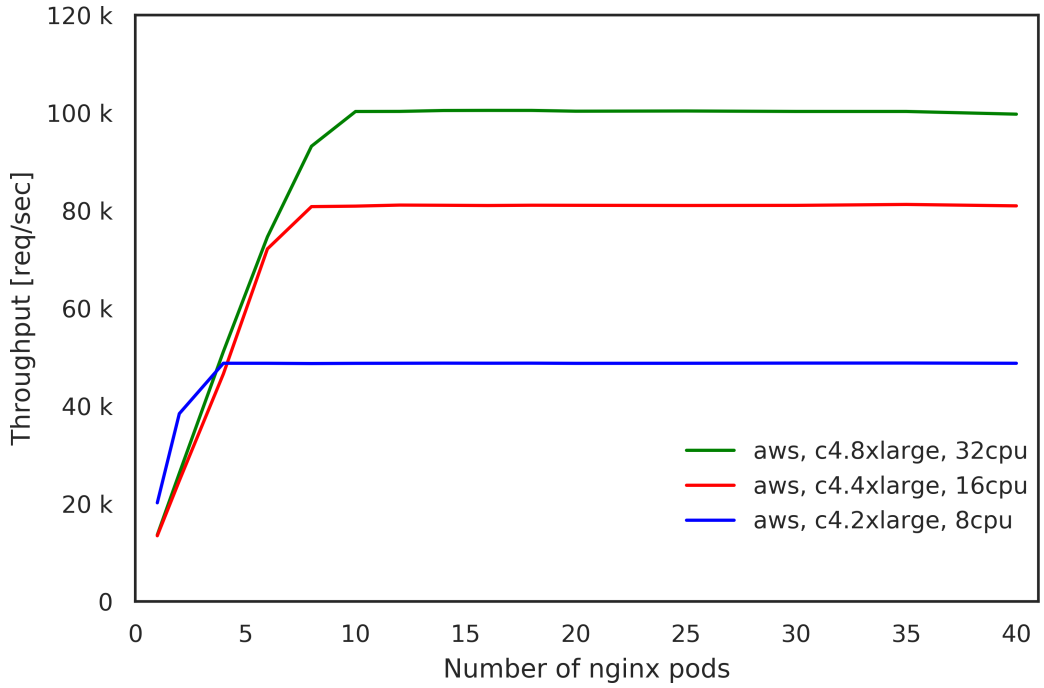


Figure 5.10: AWS with Node x 6, Client x 1, Load balancer x 1. Custom instance.

5.4 Summary

In this chapter portability and performance level of proposed load balancer in 1 Gbps network environments has been discussed. The throughput levels of a load balancer are dependent on settings for multicore packet processing. It is clear that the case that utilizes all of the CPU cores better performs than the case with only four CPU cores utilized. It is better to use as many CPU cores as possible for packet processing. The throughput levels are also very dependent on the back end mode of the flannel overlay network. The host-gw mode where no tunneling is used resulted in the best performance level.

The performance levels of ipvs-nat, iptables DNAT and nginx have also been compared. The proposed ipvs-nat load balancer in the container had the same performance level as load balancing function of iptables DNAT. Furthermore, in the case of L3DSR setup, the performance level of ipvs-tun load balancer has about 1.5 times larger than that of ipvs-nat and iptables DNAT.

It is also shown that the proposed load balancer can be run in GCP and AWS. The behavior of the proposed load balancer in those cloud environments is the same as that in the on-premise data center. The author concludes that the proposed load balancer is portable and outperforms the existing iptables DNAT load balancers in 1 Gbps network environments.

Chapter 6

Evaluation of redundancy and scalability

This chapter discusses the redundancy and scalability of the proposed load balancers. The ECMP technique is expected to make the load balancers redundant and scalable since all the load balancer containers act as active. The whole system is resilient to a single failure of load balancer container. Also since multiple of load balancers can be utilized simultaneously, it is expected that the throughput of the total system is increased significantly. In order to evaluate these characteristics of the ECMP technique, the author examined if the ECMP routing table is updated correctly when multiple of the load balancer *pods* are started. After that, in order to explore the scalability, the author also measured the throughput of the cluster of load balancers. Finally, the author examined how quick those ECMP routing table updates are. The following sections explain the evaluation in detail.

6.1 Evaluation method

Figure 6.1 shows the schematic diagram of the experimental setup. And Table 6.1 summarizes hardware and software specifications for the experiments. Multiple *pods* are deployed on multiple nodes in the Kubernetes cluster. In each *pod*, an nginx web server pod that returns the IP address of the *pod* are running. There are multiple nodes for load balancers and on each of the nodes, single load balancer *pod* is deployed. Each load balancer *pod* consists of both an ipvs container and an exabgp container. The routing table of the benchmark client is updated by BGP protocol through a route reflector.

Using these hardware and software setups the following four types of evaluations have been carried out; 1) Evaluation of ECMP functionality. The author examined if ECMP routing table is correctly updated. 2) Evaluation of the scalability. The author evaluated how throughput is improved by running multiple ipvs pods simultaneously. 3) Evaluation of ECMP response. The author evaluated the delay between the time ipvs pods are started or stopped until the time ECMP routing table reflected the change.

The throughputs are measured using wrk in the same manner as in Chapter 5. Notable differences from the previous throughput experiment in Figure 5.1 are; There four nodes for load balancers instead of one. Also, the 10 Gbps NIC is used for the benchmark client since for scalability experiment, there are multiple of ipvs container load balancers that can fill up 1 Gbps bandwidth. Some of the software has also been updated to the most recent versions at the time of the experiment.

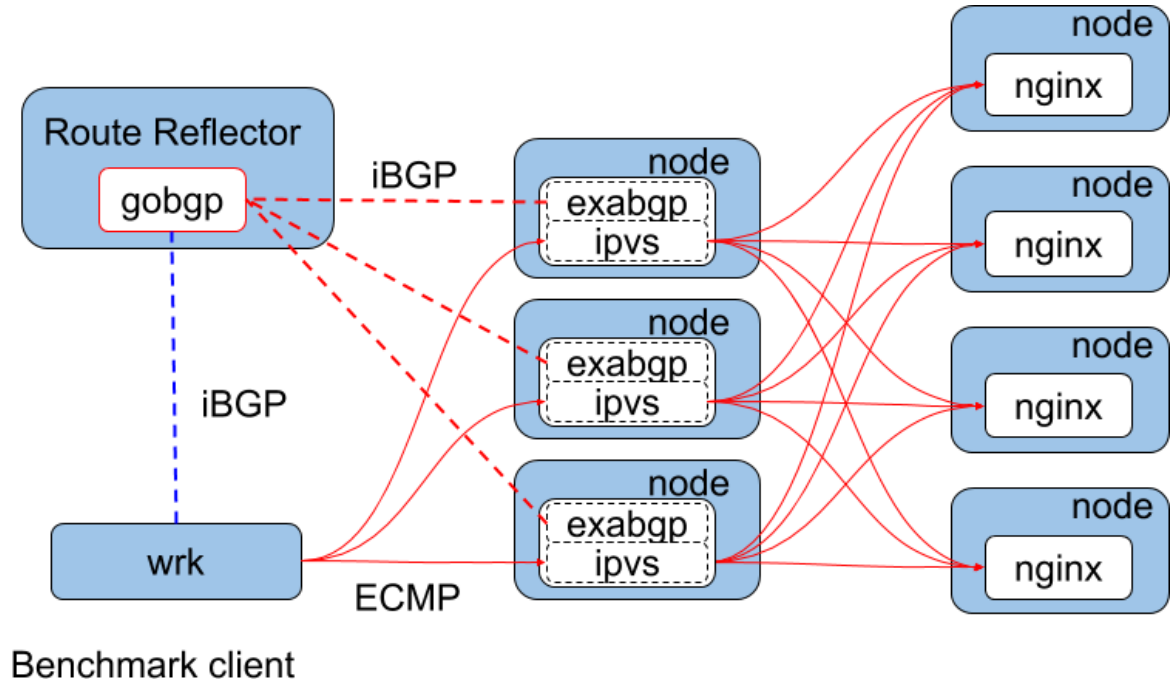


Figure 6.1: Experimental setups.

[Hardware Specification]
CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading)
Memory: 32GB
NIC: Broadcom BCM5720 Giga bit
(Node x 6, Load Balancer x 4)

CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading)
Memory: 32GB
NIC: Intel X550
(Client x 1)

[Node Software]
OS: Debian 9.5, linux-4.16.8
Kubernetes v1.5.2
flannel v0.7.0
etcd version: 3.0.15

[Container Software]
Keepalived: v1.3.2 (12/03,2016)
nginx : 1.15.4(web server)

Table 6.1: Hardware and software specifications.

6.2 ECMP functionality

10.1.1.0/24 via 10.0.0.106 dev eth0 proto zebra metric 20

(a) With single load balancer *pod*.

10.1.1.0/24 proto zebra metric 20
nexthop via 10.0.0.105 dev eth0 weight 1
nexthop via 10.0.0.106 dev eth0 weight 1
nexthop via 10.0.0.107 dev eth0 weight 1

(b) With three load balancer *pod*s.

10.1.1.0/24 pro to zebra metric 20
nexthop via 10.0.0.107 dev eth0 weight 1
nexthop via 10.0.0.105 dev eth0 weight 1
nexthop via 10.0.0.106 dev eth0 weight 1
10.1.2.0/24 proto zebra metric 20
nexthop via 10.0.0.107 dev eth0 weight 1
nexthop via 10.0.0.106 dev eth0 weight 1

(c) For a service with three load balancer *pod*s and a service with two load balancer *pod*s.

Table 6.2: ECMP routing tables.

First, the author examined ECMP functionality by monitoring the routing table on the benchmark client. Table 6.2 (a) shows the routing table entry on the router when a single load balancer pod existed. From this entry, we can tell that packets toward 10.1.1.0/24 are forwarded to 10.0.0.106 where the load balancer pod is running. There is also a keyword, zebra, which indicates that zebra controls this routing rule.

When the number of the load balancer pods was increased to three, the routing table becomes to have entries in Table 6.2 (b). There are three next hops towards 10.1.1.0/24 each of which being the node where the load balancer pods are running. The weights of the three next-hops are equal, i.e., 1. The update of the routing entry was almost instant as the author increased the number of the load balancers.

Table 6.2 (c) shows the case where the author additionally started new service with two load balancer pods with service addresses in 10.1.2.0/24 range. It was possible to accommodate two different services with different IP addresses, one with three load balancers and the other with two load balancers on a group of nodes, 10.0.0.105, 10.0.0.106 and 10.0.0.107. The update of the routing entry was almost instant as the author started the load balancers for the second service.

6.3 Scalability

The throughput measurement was also carried out to show that ECMP technique increases the throughput as the number of the load balancers is increased. Figure 6.2 shows the results of the measurements. There are four solid lines in the figure, each corresponding the throughput result when there are one through four of the proposed load balancers.

As can be seen in the figure, as we increased the number of the pod the throughput increased linearly to a certain level after which it saturated. The saturated levels, i.e. performance levels, depend on the number of the ipvs load balancer pods (lb x 1 being the case with one ipvs pods, and lb x2 being two of them and as such). The performance levels increase linearly as we increase the number of the load balancers. The performance level did not scale further when the number of load balancers was increased more than four. This was because the performance of the benchmark client was hitting the ceiling, i.e., the CPU usage was 100% when the total

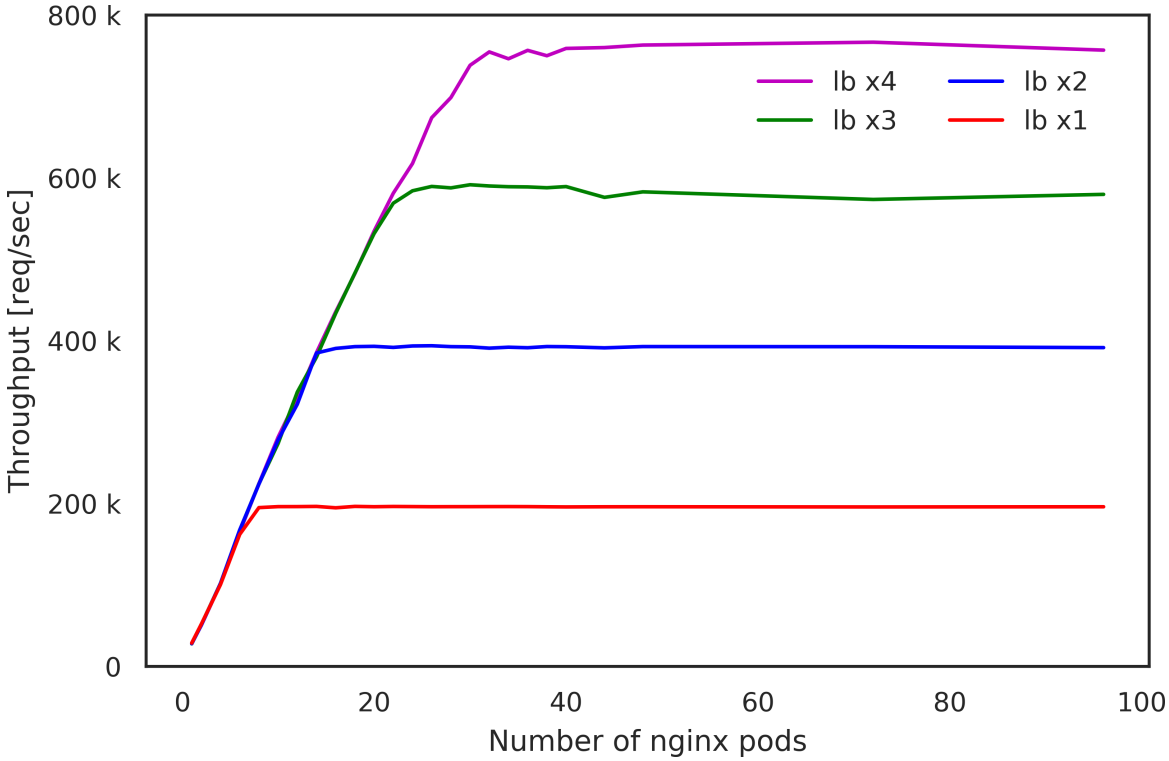


Figure 6.2: Throughput of ECMP redundant load balancer.
The throughputs are measured for a single load balancer(lb x1), two(lb x2), three(lb x3) and four(lb x4) load balancers.

throughput was around 780k [req/sec]. The author expects that replacing the benchmark client with more powerful machines will further improve the performance level this system.

6.4 ECMP response

Figure 6.3 shows the histogram of the ECMP update delay. The author measured the delays until the number of running ipvs pods is reflected into the routing table on the benchmark client. The number of the ipvs pods is changed randomly every 60 seconds for 20 hours. As we can see from the figure, most of the delays are within 6 seconds, and the largest delay was 10 seconds. We can say that ECMP routing update in our proposed architecture is quick enough.

Figure 6.4 shows the throughput measurement results when the number of the load balancers was periodically changed. The red line in the figure shows the number of the ipvs load balancer pods, which was changed randomly in every 60 seconds. The blue line corresponds to the resulting throughput. As we can see from the figure, the blue line nicely follows the shape of the red line. This indicates that new load balancers are immediately utilized after they are created. It also indicates that after removing some load balancers, the traffic to them is immediately directed to the existing load balancers.

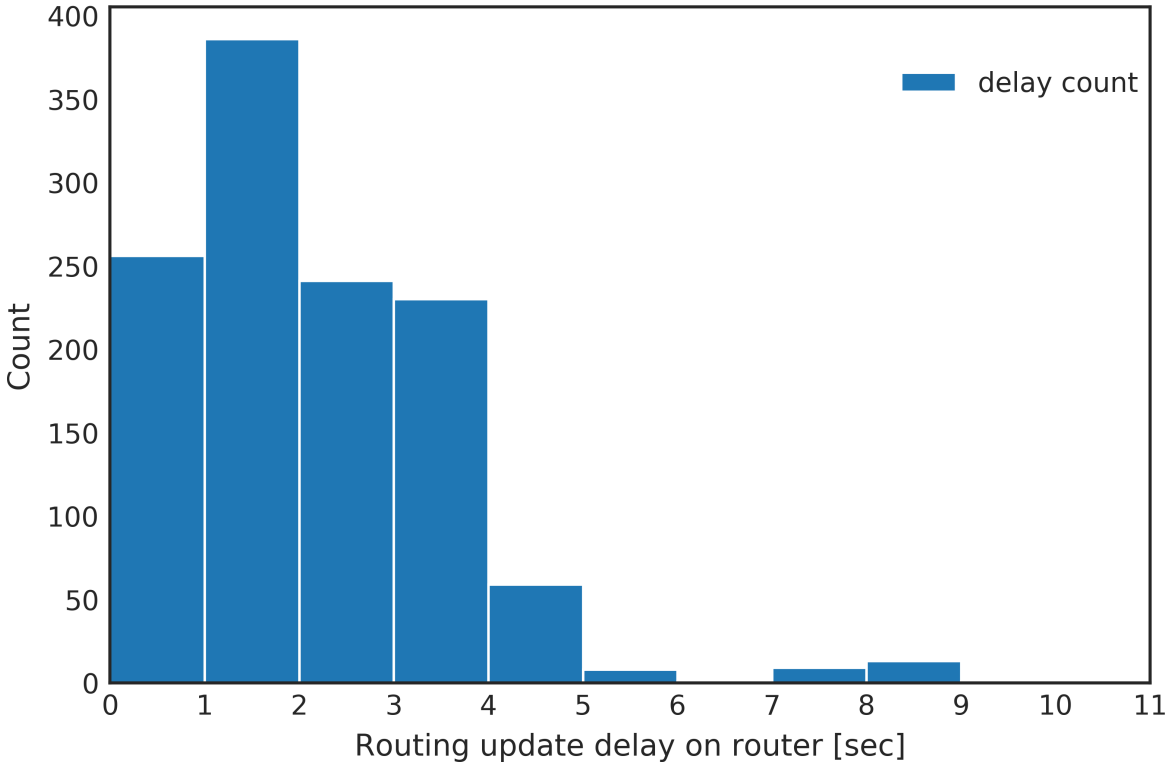


Figure 6.3: Caption 2

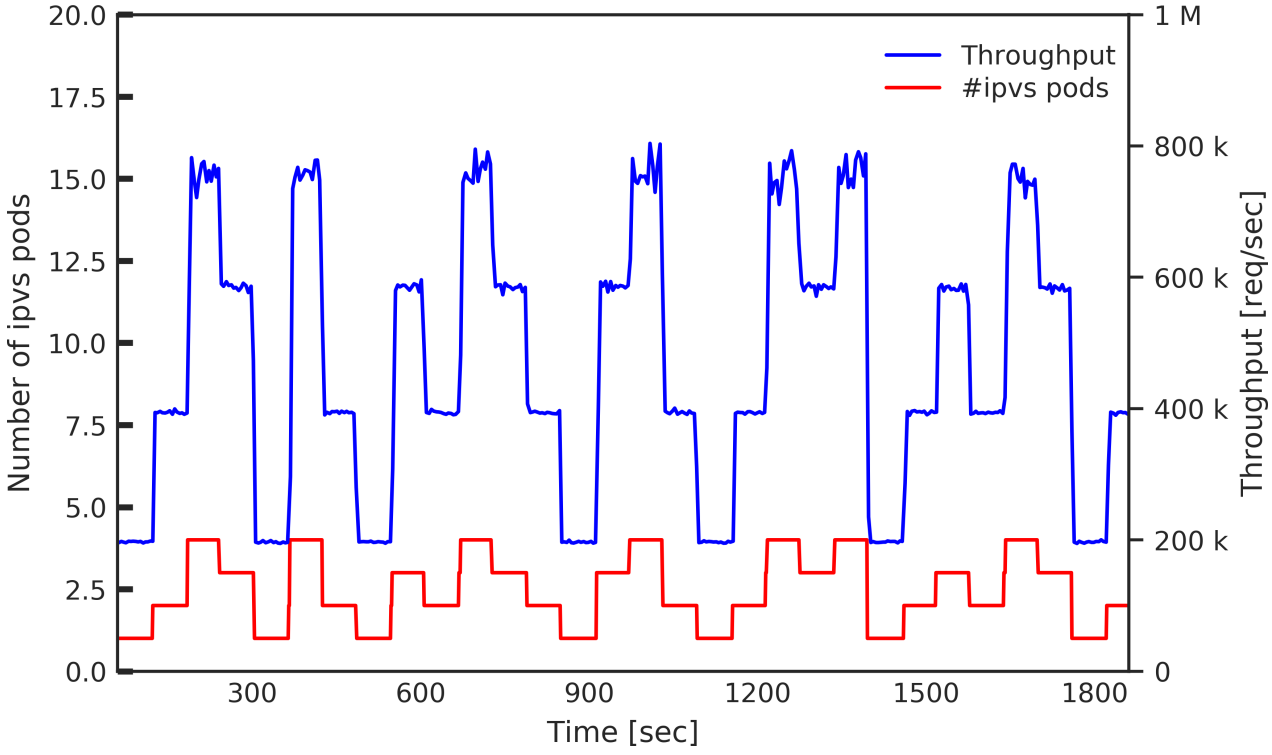


Figure 6.4: Caption 2

6.5 Summary

In this chapter, the redundancy and scalability of the proposed load balancers have been discussed. The author verified that ECMP routing table was properly created in the experimental system. The update of the ECMP routing table was quick enough, i.e., within 10 seconds, throughout 20 hours experiment and the routing table was always correct. The scalability of the load balancer was also examined and it has been found that maximum performance levels scaled linearly as the number of the load balancer pods was increased to four. The maximum throughput level obtained through the experiment was 780k [req/sec], which is limited due to the maximum CPU performance of the benchmark client rather than the performance of the load balancer cluster.

Chapter 7

Load balancer performance in 10Gbps environmets

Up until this chapter most of the experiments are done in 1Gbps network environments. The proposed load balancers have shown decent performance levels in 1Gbps environment. However, it is essential to investigate the feasibility of the proposed load balancers in 10Gbps network environments. In this chapter, the author carries out throughput measurements of ipvs-nat, ipvs-tun, and iptables DNAT in 10Gbps environment. Then the author improves the performance levels of ipvs-nat and ipvs-tun by setting up these load balancers in the node net namespace. Also presented is the novel software load balancer using eXpress Data Plane(XDP) technology, as an alternative to ipvs software load balancers.

7.1 Throuput of ipvs-nat, ipvs-tun and iptables DNAT

Figure 7.1 shows the packte flow for ipvs-nat and iptables DNAT, and Figure 7.2 shows that for ipvs-tun. The 10Gbps NICs are used for benchmark client and the node for the load balancer. In the case of the ipvs-nat and iptables DNAT, the response packets from nginx pods are returned to the load balancer, and then load balancer returns it to the client. In contrast, in the case of ipvs-tun, the response packets from nginx pods are directly returned to the client. Since the load balancer does not have to process the response packet, the better performance level is expected for ipvs-tun.

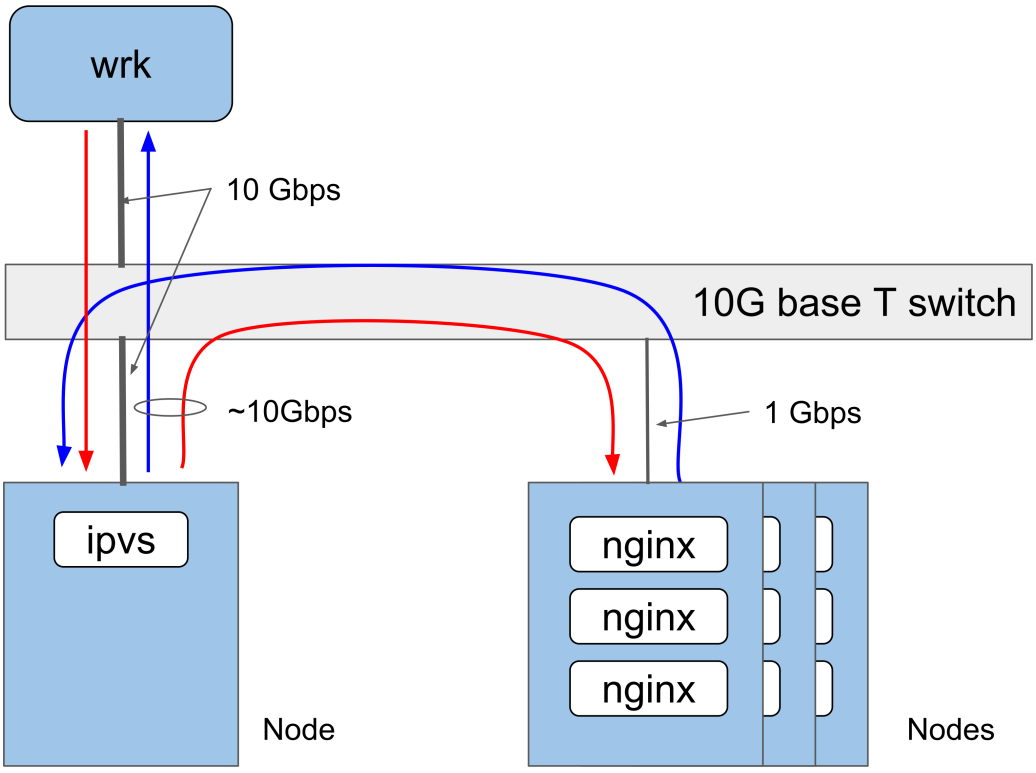


Figure 7.1: Packet flow of ipvs-nat and iptables DNAT.

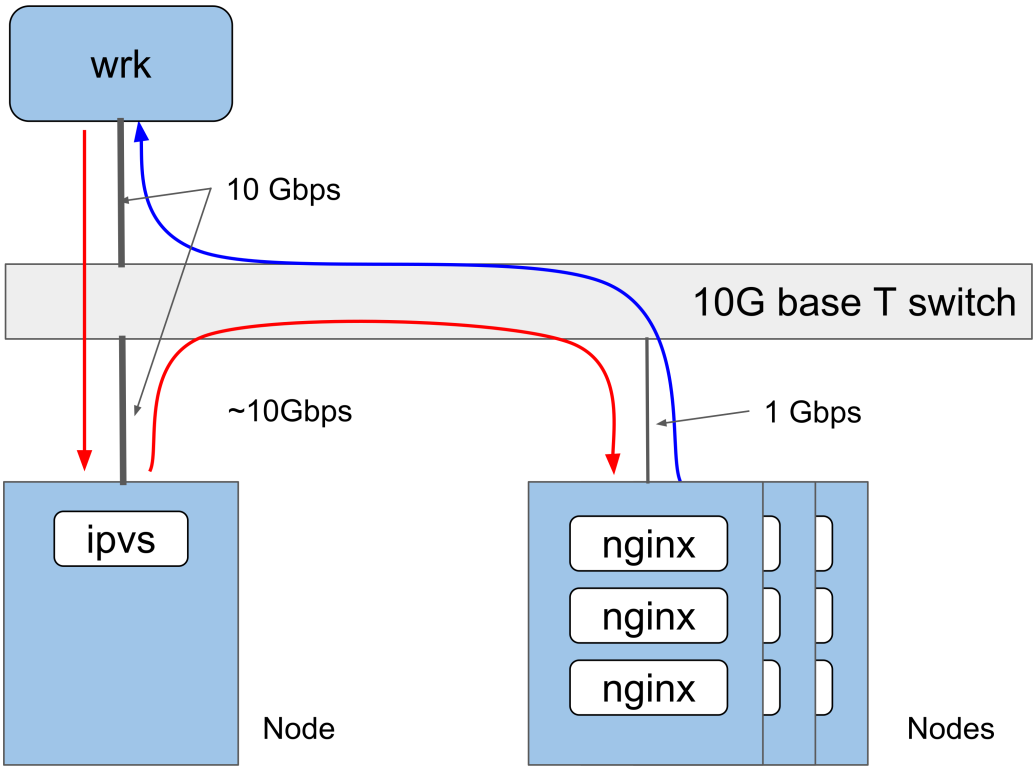


Figure 7.2: Packet flow of ipvs-tun.

Figure 7.3 shows the throughput of ipvs-tun, ipvs-nat and iptables DNAT in 10Gbps environment. We can see the general characteristics of a load balancer where the throughput increases linearly to a certain level as the number of nginx container increases, and then eventually saturates. These saturation levels are the performance limits of each of the load balancers, which is determined by packet forwarding efficiency or the bandwidth of the network. The performance limit of the iptables DNAT is close to 780k [req/sec], where the CPU usage of the benchmark client becomes 100%.

Table 7.1 summarizes the throughput of ipvs-tun, ipvs-nat and iptables DNAT at 40 nginx pods in 10 Gbps and 1 Gbps networks. By using 10 Gbps network, the performance levels for all of these load balancer are improved. However, the magnitudes of the improvements are different among the types of the load balancers. While the throughput of the ipvs-nat is 334833 [req/sec], that of the iptables DNAT is 777640 [req/sec]. This suggests that the packet forwarding of the iptables DNAT is more efficient than that of ipvs-nat. Although the throughput of the ipvs-tun, 730975 [req/sec] is better than ipvs-nat because of the L3DSR settings, it still falls short of that of iptables DNAT. It seems that containerized ipvs load balancers are inherently less efficient than the iptables DNAT, which could be attributed to either overhead of container network(veth+bridge) or kernel code for ipvs itself. In order to investigate this issue, the author conducted a throughput measurement for ipvs-nat and ipvs-tun that are set up in node net namespaces in the next section.

	Throughput [req/sec]	
Type of load balancer	1Gbps	10Gbps
iptables DNAT	193198	777640
ipvs-nat	195666	334833
ipvs-tun	292660	730975

Table 7.1: Performance levels in 1Gbps and in 10Gbps.
Throughput results of the load balancers at 40 nginx pods from the data for the Figures 5.8 and 7.3 are shown.

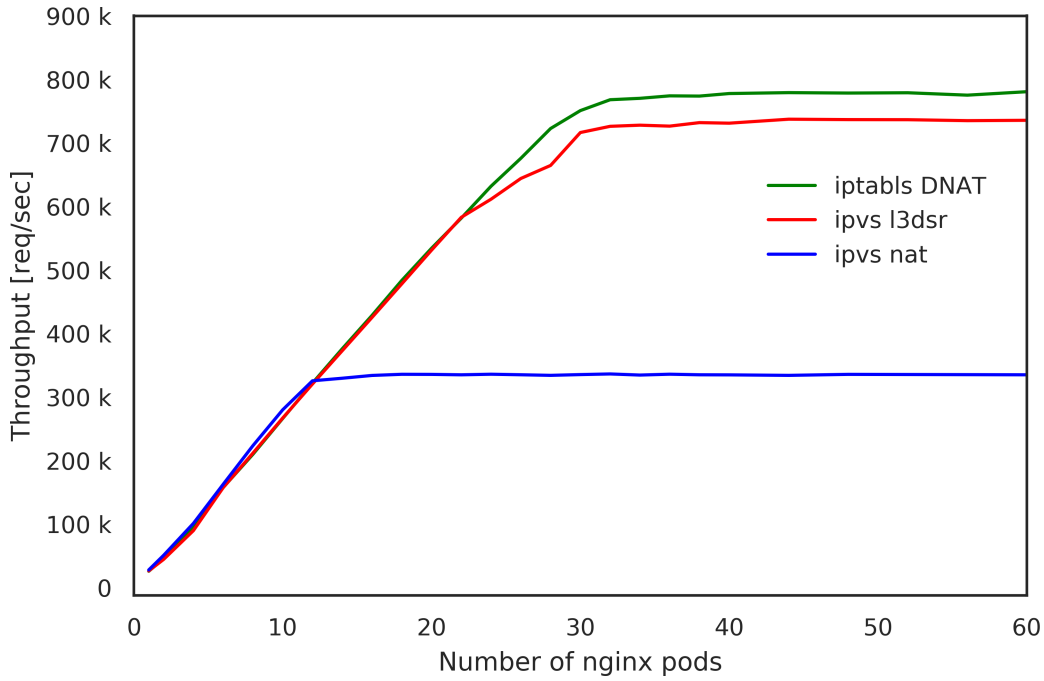


Figure 7.3: Throughput of load balancers in 10 Gbps.

7.2 Throuput of ipvs-nat, ipvs-tun and iptables DNAT

In order to improve the throughput of the ipvs load balancers by removing the overhead of container network, the ipvs load balancers were set up in node net namespaces. Appendix XXX shows inside of ipvs container script that launches the keepalived in node net namespaces. By doing so, the load balancing tables are created in the node net namespace.

Figure 7.4 shows the throughput of ipvs-nat and ipvs-tun in the node namespace together with the throughput of the iptables DNAT. The throughput of the ipvs-tun is almost identical to that of iptables DNAT, which is limited by CPU power of the benchmark client. Although the throughput of the ipvs-nat is smaller than that of the iptables DNAT, it is clearly improved from the result in Figure 7.3.

Table 7.2 compares the throughput of ipvs load balancers in the pod namespace and in the node namespace at 40 nginx pods. The thoughput data are taken from the results in the Figures 7.3 and 7.4. We can see that maximum throughputs can be improved in the case of load balancers in node namespace both for the ipvs-nat and the ipvs-tun.

	Throughput [req/sec]	
Type of load balancer	pod name space	node name space
iptables DNAT	NA	777640
ipvs-nat	334833	699635
ipvs-tun	730975	779932

Table 7.2: Performance levels in pod namespace and in node namespace.
Throughput results of the load balancers at 40 nginx pods from the data for the Figures 7.3 and 7.4 are shown.

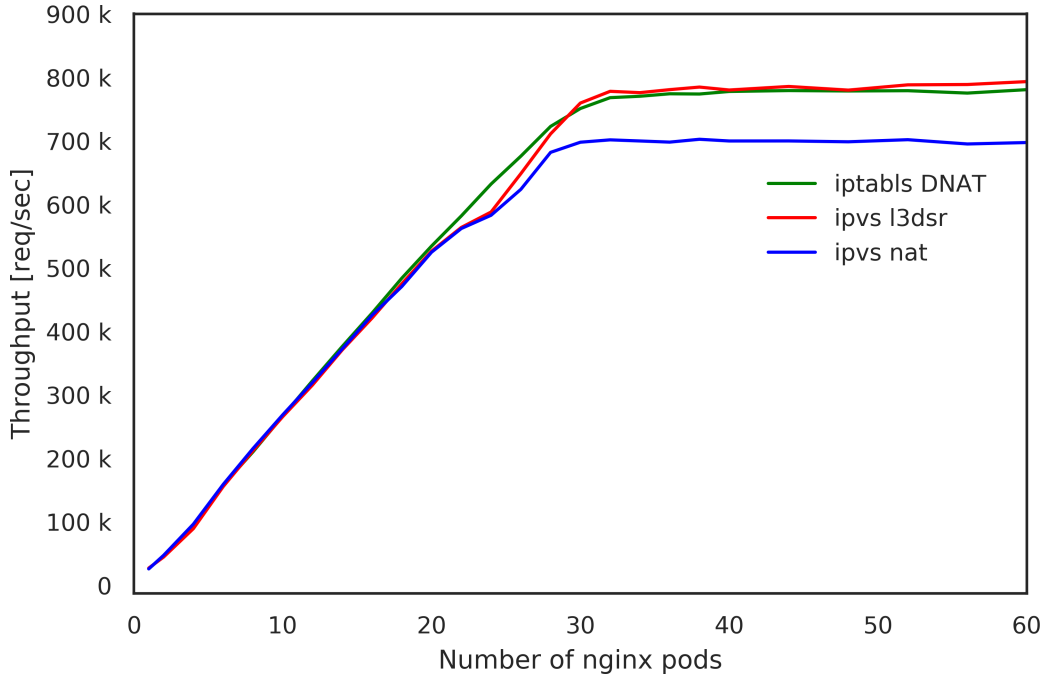


Figure 7.4: Throughput of load balancers in node name space.

7.3 XDP load balancer

7.4 Summary

In this chapter, the author carried out throughput measurements of ipvs-nat, ipvs-tun, and iptables DNAT in 10Gbps environment. From the results the general characteristics of a load balancer are observesd. The throughput increases linearly to a certain level as the number of nginx container increases, and then eventually saturates. The performance levels for of the load balancers are improved by using 10Gbps network. However, the throughputs of ipvs-nat and ipvs-tun are smaller than that of iptables DNAT.

Then the author improved the performance levels by setting up ipvs-nat and ipvs-tun load balancers in the node net namespace to remove overhead of the container network. The throughput of the ipvs-tun became almost identical to that of iptables DNAT, and the throughput of the ipvs-nat also improved to the level close to that of the iptables DNAT.

The authot also presented the novel software load balancer using eXpress Data Plane(XDP) technology, as an alternative to ipvs software load balancers. And brah brah brah.....

Chapter 8

Limitations and future work

The current limitations of this study are; 1) Although our proposed architecture is feasible where users can set up iBGP peer connections to upstream routers, currently major cloud providers do not seem to provide such services. 2) ... These should be addressed in the future work. For other future work we plan to improve performance of a single software load balancer on standard Linux box using Xpress Data Plane(XDP) technology.

Chapter 9

Conclusion

9.1 Conclusions

In this dissertation, the author proposed a portable load balancer with ECMP redundancy for the Kubernetes cluster systems that is aimed at facilitating migration of container clusters for web services.

The author 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 appropriate 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 throughput levels of a load balancer are dependent on settings for multicore packet processing. It has been better to use as many CPU cores as possible for packet processing. The throughput levels are also very dependent on the back end mode of the flannel overlay network. The host-gw mode where no tunneling is used resulted in the best performance level. It is clear that the case that utilizes all of the CPU cores better performs than the case with only four CPU cores utilized. The performance levels of ipvs(nat), iptables DNAT and nginx have also been compared. The proposed ipvs(nat) load balancer in the container had the same performance level as load balancing function of iptables DNAT. Furthermore in the case of L3DSR setup, the performance level of ipvs(tun) load balancer has about 1.5 times larger that that of ipvs(nat) and iptables DNAT. It is also shown that the proposed load balancer can be run in GCP and AWS. The behavior of the proposed load balancer in those cloud environments is the same as that in the on-premise data center. The author conclude that the proposed load balancer is portable and performs at the same level as the existing load balancers.

The ECMP technique is expected to make the load balancers redundant and scalable since all the load balancer containers act as active. The whole system is resilient to a single failure of load balancer container. Also since multiple of load balancers can be utilized simultaneously, it is expected that the throughput of the total system is increased significantly. In order to evaluate these characteristics of the ECMP technique, the author examined if the ECMP routing table is updated correctly when multiple of the load balancer *pods* are started. After that, in order to explore the scalability, the author also measured the throughput of the cluster of load balancers. Finally, the author examined how quick those ECMP routing table updates are. The author verified that ECMP routing table was properly created in the experimental system. The update of the ECMP routing table was correct and quick enough, i.e., within 10 seconds, throughout 20 hours experiment. The scalability of the load balancer was also examined and it has been found that maximum performance levels scaled linearly as the number of the load balancer pods was increased to four. The maximum throughput level obtained through the experiment was 780k [req/sec], which is limited due to the maximum CPU performance

of the benchmark client rather than the performance of the load balancer cluster.

The limitations of this work that authors aware of include the followings: 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.

Bibliography

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

[2] The Kubernetes Authors. Kubernetes | production-grade container orchestration, 2017.

[3] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI*, pages 523–535, 2016.

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

[5] Martin A. Brown. Guide to IP Layer Network Administration with Linux, 2007.

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

[7] Wensong Zhang. Linux virtual server for scalable network services. *Ottawa Linux Symposium*, 2000.

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

[9] Gilberto Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, 2017.

[10] The Kubernetes Authors. Ingress resources | kubernetes, 2017.

[11] The Kubernetes Authors. Federation, 2017.

[12] Michael Pleshakov. Nginx and nginx plus ingress controllers for kubernetes load balancing, December 2016.

[13] NGINX Inc. Nginx ingress controller, 2017.

[14] Bowei Du Prashanth B, Mike Danese. kube-keepalived-vip, 2016.

[15] Alexandre Cassen. Keepalived for linux.

[16] Docker Core Engineering. Docker 1.12: Now with built-in orchestration! - docker blog, 2016.

[17] Docker Inc. Use swarm mode routing mesh | Docker Documentation, 2017.

[18] Andrey Sibiryov. Gorb go routing and balancing, 2015.

[19] Tero Marttila. Design and implementation of the clusterf load balancer for docker clusters. Master's thesis, aalto university, 2016-10-27.

[20] Inc CoreOS. etcd | etcd Cluster by CoreOS.

[21] HashiCorp. Consul by HashiCorp.

[22] Inc CoreOS. Backend.

[23] Tom Herbert and Willem de Bruijn. Scaling in the Linux Networking Stack.

[24] ktaka ccmp. ktaka-ccmp/ipvs-ingress: Initial release, July 2017.

[25] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, and Jasper Spaans. Linux Advanced Routing & Traffic Control HOWTO, 2002.

[26] Will Glozer. wrk - a http benchmarking tool, 2012.

[27] Van Jacobson, Craig Leres, and S McCanne. The tcpdump manual page. *Lawrence Berkeley Laboratory, Berkeley, CA*, 143, 1989.

Appendix A

ingress controller

```
package main

import (
        "log"
        "net/http"
        "os"
        "syscall"
        "os/exec"
        "strings"
        "text/template"
        "github.com/spf13/pflag"
        api "k8s.io/client-go/pkg/api/v1"
        nginxconfig "k8s.io/ingress/controllers/nginx/pkg/config"
        "k8s.io/ingress/core/pkg/ingress"
        "k8s.io/ingress/core/pkg/ingress/controller"
        "k8s.io/ingress/core/pkg/ingress/defaults"
)

var cmd = exec.Command("keepalived", "-nCDlf", "/etc/keepalived/ipvs.conf")

func main() {
        ipvs := newIPVSController()
        ic := controller.NewIngressController(ipvs)
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
        cmd.Start()
        defer func() {
                log.Printf("Shutting down ingress controller...")
                ic.Stop()
        }()
        ic.Start()
}

func newIPVSController() ingress.Controller {
        return &IPVSController{}
}

type IPVSController struct{}

func (ipvs IPVSController) SetConfig(cfgMap *api.ConfigMap) {
        log.Printf("Config map %+v", cfgMap)
}

func (ipvs IPVSController) Reload(data []byte) ([]byte, bool, error) {
        cmd.Process.Signal(syscall.SIGHUP)
        out, err := exec.Command("echo", string(data)).CombinedOutput()
```

```
        if err != nil {
                return out, false, err
        }
        log.Printf("Issue kill to keepalived. Reloaded new config %s", out)
        return out, true, err
}

func (ipvs IPVSController) OnUpdate(updatePayload ingress.Configuration) ([]byte, error)
    ↪ {
        log.Printf("Received OnUpdate notification")
        for _, b := range updatePayload.Backends {
                type ep struct{
                        Address,Port string
                }
                eps := []ep{}
                for _, e := range b.Endpoints {
                        eps = append(eps, ep{Address: e.Address, Port: e.Port})
                }

                for _, a := range eps {
                log.Printf("Endpoint %v:%v added to %v:%v.", a.Address, a.Port, b.Name, b
                    ↪ .Port)
                }

                if b.Name == "upstream-default-backend" {
                        continue
                }
                cnf := []string{"/etc/keepalived/ipvs.d/" , b.Name , ".conf"}
                w, err := os.Create(strings.Join(cnf, ""))
                if err != nil {
                        return []byte("Ooops"), err
                }
                tpl := template.Must(template.ParseFiles("ipvs.conf.tmpl"))
                tpl.Execute(w, eps)
                w.Close()
        }

        return []byte("hello"), nil
}

func (ipvs IPVSController) BackendDefaults() defaults.Backend {
        // Just adopt nginx's default backend config
        return nginxconfig.NewDefault().Backend
}

func (ipvs IPVSController) Name() string {
        return "IPVS Controller"
}

func (ipvs IPVSController) Check(_ *http.Request) error {
        return nil
}

func (ipvs IPVSController) Info() *ingress.BackendInfo {
```

```
        return &ingress.BackendInfo{
                Name:       "dummy",
                Release:    "0.0.0",
                Build:      "git-00000000",
                Repository: "git://foo.bar.com",
        }
}

func (ipvs IPVSController) OverrideFlags(*pflag.FlagSet) {
}

func (ipvs IPVSController) SetListers(lister ingress.StoreLister) {
}

func (ipvs IPVSController) DefaultIngressClass() string {
        return "ipvs"
}
```

Appendix B

ECMP settings

B.1 Exabgp configuration on the load balancer container.

exabgp.conf:

```
neighbor 10.0.0.109 {
 description "peer1";
 router-id 172.16.20.2;
 local-address 172.16.20.2;
 local-as 65021;
 peer-as 65021;
 hold-time 1800;
   static {
     route 10.1.1.0/24 next-hop 10.0.0.106;
   }
}
```

B.2 Gobgpd configuration on the route reflector.

gobgp.conf:

```
global:
  config:
    as: 65021
    router-id: 10.0.0.109
    local-address-list:
    - 0.0.0.0 # ipv4 only
  use-multiple-paths:
    config:
      enabled: true

peer-groups:
  - config:
      peer-group-name: k8s
      peer-as: 65021
    afi-safis:
      - config:
          afi-safi-name: ipv4-unicast

dynamic-neighbors:
```

```
  - config:
      prefix: 172.16.0.0/16
      peer-group: k8s

neighbors:
  - config:
      neighbor-address: 10.0.0.110
      peer-as: 65021
    route-reflector:
      config:
        route-reflector-client: true
        route-reflector-cluster-id: 10.0.0.109
    add-paths:
      config:
        send-max: 255
        receive: true
```

B.3 Gobgpd and zebra configurations on the router.

gobgp.conf:

```
global:
  config:
    as: 65021
    router-id: 10.0.0.110
    local-address-list:
    - 0.0.0.0

  use-multiple-paths:
    config:
      enabled: true

neighbors:
    - config:
        neighbor-address: 10.0.0.109
        peer-as: 65021
      add-paths:
        config:
          receive: true

zebra:
  config:
    enabled: true
    url: unix:/run/quagga/zserv.api
```

```
    version: 3
    redistribute-route-type-list:
      - static
```

zebra.conf:

```
hostname Router
log file /var/log/zebra.log
```

Appendix C

Analysis of the performance limit

The maximum throghput in this series of experiment is roughly, 190k[req/sec] for both ipvs an the iptables DNAT. At first, it was not clear what caused this limit. The author analyzed the kind of packets that flows during the experiment using tcpdump[27] as follows; 1) A wrk worker opens multiple connections and sends out http request to the web servers. The number of connections is determined by the command-line option, eg. 800/40 = 20 connection in the case of command-line in Table 5.1. The worker sends out 100 requests to the web server within each connection, and closes it either if all of the responses are recieved or time out occurs. 2) As in seen in Listing C.1, tcp options were mss(4 byte), sack(2 byte), ts(10 byte), nop(1 byte) and wscale(3 byte), for SYN packets. For other packets, tcp options were, nop(1 byte), nop(1 byte) and ts(10 byte). 3) The author classified the types of packes and counted the number of each type in a single connection, which is 100 http requests. Table C.1,C.2,C.3 summarize the data size of 100 request, including TCP headr, IP header, Ether header and overheads. From this analysis, it was found that per each HTTP request and response, request data with the size of 227.68[byte] and response data with the data(http content)+437.68[byte] were being sent.

Since the node for load balancer recives and transmits both request and response packets using single network interface, each 1Gbps half duplex of full duplex must accomodate request and response data size. Therefore the theoretical maximum throughput can be expressed as;
throughput[req/sec] = band width[byte/sec]/(request + response)
= 1e9/8/(data+665.36)

Figure 5.3 shows plot of theoretical maximum throughput 1Gbps ethernet together with actual benchmark results. Since experimnetal results agrees well with theory, the author concludes that when “RPS = on”, ipvs performance limit is due to the 1Gbps bandwidth.

```
1  curl -s  http://172.16.72.2:8888/1000
2  tcpdmup(response):
3
4  03:09:27.968942 IP 172.16.72.2.8888 > 192.168.0.112.60142:
5  Flags [S.], seq 2317920646, ack 648140715, win 28960, options [mss 1460,sackOK,TS val
       2274012282 ecr 2324675546,nop,wscale 8], length 0
6  03:09:27.969685 IP 172.16.72.2.8888 > 192.168.0.112.60142:
7  Flags [.], ack 85, win 114, options [nop,nop,TS val 2274012282 ecr 2324675546], length
       0
8  03:09:27.969945 IP 172.16.72.2.8888 > 192.168.0.112.60142:
9  Flags [P.], seq 1:255, ack 85, win 114, options [nop,nop,TS val 2274012282 ecr
       2324675546], length 254
10 03:09:27.969948 IP 172.16.72.2.8888 > 192.168.0.112.60142:
11 Flags [P.], seq 255:1255, ack 85, win 114, options [nop,nop,TS val 2274012282 ecr
       2324675546], length 1000
12 03:09:27.970846 IP 172.16.72.2.8888 > 192.168.0.112.60142:
13 Flags [F.], seq 1255, ack 86, win 114, options [nop,nop,TS val 2274012282 ecr
       2324675547], length 0
```

Listing C.1: An example of the tcpdump output

Type of Packet	Payload [byte]	Header [byte]	Count	Total [byte]
SYN	0	98	1	98
ACK	0	90	102	9,180
Push(GET)	44	90	100	13,400
FIN+ACK	0	90	1	90
Total				22,768

Table C.1: Request data size for 100 HTTP requests in wrk measurement.

Type of Packet	Payload [byte]	Header [byte]	Count	Total [byte]
SYN+ACK	0	98	1	98
ACK	0	90	2	180
Push(GET)	254	90	100	34,400
Push(DATA)	data	90	100	100x(data+90)
FIN+ACK	0	90	1	90
Total			100x(data+90)+34,768	

Table C.2: Response data size for 100 HTTP requests in wrk measurement.

Type of field	SYN	ACK, SYN+ACK, FIN+ACK, PUSH
preamble	8	8
ether header	14	14
ip header	20	20
tcp header	20 + 20(tcp options)	20 + 12(tcp options)
fcs	4	4
inter frame gap	12	12
Total [byte]	98	90

Table C.3: Header sizes of TCP/IP packet in Ethernet frame.