A Study on Portable Load Balancer for Container Clusters

by

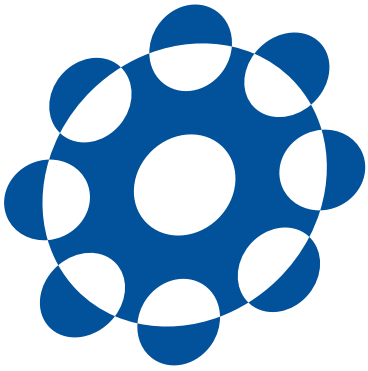
Kimitoshi Takahashi

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

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 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 infromation 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 send 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.

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 controlled 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. 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.

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 those environments without a load balancer that is supported by the Kubernetes, e.g., in an on-premise data center, 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. 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 8.2.

Chapter 2

Related Work

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.

Chapter 3

Background

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.

3.1 Overlay Network

3.1.1 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 3.1 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 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 3.1. 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.

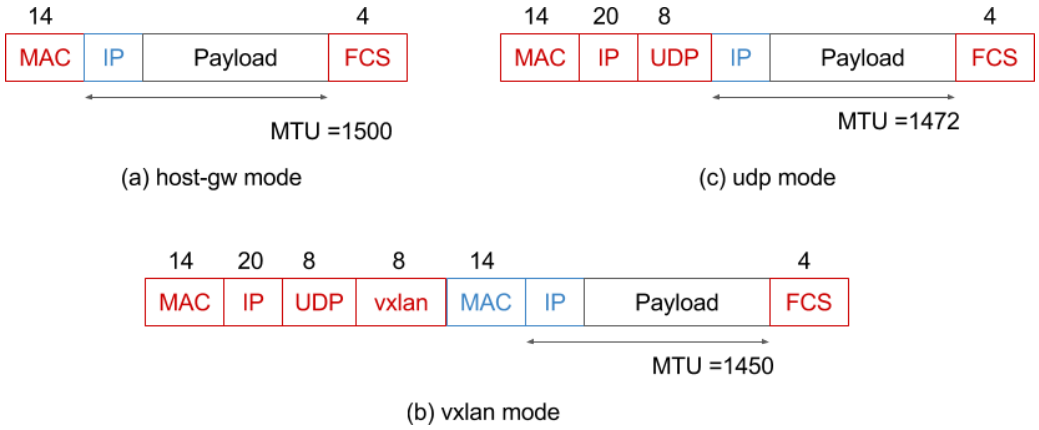


Figure 3.1: frame diagram

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

Table 3.1: Viable flannel backend modes. In cloud environment tunnelig using vxlan or udp is needed.

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

3.1.2 Calico

no-tunnel

ipip

3.2 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.

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

3.2.1 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 3.2 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
```

3.2.2 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.

3.2.3 rps

3.2.4 Others

rfs

xps

xfs

3.3 Other parameters

3.3.1 tcp congestion mode

3.4 Cloud Load Balancers

3.4.1 Maglev

3.4.2 Ananta

3.4.3 GCP Load Balancer

GCP experimental data.

3.5 Summary

Chapter 4

Load Balancer Architecture

This chapter provides discussion of load balancer suitable for container clusters. First we discuss problems of conventions architecture in Section 4.1. Then we discuss architectual choices and propose the best one in Section 4.2. After that we discuss the how to inplement a portable load balancer in Section 4.3. Finally we discuss the routig and redundancy architecture in Section 4.4.

4.1 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 4.1 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.

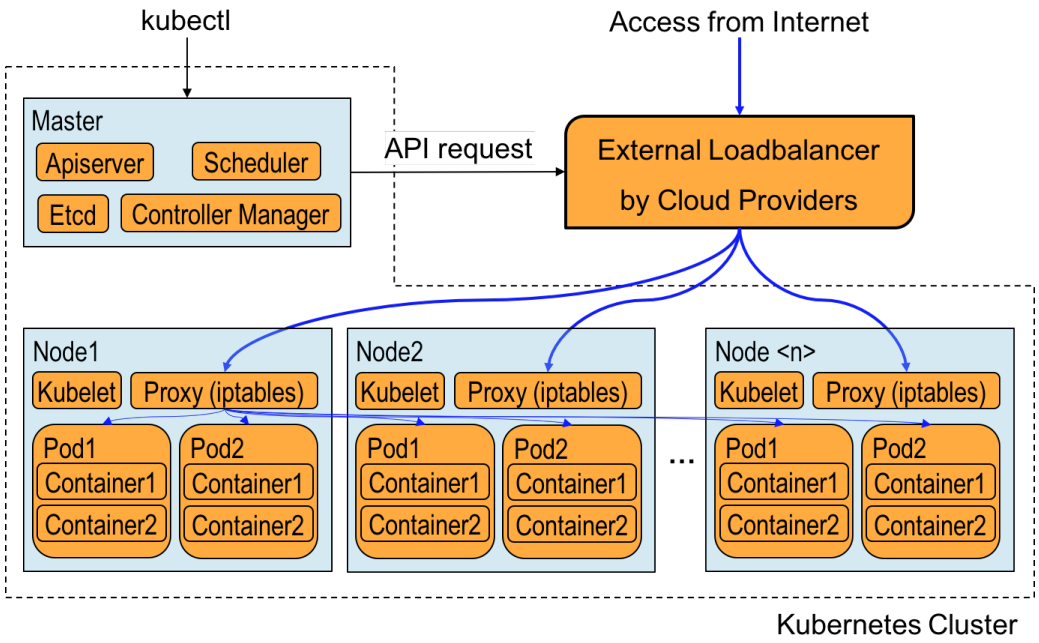


Figure 4.1: Conventional architecture of a Kubernetes cluster.

4.2 Proposed Architecture

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

The problems of Kubernetes architecture in Figure 4.2 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.

There are several 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 and make them supported by Kubernetes. d) Provide software load balancers as a part of container clusters.

The a) is impossible. The b) does not improve the usability of the container cluster since the specific hardware is always needed. The c) and d) are viable solutions since they can be realized using commodity hardware.

We do not know how many load balancers are needed for a specific web service. We also do not know how many of the services are deployed in a data center. Therefore a cluster of load balancers should be elastic, i.e., it should change the capacity on demand. Also, we also should be able to deploy many of load balancer cluster on demand. It would be better if we could share the same server pool with web servers, since capacity planning for hardware to host load balancer clusters may be difficult.

For such application needs the container cluster seems best suited. Therefore the author proposes the architecture, d) where a cluster of load balancers are also deployed as a cluster of containers.

Figure 4.2 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

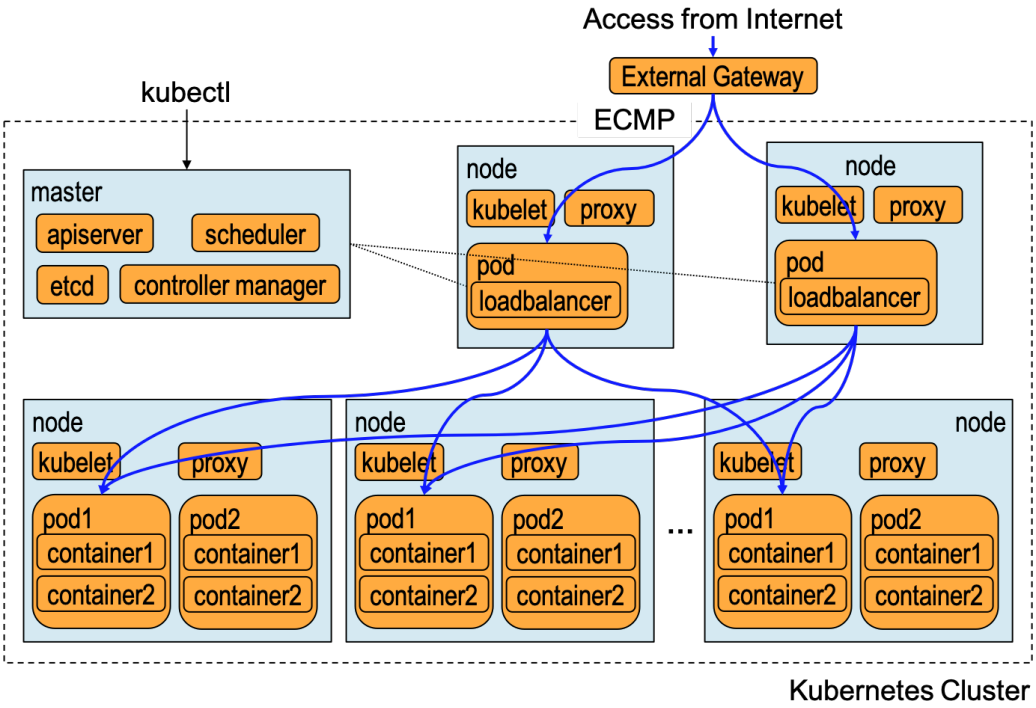


Figure 4.2: Kubernetes cluster with proposed load balancer.

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.

4.3 Portable Load Balancer

In order to implement a software load balancer that is runnable in any environment, ipvs is containerized. I 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.

4.4 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

[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.

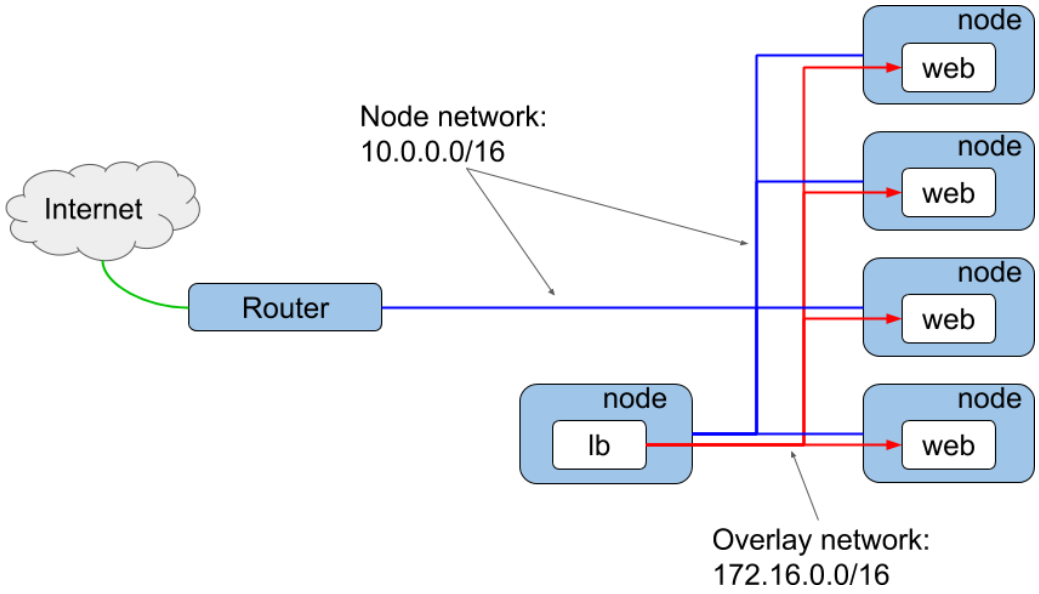


Figure 4.3: The network architecture of an exemplified container cluster system. A load balancer(lb) pod(the white box with "lb") and web pods are running on nodes(the blue boxes). The traffic from the internet are forwarded to the lb pod by the upstream router using the node network, and the distributed to web pods using the overlay network.

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.

4.4.1 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. 4.3 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. 4.3 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.

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

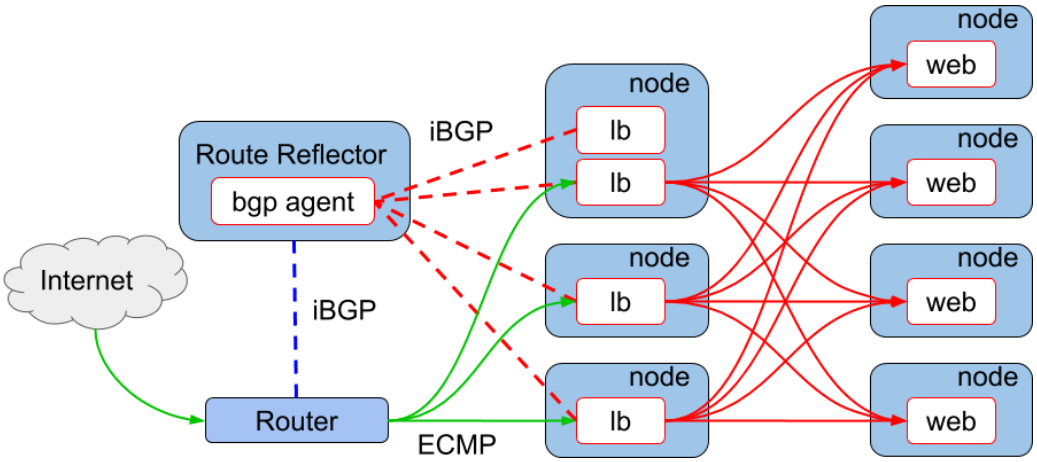


Figure 4.4: The proposed architecture of load balancer redundancy with ECMP. The traffic from the internet is distributed by the upstream router to multiple of lb pods using hash-based ECMP and then distributed by the lb pods to web pods using Linux kernel's ipvs. The ECMP routing table on the upstream router is populated using iBGP.

4.4.2 ECMP

Fig. 4.4 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.

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. 4.4,

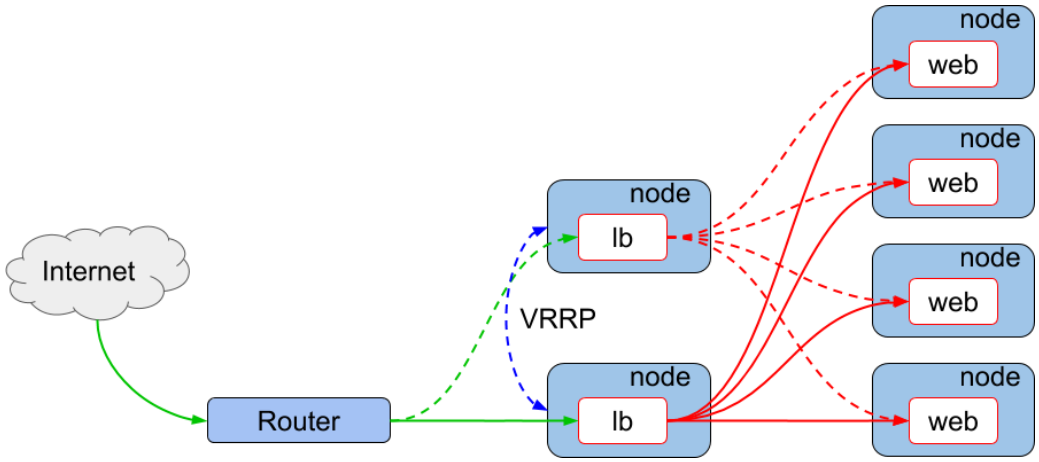


Figure 4.5: An alternative redundant load balancer architecture using VRRP.
The traffic from the internet is forwarded by the upstream router to a active lb node and then distributed by the lb pods to web pods using Linux kernel's ipvs. The active lb pod is selected using VRRP protocol.

we could also place another route reflector for redundancy purpose.

4.4.3 VRRP

Fig. 4.5 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.

4.4.4 Kubernetes

In the Cloud environment BGP peering services are not offered. In such cases, a Load balancer should update the routing table of cloud infrastructure.

4.5 Summary

Chapter 5

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 **??**.

5.1 Proof of concept system architecture

Fig. 5.1 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 .4.

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 .3.

5.2 Ipvs container

The proposed load balancer needs to dynamically reconfigure the IPVS balancing rules whenever *pods* are created/deleted. Figure 5.2 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

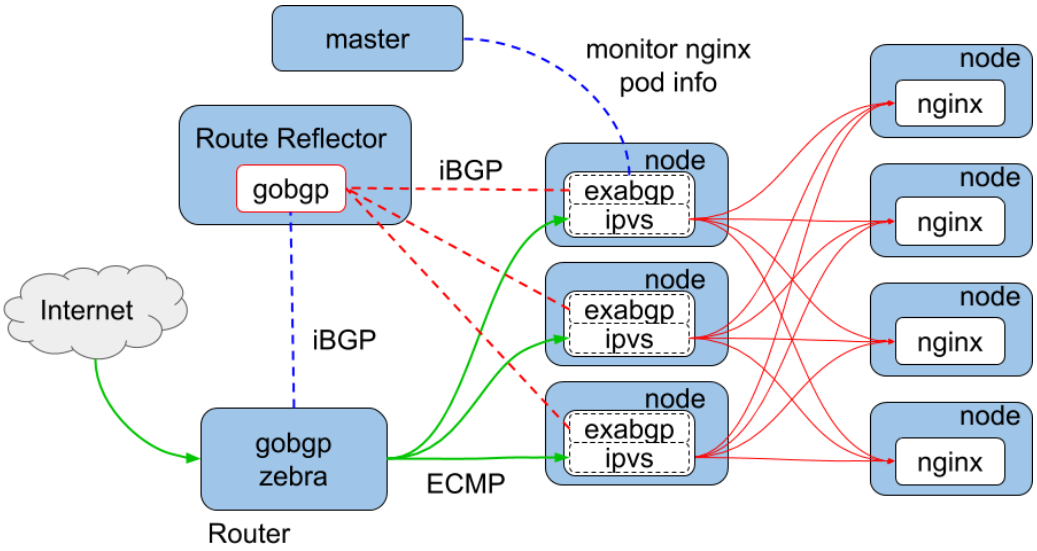


Figure 5.1: An experimental container cluster with proposed redundant software balancers.
The master and nodes are configured as Kubernetes's master and nodes on top of conventional Linux boxes, respectively. The route reflector and the upstream router are also conventional Linux boxes.

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 5.3 and Figure 5.4 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.

5.3 BGP software container

In order to implement the ECMP redundancy, we also containerized exabgp using Docker. Fig.5.5 (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.5.5 (??) 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

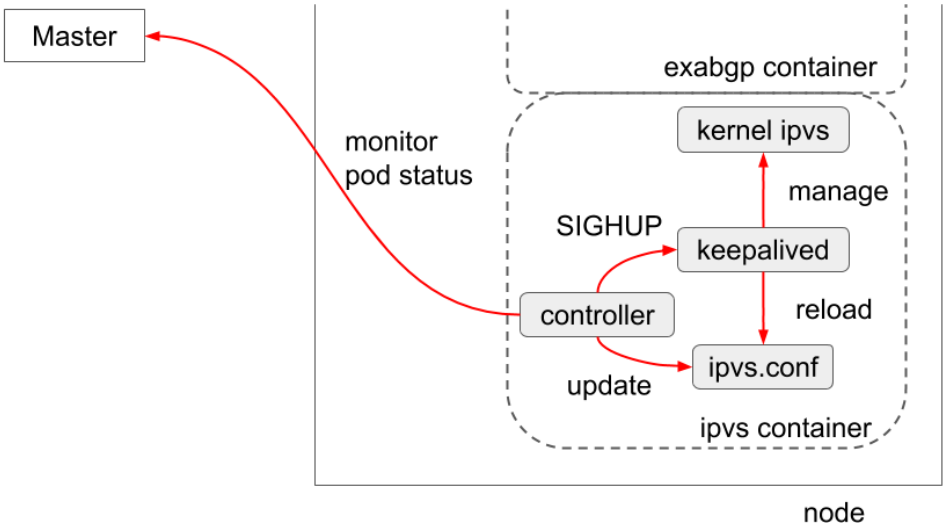


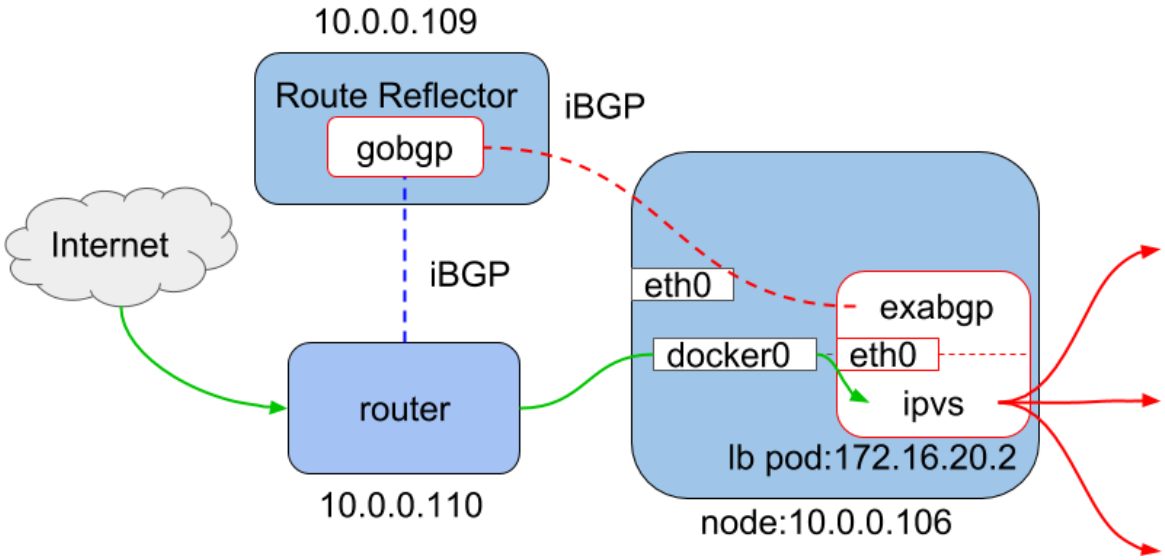
Figure 5.2: Implementation

```
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 5.3: 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 5.4: 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 5.5: (a) Network path by the exabgp container. (b) Required settings in the exabgp container.

container net namespace. A configuration of exabgp is shown in .2.

5.4 ingress controller

5.5 choice bgp software

5.6 Summary

Chapter 6

Performance analysys of a portable load balancer

This chapter discuss the performance level of a single ipvs load balancer. We evaluated the load balancer functionality using physical servers in on-premise data center and compared performance level with existing iptables DNAT and nginx as a load balancer. We also carried out the same performance measurement in GCP and AWS to show the containerized ipvs load balancer is runnable even in the cloud environment. The following sections explain these in further detail.

6.1 On-premise experiment with 1Gbps Load balancer

6.1.1 Benchmark method

We measured the performance of the load balancers using the wrk. Figure 6.1 illustrates a schematic diagram of our experimental setup. Multiple *pods* are deployed on multiple nodes in the Kubernetes cluster. In each *pod*, a Nginx web server that returns the IP address of the *pod* are running. We then set up the IPVS, iptables DNAT, and Nginx load balancers on one of the nodes(the top right node in the Figure 6.1).

We measured the throughput, Request/sec, of the web service running on the Kubernetes cluster as follows: The HTTP GET requests are sent out by the wrk on the client machine toward the nodes, 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* will return 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.

Figure **??** shows an example of the command-line for wrk and the corresponding output. The command-line in Figure **??** 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 information including per thread statistics, error counts, Request/sec and Transfer/sec.

Figure **??** shows hardware and software configuration used in our experiments. We configured Nginx HTTP server to return a small HTTP content, the IP address of the *pod*, to make a relatively severe condition for load balancers. The size of the character string making up an IP address is limited to 15 bytes. If we had chosen the HTTP response size so that most of the IP packet resulted in maximum transmission unit(MTU), the performance would have been limited by the Ethernet bandwidth. However, since we used small HTTP responses, we could purely measure the load balancer performance.

We used a total of eight servers; 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 figure. The hardware we used had eight physical CPU cores and a 1Gbps NIC with 4 rx-queues.

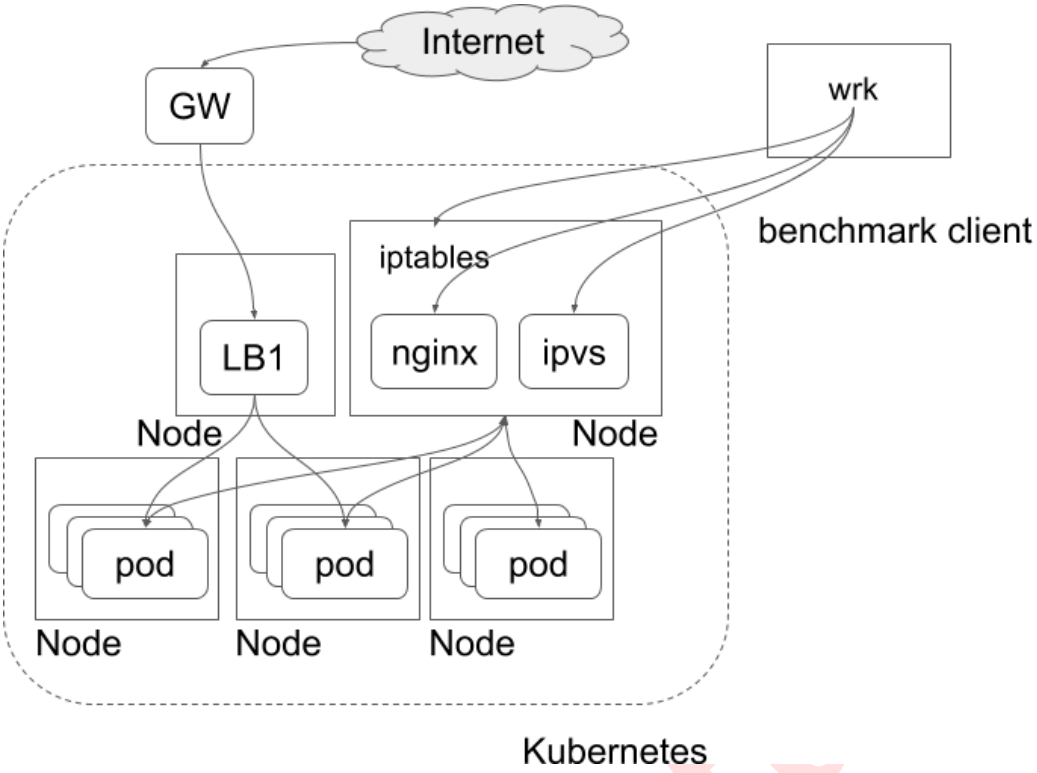


Figure 6.1: Benchmark setup


```
[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 6.1

Figure 6.2

[Hardware Specification]
CPU: Xeon E5-2450 2.10GHz x 8 (with 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 6.2

6.1.2 Results

Effect of multicore proccessing

Effect of overlay network

Comparison of different load balancer

Figure 6.5 shows the IPVS load balancer performance, that is, the throughput (Request/sec) of the Nginx web server *pods* in our experiments. As for the overlay network, we measured the performance for three flannel backend modes, host-gw (Figure 6.5(a)), vxlan (Figure 6.5(b)) and udp (Figure 6.5(c)). The following RSS and RPS setting were compared:

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

Except for the udp cases, we can see the trend in which the throughput linearly increases as the *pod* number increases and then it eventually saturates. The saturated performance levels indicates the maximum performance of the IPVS load balancer. The maximum performance limits depend on the flannel backend mode type and the (RSS, RPS) settings. From the results in Figures 6.5(a,b), it can be seen that if we turn off distributed packet processing, *i.e.*, when "(RSS, RPS) = (off, off)", performance degrades significantly. In this case, the performance bottleneck is primarily due to packet processing in a single core.

If we compare the results for the cases when "(RSS, RPS) = (on, off)" and "(RSS, RPS) = (off, on)", the latter is better than the former. This is understandable, since in the case of "RPS = on", eight physical cores can be used whereas in the case of "RSS = on" only four cores can be used, on the hardware used in our experiment. The performance bottleneck of the case when "RSS = on" is considered to be due to the fact that the packet processing is only done on four CPU cores. At first, it was not clear what caused the performance limit for the case when "RPS = on", however we now suspect this is due to 1Gbps bandwidth limitation. A packet level

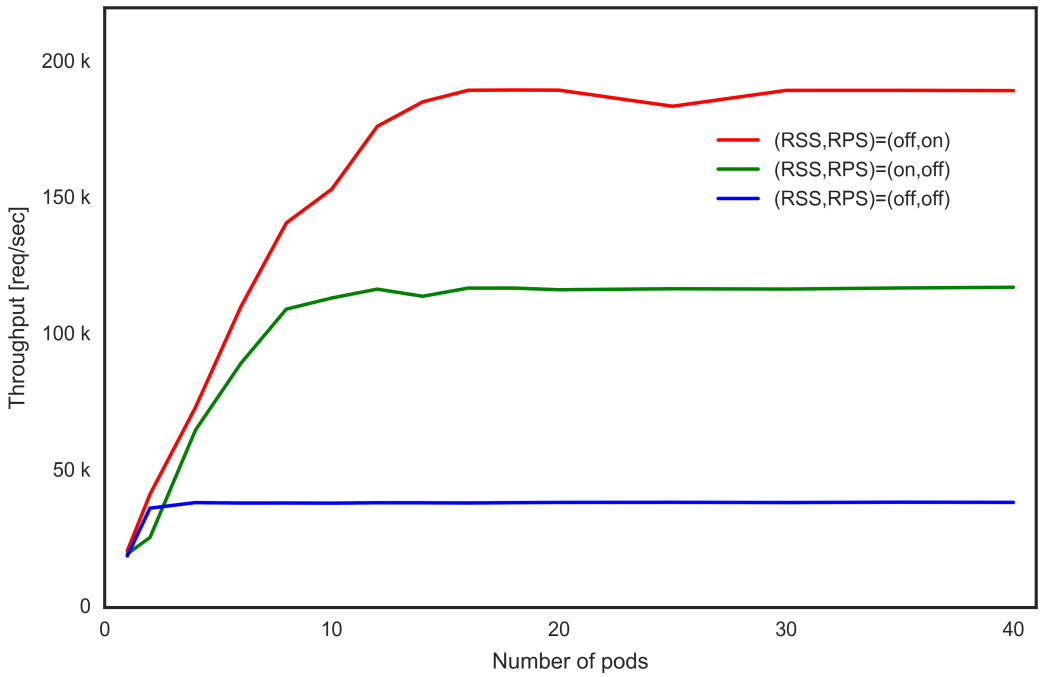


Figure 6.3: IPVS and iptables comparison

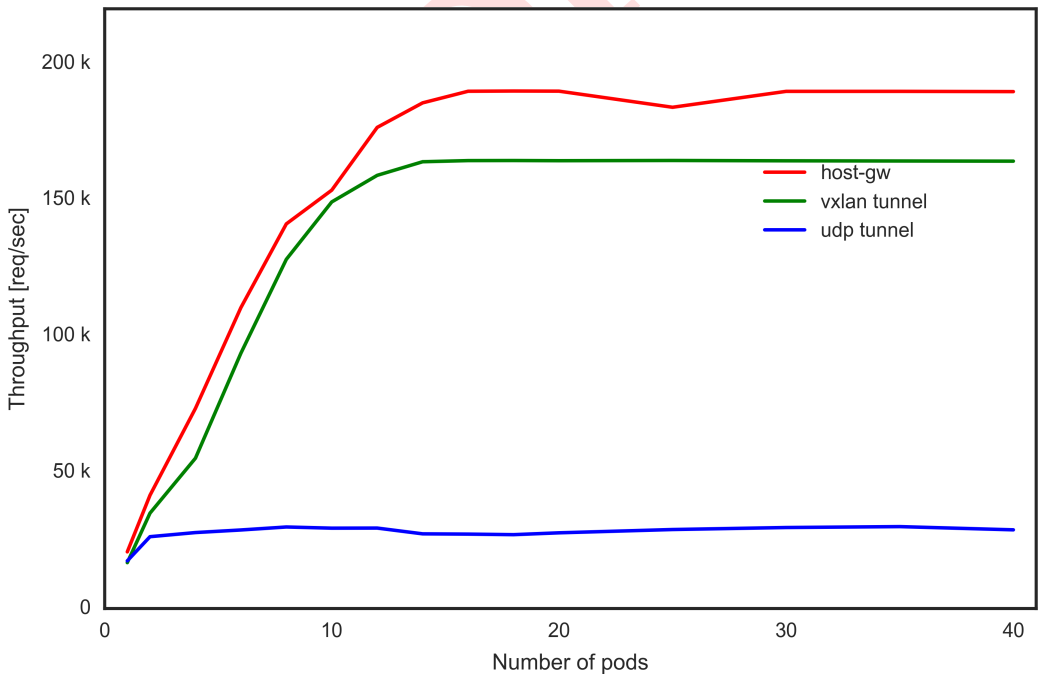


Figure 6.4: iptables results

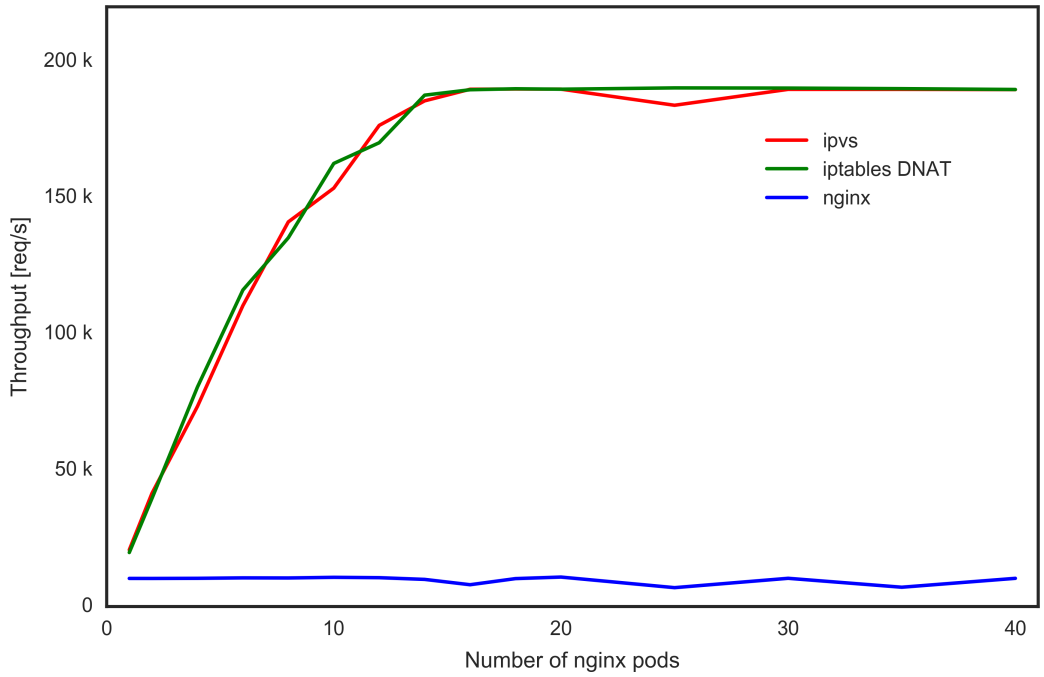


Figure 6.5: IPVS results

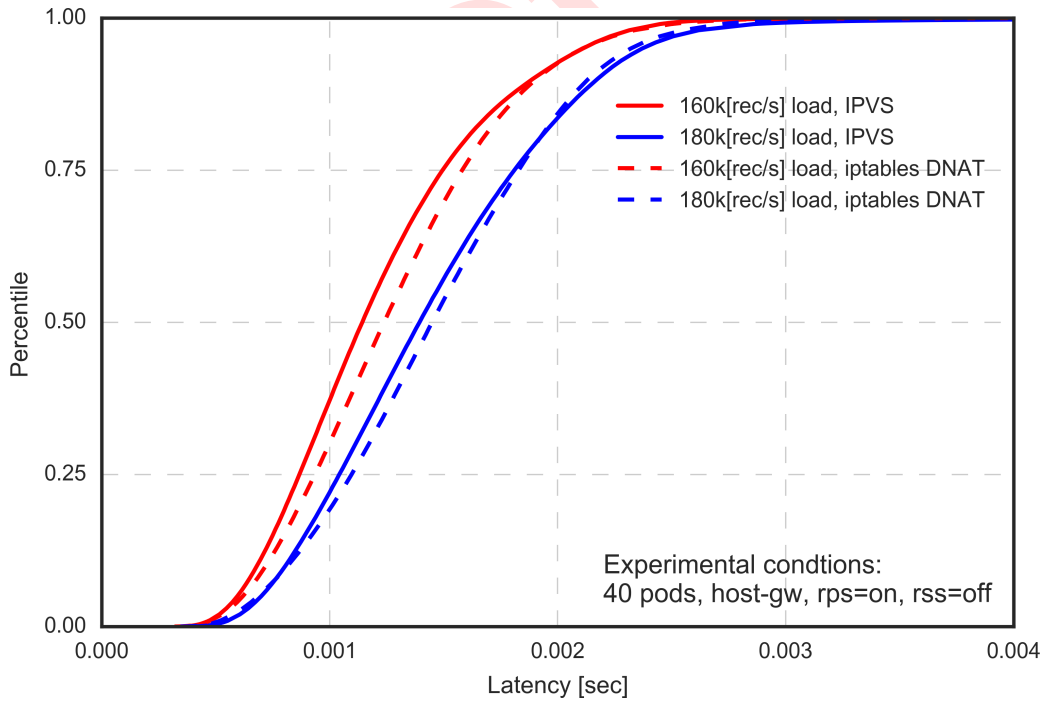


Figure 6.6: Latency cumulative distribution function

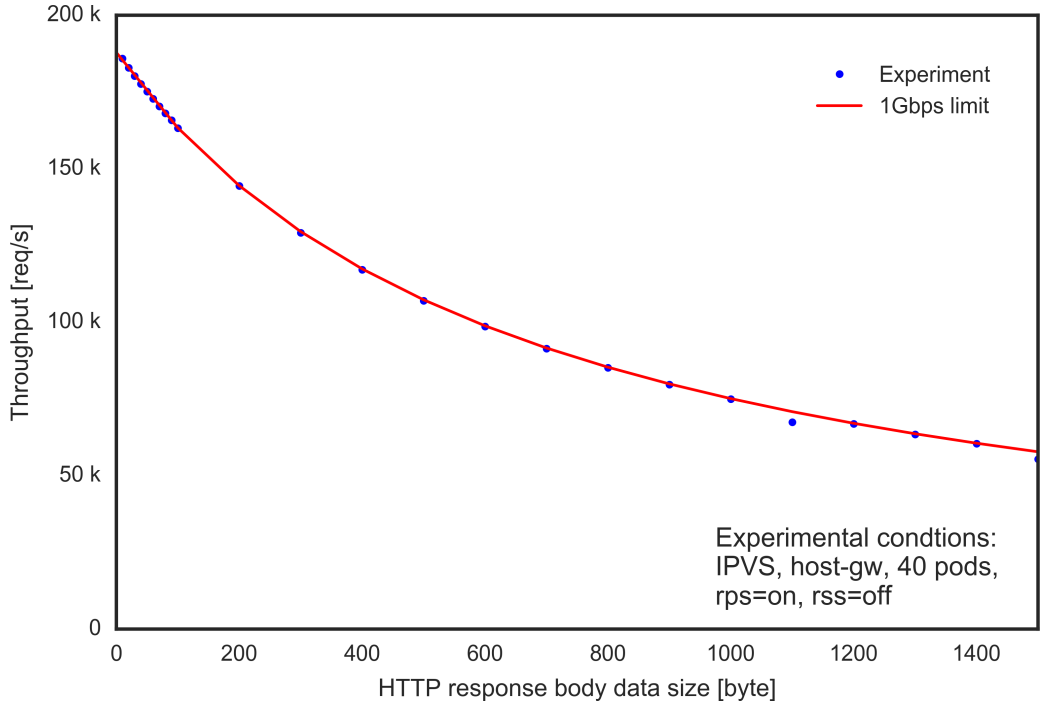


Figure 6.7: Performance limitation due to 1Gbps bandwidth

analysis using tcpdump[26] revealed that 622.72 byte of extra HTTP headers, TCP/IP headers and ethernet frames are needed for each request in the case of the wrk benchmack program. This results in the upper limitation of 196,627 [req/s], where the date size of HTTP response body is 13 byte, or typical data size in our experiment. Figure 6.7 shows upper limitation of the performance level for 1Gbps ethernet together with actual benchmark results and we can conclude that when "RPS = on", IPVS performance is limited by bandwidth.

If we compare the performances 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.

Figure 6.3 compares the performance measurement results among the IPVS, iptables DNAT, and Nginx load balancers with the condition of "(RSS, RPS) = (off on)". The proposed IPVS load balancer exhibits almost equivalent performance as the existing 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 Nginx as a load balancer is expected to be similar to the performance as a web server. Figure 6.6 compares Cumulative Distribution Function(CDF) of the load balancer latency at the constant load. The latencies are a little bit smaller for IPVS, however we consider the difference almost negligible. For example, the median value at 160K[req/s] load for IPVS and iptables DNAT are, 1.1 msec and 1.2 msec, respectively. So, we can conclude our proposed load balancer showed no performance degradation while providing potability.

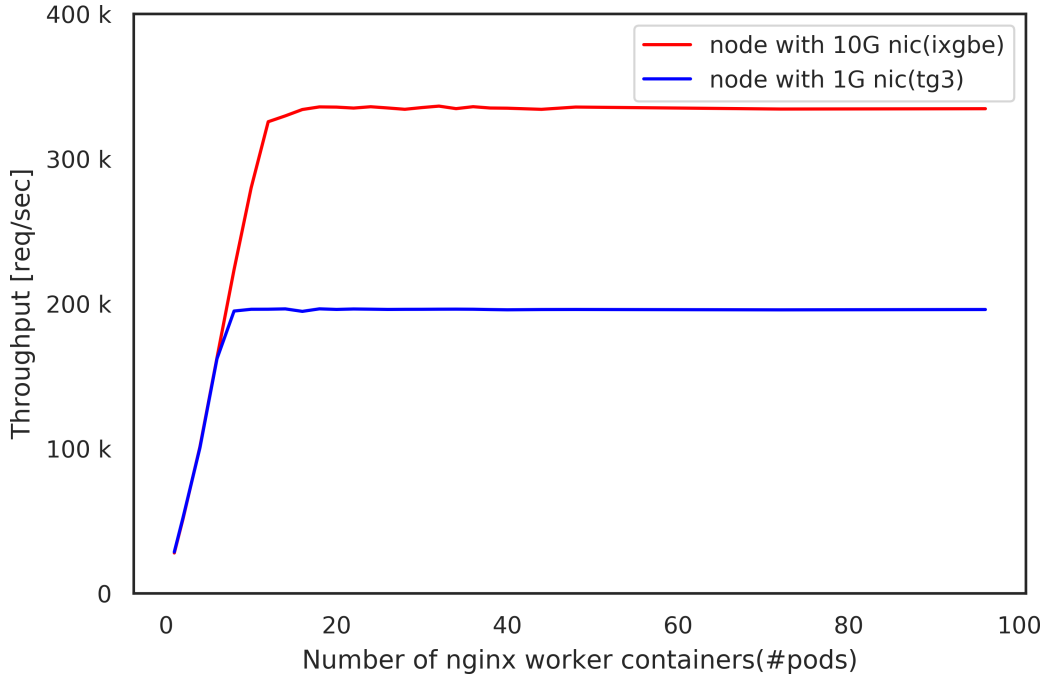


Figure 6.8: GCP

L3DSR

6.2 On-premise experiment with 10Gbps Load balancer

6.3 Cloud experiment

Fig. **??** (??) and Fig. **??** (??) show the load balancer performance levels that are measured in GCP and AWS, respectively. These are aimed to show that our proposed load balancer can be run in cloud environments and also functions properly.

Both results show similar characteristics as the experiment in an on-premise data center in Fig. **??** (**??**), where throughput increased linearly to a certain saturation level that is determined by either network speed or machine specifications. Since in the cases of cloud environments we can easily change the machine specifications, especially CPU counts, we measured throughput with several conditions of them. From the first look of the results, since changing CPU counts changed the load balancer's throughput saturation levels, we thought VM's computation power limited the performance levels. However, since there are cases in the cloud environment, where changing the VM types or CPU counts also changes the network bandwidth limit, a detailed analysis is further required in the future to clarify which factor limits the throughput in the cases of these cloud environments. Still, we can say that the proposed ipvs load balancers can be run in GCP and AWS, and function properly until they reach the infrastructure limitations.

6.4 Resource Consumption

6.5 Summary

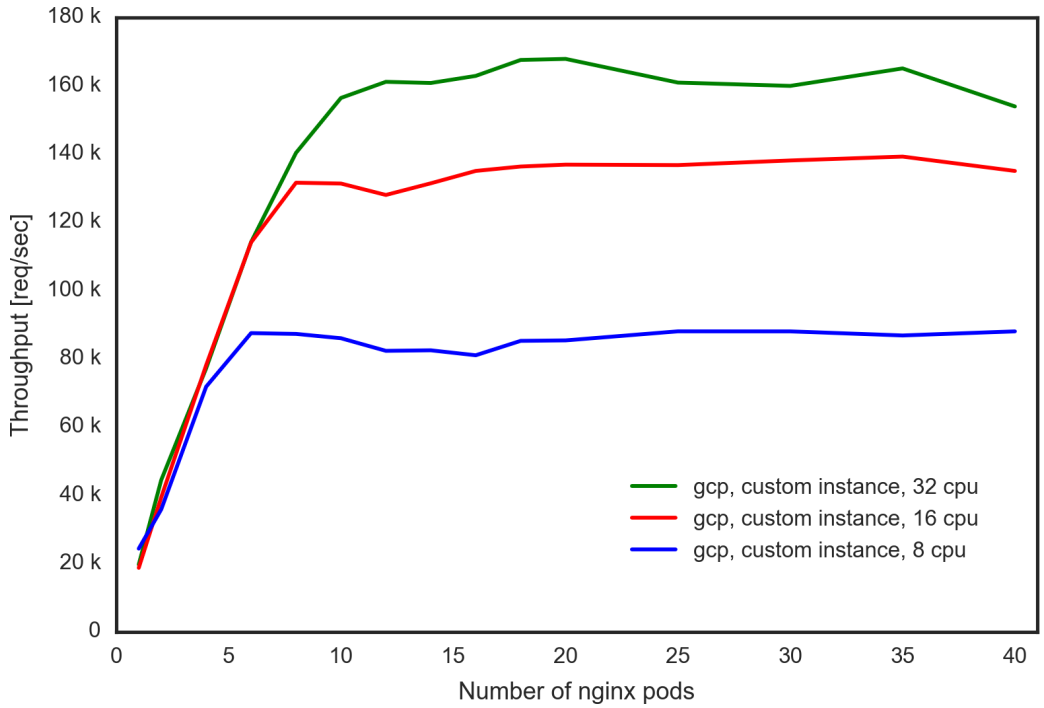


Figure 6.9: GCP

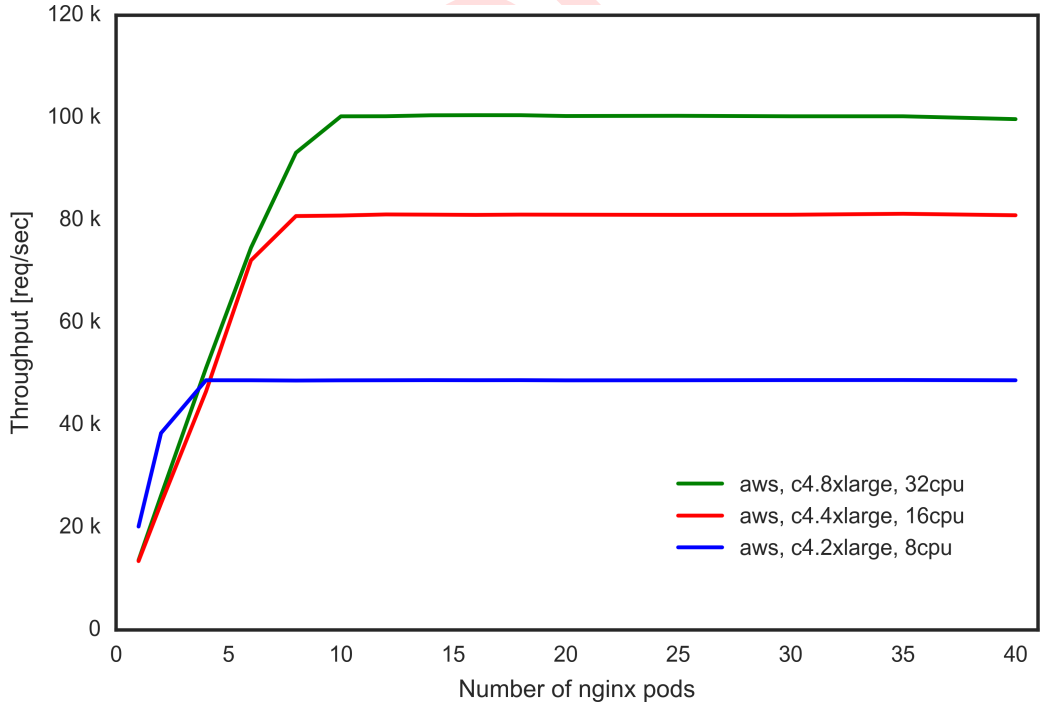


Figure 6.10: AWS with Node x 6, Client x 1, Load balancer x 1. Custom instance.

Chapter 7

Evaluation of redundancy architecture

This chaper discuss the redundancy and scalability.... (2) Redundancy and Scalability: We evaluated ECMP functionality by watching routing table updates on the router when the new load balancer is added or removed. We also evaluated the performance level by changing the number of load balancers.

The following subsections explain the evaluation in detail.

7.1 Redundancy and Scalability

The ECMP technique is expected to make the load balancers redundant and scalable since all the load balancer containers act as active. We examined the behavior of the ECMP routing table updates, by changing the number of the load balancers. After that, in order to explore the scalability, we also measured the throughput from a benchmark client with ECMP routes when multiple of the ipvs container load balancers are deployed.

Fig. **??** shows the schematic diagram of the experimental setup and also summarizes hardware and software specifications. Notable differences from the previous throughput experiment in Fig. 6.1 are as follows; 1) Each load balancer pods now consists of both an ipvs container and an exabgp container. 2) The routing table of the benchmark client is updated by BGP protocol through a route reflector. 3) The NIC of the benchmark client has been changed to 10 Gbps card since now we have multiple of ipvs container load balancers that are capable of filling up 1 Gbps bandwidth. 4) Some of the software have been updated to the most recent versions at the time of the experiment.

First, we examined ECMP functionality by watching the routing table on the benchmark client. Table 7.2 (a) shows the routing table entry on the router when a single load balancer pod exists. From this line, 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. It also shows that this routing rule is controlled by zebra.

When the number of the load balancer pods is increased to three, we can see the routing table entry in Table 7.2 (b). We have 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 all 1. The update of the routing entry was almost instant as we increased the number of the load balancers.

Table 7.2 (c) shows the case where we additionally started new service with two load balancer pods with service addresses in 10.1.2.0/24 range. We could 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(10.0.0,105,10.0.0,106,10.0.0,107). The update of the routing entry was almost instant as we started the load balancers for the second service.

Fig. 7.2 shows histogram of the ECMP update delay, where we measured the delays until the number of running ipvs pods is reflected in the routing table on the benchmark client, as we change the number of the ipvs pods 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 during the 20 hours experiment was 10 seconds. We can conclude that ECMP routing update in our proposed architecture is quick enough.

We also carried out throughput measurement to show that our proposed architecture increases the throughput as we increase the number of the load balancers. Fig. 7.3 shows the results of the measurements.

[Hardware Specification] CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading) Memory: 32GB NIC: Broadcom BCM5720 Giga bit (Node x 6, Client x 1) CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading) Memory: 32GB NIC: Intel X550 (Load Balancer 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 7.1

There are four solid lines in the figure, each corresponding the throughput result when there are one through four of the proposed load balancers. 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), which increases linearly as we increases the number of the load balancers. The dotted line in the figure shows the throughput result when there were five load balancers. It had almost the same performance level as the case when there were four load balancers, and did not scale further. We suspect that this was because we used up CPU power of the benchmark client since the CPU idle was 0% when there were more than four load balancers. We expect that replacing the benchmark client with more powerful machines, or changing the experimental setup so that multiple benchmark clients can access the load balancers through an ECMP router, will improve the performance level further.

Fig. 7.4 shows the throughput measurement results when we periodically changed the number of the load balancers. The red line in the figure shows the number of the ipvs load balancer pods, which we changed randomly 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, and after removing some load balancers, the traffic to them is immediately directed to the existing load balancers.

7.2 Resource Consumption

10.1.1.0/24 via 10.0.0.106 dev eth0 proto zebra metric 20

(a)

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)

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)

Table 7.2: ECMP routing tables.

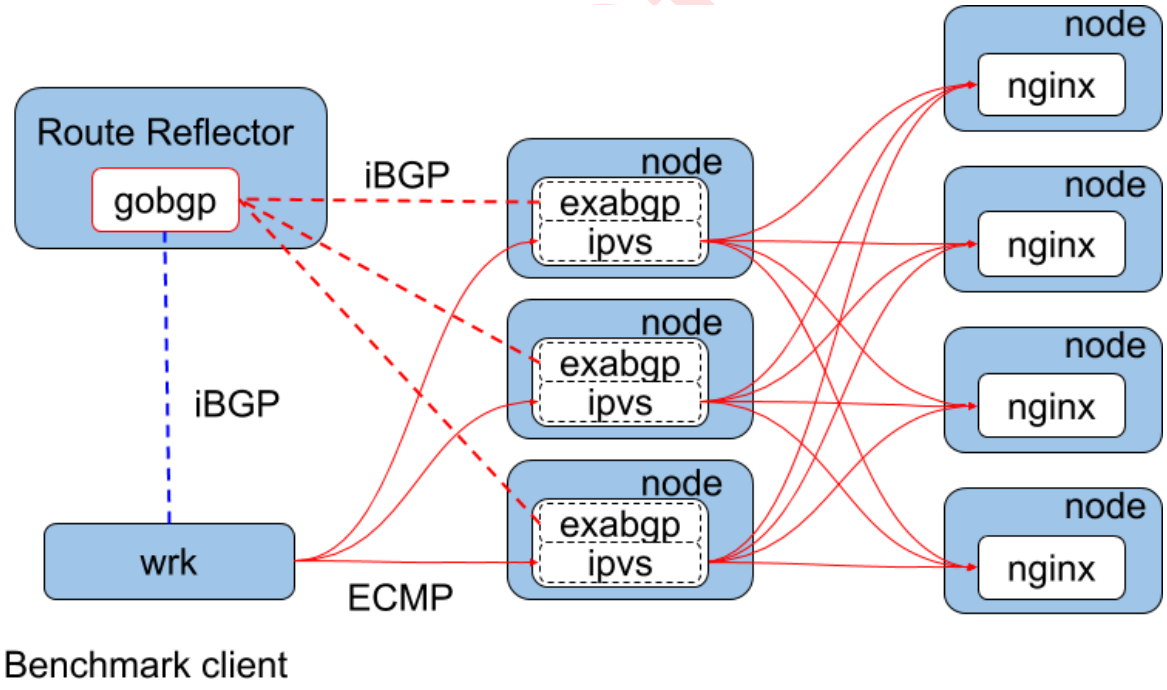


Figure 7.1

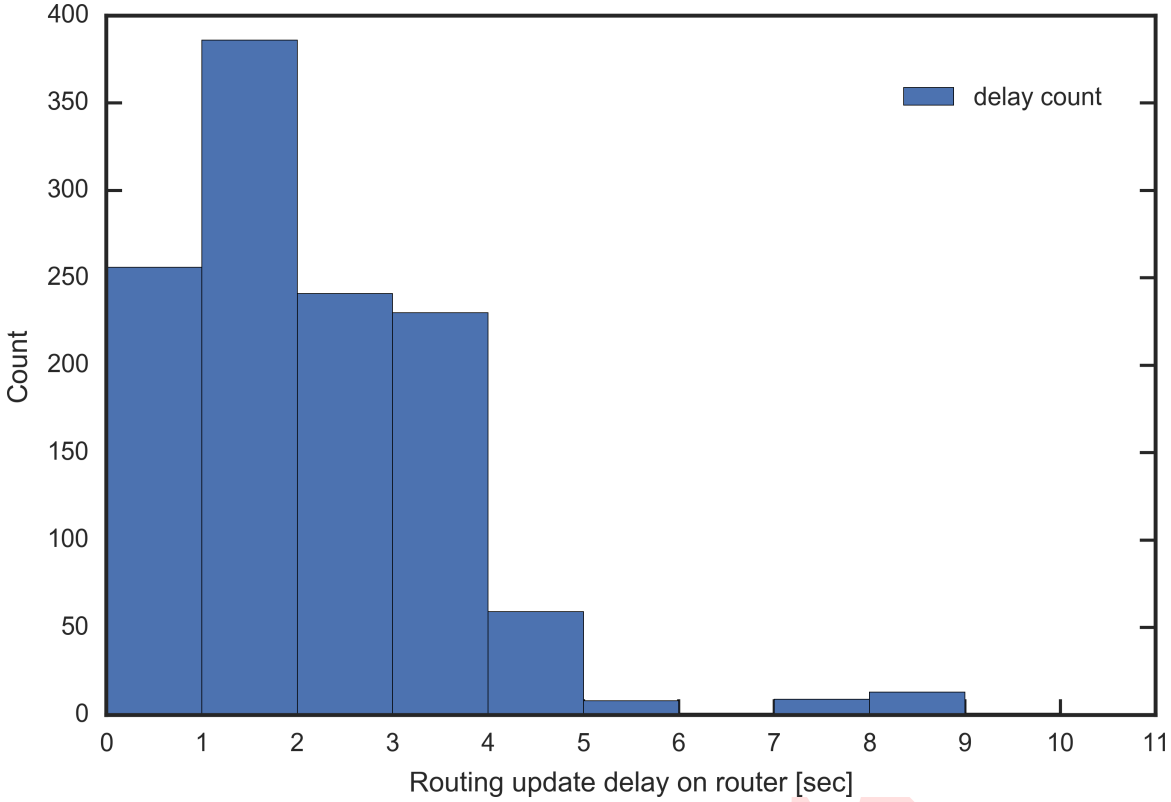


Figure 7.2: Caption 2

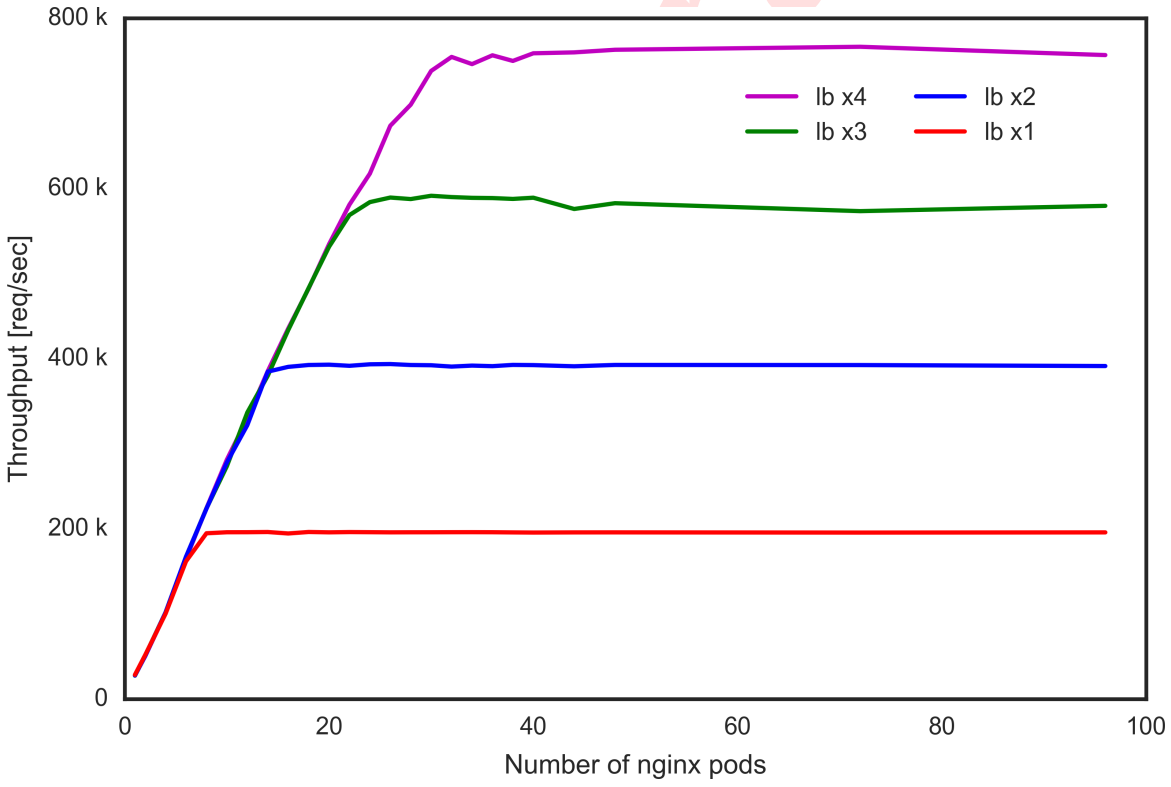


Figure 7.3: Caption 1

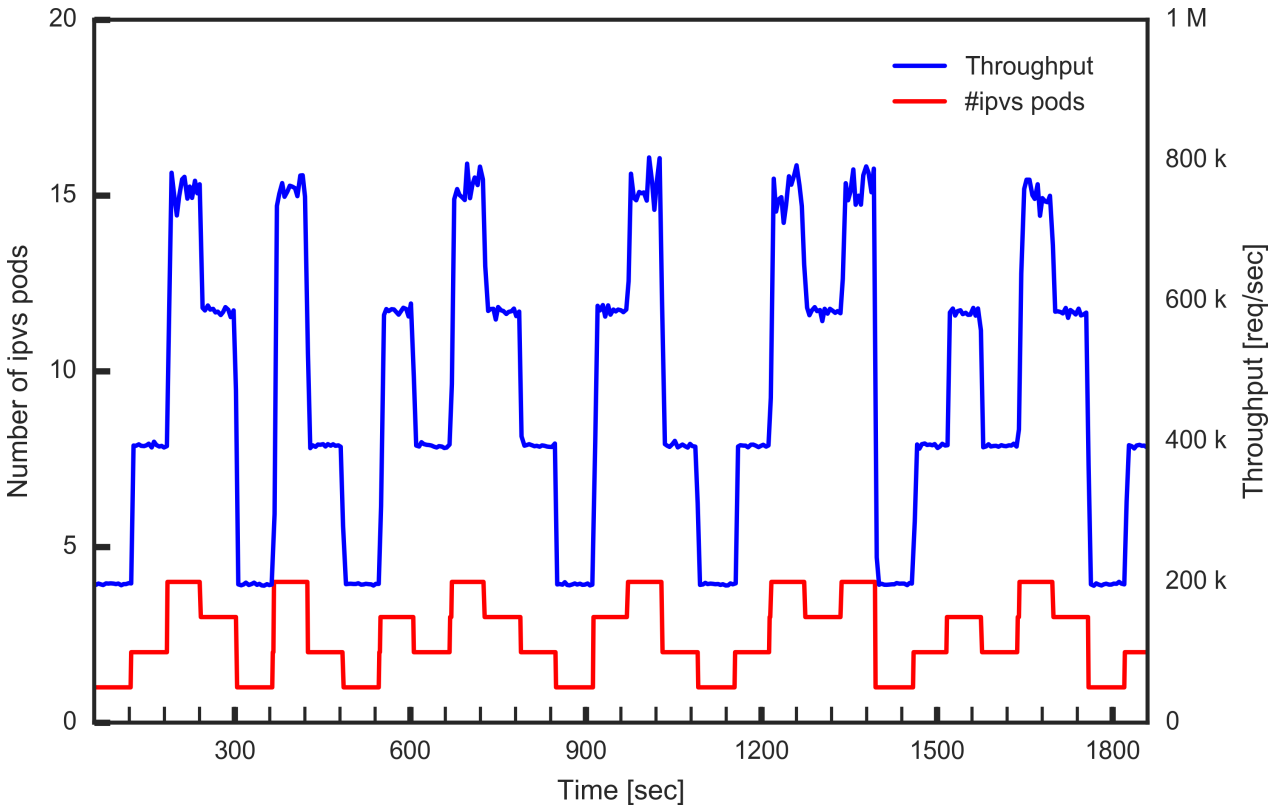


Figure 7.4: Caption 2

Chapter 8

Conclusion

8.1 Conclusions

In this paper, we 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. We implemented an experimental web cluster system with multiple of load balancers and web servers using Kubernetes and OSSs on top of standard Linux boxes to prove the functionality of the proposed architecture. We conducted performance measurements and found that the ipvs based load balancer in container functioned properly both in on-premise datacenter and cloud environments while it showed the comparable performance levels as the existing iptables DNAT based load balancer. We also carried out expriments to verify the feasibility of ECMP redundancy in on-premise data center, and revealed that it functions properly with linear scalability up to four load balancers. 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.

8.2 Conclusions

In this paper, we proposed a portable load balancer for the Kubernetes cluster systems that is aimed at facilitating migration of container clusters for web services. We implemented a containerized software load balancer that is run by Kubernetes as a part of container cluster, using Linux kernel's IPVS, as a proof of concept. In order to discuss the feasibility of the proposed load balancer, we built a Kubernetes cluster system and conducted performance measurements. Our experimental results indicate that the IPVS based load balancer in container improves the portability of the Kubernetes cluster system while it shows the similar performance levels as the existing iptables DNAT based load balancer. We also clarified that choosing the right operating modes of overlay networks is important for the performance of load balancers. For example, in the case of flannel, only the vxlan and udp backend operation modes could be used in the cloud environment, and the udp backend significantly degraded their performance. Furthermore, we also learned that the distribution of packet processing among multiple CPUs was very important to obtain the maximum performance levels from load balancers.

The limitations of this work that authors aware of include the followings: 1) We have not discussed the load balancer redundancy. Routing traffic to one of the load balancers while keeping redundancy in the container environment is a complex issue, because standard Layer 2 rendandacy protocols, e.g. VRRP or OSPF[27] that uses multicast, can not be used in many cases. Further more, providing uniform methods independent of various cloud environments and on-premise datacenter is much more difficult. 2) Experiments are conducted only in a 1Gbps network environment. The experimental results indicate the performance of IPVS may be limited by the network bandwidth, 1Gbps, in our experiments. Thus, experiments with the faster network setting, e.g. 10Gigabit ethernet, are needed to investigate the feasibility of the proposed load balancer.

3) We have not yet compared the performance level of proposed load balance with those of cloud provider's load balancers. It shoud be fair to compare the performance of proposed load balancer with those of the combination of the cloud load balancer and the iptables DNAT. The authors leave these issues for future work and they will be discussed elsewhere.

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

[27] John Moy. Ospf version 2. 1997.

.1 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"
}
```

.2 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;
   }
}
```

.3 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
```


.4 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
```