

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

Kento Aida(Chair)	National Institute of Informatics / Sokendai
Atsuko Takefusa	National Institute of Informatics / Sokendai
Michihiro Koibuchi	National Institute of Informatics / Sokendai
Takashi Kurimoto	National Institute of Informatics / Sokendai
Atsuko Takefusa	National Institute of Informatics / Sokendai
Shigetoshi Yokoyama	National Institute of Informatics / Gunma University

Acknowledgments

[Filled in later]

Abstract

Today a vast majority of the people in the world use PCs or smartphones to communicate with friends, check up the news, watch the videos, play games, etc., through the Internet. These services are called web services because they utilize web technology through HTTP(S) protocols. Web services are generally provided by a cluster of web server programs, database server programs, and load balancers. Web service providers deploy these programs on a cluster of physical servers in an on-premise data center or on a cluster of VMs in cloud infrastructures.

Recently Linux container technology and clusters of the containers have come to draw attention because they 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 data centers. Service migrations prevent a service to be locked-in a single cloud provider or a single location and enable users to meet their business needs, e.g., preparing for a natural disaster, lower the cost of infrastructure and comply the regulations.

In order for a web service to be deployed easily in different base infrastructures, container management systems are often used. However existing container management systems lack the generic capability to route the traffic from the internet into web service 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 base infrastructures, it becomes users responsibility to route the traffic into their cluster while keeping the redundancy and scalability. This means that users are happy only in the major cloud providers including GCP, AWS, and Azure; thus they could easily be locked-in those infrastructures.

In this dissertation, the author proposes a load balancer architecture that is usable

in any of the base infrastructure, including cloud providers and on-premise data centers, in order to free users from lock-ins. The proposed load balancer architecture utilizes software load balancers with container technology to make the load balancers runnable in any base infrastructure. It also utilizes ECMP technology to make multiple load balancers active, and thereby to provide redundancy and scalability.

The author implemented a containerized software load balancer that is run by Kubernetes as a part of container cluster, using Linux kernel's IPVS. In order to discuss the feasibility of the proposed load balancer, performance measurements are conducted in 1 Gbps network environment. It was shown that the proposed load balancers are runnable in an on-premise data center, GCP and AWS. It can be said that the proposed load balancers are portable. The throughput levels of a load balancer are dependent on settings for multi-core packet processing. It was shown to be better to use as many CPU cores as possible for packet processing. The throughput levels are also very dependent on the overlay network backend mode and overhead of the container network, i.e., veth+bridge. The host-gw mode where no tunneling is used resulted in the best performance level, and the vxlan mode resulted in the second best. Although the overheads of the container network are invisible in 1 Gbps network environment, they are visible in 10 Gbps network environment. In the experiment in 1 Gbps network environment, the ipvs-nat load balancer in the container had the same performance level as load balancing function of iptables DNAT in the node net namespace. Furthermore, the performance level of ipvs-tun load balancer in a container with the L3DSR setup was about 1.5 times larger than that of iptables DNAT. Therefore in 1 Gbps network environment, the proposed load balancer is portable while it has the 1.5 times better performance level or the same performance level depending on the mode of operation.

Also implemented is the ECMP setups where multiple of the load balancer containers are deployed, each advertising the route to the service VIP. The ECMP technique makes 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 by utilizing multiple of load balancers simultaneously, the throughput of the total system is increased significantly. These characteristics are evaluated by checking the routing table of the upstream router and throughput measurement. 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 maximum performance levels of the cluster of load balancers scaled linearly as the number of the load balancer pods was increased up to four of them.

The author also extended the throughput measurement into 10 Gbps network environment. It was revealed that ipvs-nat and ipvs-tun load balancers in containers had lower performance levels compared with the iptables DNAT. By setting up the load balancing table in node net namespaces, the performance levels of ipvs-nat and ipvs-tun became closer to that of the iptables DNAT, which is suggesting that the overhead of the container network is no longer invisible in 10 Gbps network environment. The author is currently implementing and evaluating a novel software load balancer using XDP technology to provide a better alternative to ipvs as a portable load balancer.

The outcome of this 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 this 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.

Contents

List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Motivation	1
1.1.1 Web application cluster	1
1.1.2 Migration of web application cluster	3
1.1.3 Ideal infrastructure for migration of web application	4
1.2 Infrastructure for web applications	5
1.2.1 On-premise data center	5
1.2.2 Cloud computing	6
1.2.3 Container technology	7
1.2.4 Container Orchestrator	9
1.3 Portable software load balancer	12
1.3.1 Load balancer for container clusters	12
1.3.2 Problems of Kubernetes	13
1.3.3 Proposed solution	14
1.3.4 Contribution	15
1.4 Outline	16
2 Background	17
2.1 Container Network	18
2.1.1 bridge+veth	18

2.1.2	macvlan, ipvlan	18
2.1.3	hostnetwork	18
2.2	Overlay Network	18
2.2.1	General concept	18
2.2.2	Flannel	19
2.2.3	Calico	21
2.3	Multicore Packet Processing	21
2.3.1	rss	21
2.4	Linux vritual server	24
2.5	Summary	25
3	Architecture and Implementation	27
3.1	Architecture	27
3.1.1	Problem of Conventional Architecture	28
3.1.2	Load balancer in container	31
3.1.3	Redundancy with ECMP	32
3.2	Implementation	34
3.2.1	Experimental system architecture	34
3.2.2	Ipv6 container	36
3.2.3	BGP software container	38
3.3	Summary	42
4	Performance Evaluation	43
4.1	Performance analysis of proposed load balancer	44
4.2	Cloud experiment	57
4.3	Redundancy with ECMP	60
4.4	Summary	68
5	Further Improvement	69
5.1	Throuput of ipvs-nat, ipvs-tun and iptables DNAT	69
5.2	Throuput of ipvs-nat, ipvs-tun and iptables DNAT	72
5.3	XDP load balancer	73
5.4	Summary	73

Contents	xi
6 Related Work	75
6.1 Related Work	75
7 Conclusion and future work	79
7.1 Conclusions	79
Appendix A ingress controller	87
Appendix B ECMP settings	91
B.1 Exabgp configuration on the load balancer container.	91
B.2 Gobgpd configuration on the route reflector.	92
B.3 Gobgpd and zebra configurations on the router.	93
Appendix C Analysis of the performance limit	95
Appendix D VRRP	99
D.1 VRRP	99

List of Figures

1.1	An example of web application cluster.	2
1.2	Migration of web application cluster to different locations.	4
1.3	An ideal global container infrastructure.	5
1.4	The difference in physical server usage between (a) Bare Metal servers, (b) Virtual Machine and (c) Container technology.	7
1.5	A web application cluster and container orchestrator.	9
1.6	Load balancer for container clusters.	12
2.1	The network architecture of an exemplified container cluster system. .	18
2.2	Frame diagram	20
2.3	RX/TX queues of the hardware	22
3.1	Conventional architecture of Kubernetes clusters	30
3.2	Kubernetes cluster with proposed load balancer.	31
3.3	The proposed architecture of load balancer redundancy with ECMP .	33
3.4	An experimental container cluster with proposed redundant software balancers	35
3.5	Implementation of ipvs container.	37
3.6	An example of ipvs.conf	39
3.7	Example of IPVS balancing rules	40
3.8	Network path by the exabgp container	40
4.1	Benchmark setup	45
4.2	Effect of multicore packet processing on ipvs throughput	49
4.3	Effect of multicore packet processing on iptables DNAT throughput .	49

4.4	Effect of flannel backend modes on ipvs throughput	51
4.5	Effect of flannel backend modes on iptables DNAT throughput	51
4.6	Throughput of ipvs, iptables DNAT and nginx	52
4.7	Latency for ipvs and iptables DNAT	53
4.8	CPU usage of the ipvs and iptables DNAT	54
4.9	Experimental setup for L3DSR throughput measurement	55
4.10	Throughput of L3DSR using ipvs-tun.	56
4.11	Throughput measurement results in GCP	59
4.12	Throughput measurement results in AWS	59
4.13	Benchmark setup for ECMP experiment	60
4.14	Throughput of ECMP redundant load balancer	65
4.15	Throughput responsiveness	66
4.16	ECMP update delay histogram	67
5.1	Packet flow of ipvs-nat and iptables DNAT.	70
5.2	Packet flow of ipvs-tun.	71
5.3	Throughput of load balancers in 10 Gbps.	72
5.4	Throughput of load balancers in node name space.	73
D.1	An alternative redundant load balancer architecture using VRRP.	100

List of Tables

1.1	Container orchestrator comparison.	11
2.1	Viable flannel backend modes. In cloud environment tunneling using vxlan or udp is needed.	21
3.1	Comparison of open source BGP agents	36
3.2	Required settings in the exabgp container	41
4.1	Benchmark command line and output example	46
4.2	Hardware and software specifications	47
4.3	Virtual Machine specifications in GCP experiment	57
4.4	Virtual Machine specifications in AWS experiment	58
4.5	Hardware and software specifications for ECMP experiment	62
4.6	ECMP routing tables	63
5.1	Performance levels in 1Gbps and in 10Gbps.	71
5.2	Performance levels in pod namespace and in node namespace.	73
C.1	Request data size for 100 HTTP requests in wrk measurement.	97
C.2	Response data size for 100 HTTP requests in wrk measurement.	97
C.3	Header sizes of TCP/IP packet in Ethernet frame.	97

1

Introduction

1.1 Motivation

1.1.1 Web application cluster

Today, a great number of people in the world can not spend a day without using smartphones or personal computers(PCs) to retrieve information from the Internet for work or for daily life. For example, people use these devices to look up web pages, emails, social media and sometimes to play games. These services are often called web applications or web services, where information is delivered using Hyper Text Transfer Protocols(HTTP) or Hypertext Transfer Protocol Secure (HTTPS) from servers at the other end of the Internet. Web applications are provided by various organizations, including commercial companies, government, non-profitable organizations, etc.

For example, Google provides a variety of web services including, Gmail, Search engine, Google Suits, etc. Facebook provides social media service, Amazon provides shopping sites. Governments provides information regarding the service they provide

to their citizens. Schools often provide a syllabus to their students, which is important for campus life. The author calls those organizations that provide web applications, web application providers hereafter.

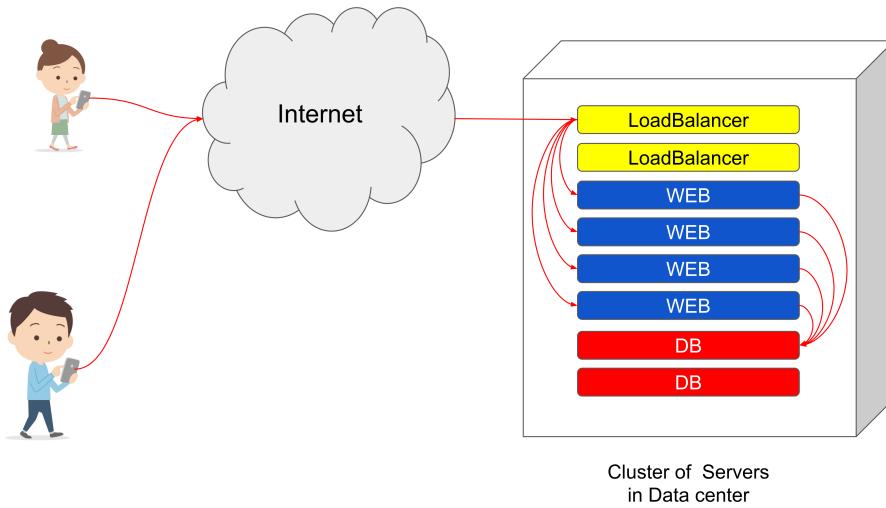


Figure 1.1: An example of web application cluster.

The load balancers distribute requests from clients to multiple web servers. The web servers form responses using data retrieved from the database servers and send it back to the clients. Sometimes the web servers also store and update important data into the database servers.

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 applications are usually computers located in a data center. In the data center multiple servers cooperate to fulfill the need of the clients. A group of these servers is often called a web application cluster or a web cluster. Figure 1.1 shows schematic diagram of an example of a web application cluster.

In this example, there are two load balancers, four web servers and two database(DB) servers that work together to respond to the requests from clients. The load balancers distribute the requests from clients to multiple web servers. Then the web servers form responses using data retrieved from the database servers and send it back to the clients. Sometimes the web servers also store and update important data into the database servers.

1.1.2 Migration of web application cluster

As web applications become an essential part of daily life, an outage of the web application service is getting to be a bigger problem. If something happens to a web application cluster in a data center, people will not be able to access the necessary information.

For example, if web pages run by local government stops, people will not be able to access the information regarding public service. If a shopping site run by a company stops, customers can no longer buy products and the revenue of the company will be negatively affected. Outages of web applications by giant companies can have an even bigger impact. An outage of Gmail or Google search engine will probably stop most of the business activities around the world. Service down of Amazon.com affect buyers and many businesses that sell products on its platform.

In order to prevent such outages, preparing another web application cluster in a different location in the case for disasters is very important. For that purpose, it is desirable if a web application cluster can be easily migrated to a different data center. Migration of a web application cluster becomes more realistic with the use of Linux container technology, which is explained later.

Migration capability of a web application cluster also has other benefits. If an e-commerce service is successful in Japan, the company that runs the service might want to start the same service in other country, for examlpe, in Europe. In this situation, the company probably wants to migrate its web applictation cluster to somewhere in Europe, because, for European customers, responses from a web site in Europe is quicker than that from a web site in Japan.

Being able to migrate a web application cluster is very important for a variety of purposes, including disaster recovery, cost performance optimizations, meeting legal compliance and shortening the geographical distance to customers. These are the main concerns for web application providers in e-commerce, gaming, Financial technology(Fintech) and Internet of Things(IoT) field. Therefore it is important for a web application provider to be able to easily deploy and migrate their web applications among different infrastructure around the world. The purpose of this research is to propose infrastructures for web application providers, where they can easily deploy their services across the world, regardless of cloud providers or data centers they use.

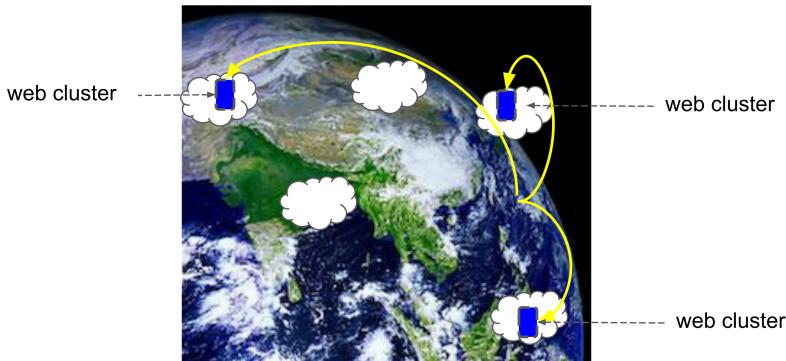


Figure 1.2: Migration of web application cluster to different locations.
It is desirable to be able to migrate a web cluster from one place to another
with the easiness of on push button.

1.1.3 Ideal infrastructure for migration of web application

In order to realize an easy migration of web applications, an ideal infrastructure probably have the following features; 1) have universal middleware to manage web clusters, 2) store data in globally consistent data storage, 3) route global traffic based on proximity to the client. Figure 1.3 shows an exemplified global container infrastructure having these features. Container orchestrators launch and manage container clusters. Important data are stored in globally consistent data storage, which is simillar to the Google spanner[8, 7] or CockroachDB[34]. Traffic is routed to the closest data center using anycast[31].

Each of these features is important and research efforts are on-going in many institutions. In this study, the author focuses on the research regarding container orchestrator as a universal middleware. By realizing global container infrastructure web application providers will be able to deploy their web applications whenever and wherever they want. Also, they will be able to move their web applications quickly depending on a variety of circumstances, including disaster recovery, cost performance optimization and compliance to government regulations due to trade wars, etc.

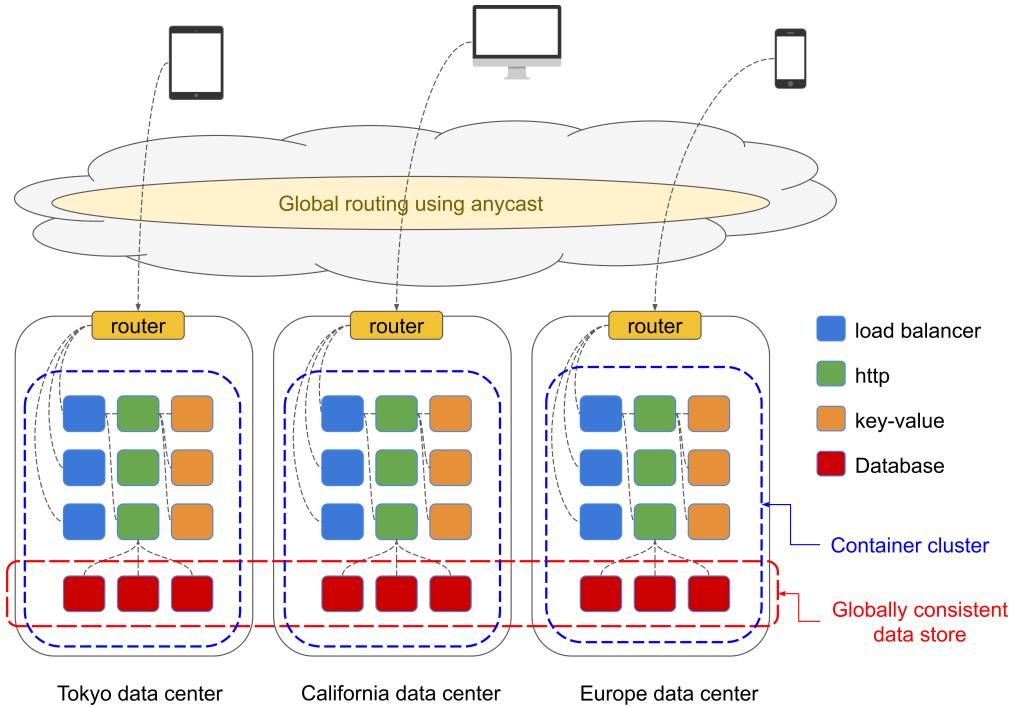


Figure 1.3: An ideal global container infrastructure.

Multiple web application clusters, each of which consisting of a cluster of containers, are deployed in three different data centers as an example. In each of the data center, container orchestrator manages the container cluster. Important data are stored in a globally consistent data store. Access from the client is routed to the closest data center using anycast.

1.2 Infrastructure for web applications

1.2.1 On-premise data center

Historically, most of the web application providers purchased servers and installed them in server housing facilities called data centers. In this type of infrastructure, web application providers typically need to sign a contract with data center company for server housing rack spaces, buy servers and install them in their rented racks by themselves. They also install OS and software stacks needed to run their web applications in the servers. Since web application providers place servers in their facilities(either owned or rented), and they are responsible for managing the servers, this type of infrastructure is often called on-premise infrastructure in contrast to Cloud

Computing infrastructure.

Preparing data centers, installing the servers and configuring software stacks for web application services often require a considerable amount of time, money and effort. If web application providers want to expand their services to different countries or if they want to prepare for natural disasters by preparing an additional web application cluster in a different data center, they probably need about the same amount of time, money and effort required to build their original infrastructures. Therefore migration of web application in this type of infrastructures has always been a daunting task.

1.2.2 Cloud computing

The emergence of Cloud Computing made many things easier for web application providers than before. Cloud computing utilizes a virtual machine(VM) technology, e.g. KVM, Xen, and VMware. Cloud computing service providers offer VMs to web application providers with pay-per-use billings.

Figure 1.4 compares different type of usages of a single physical server and Figure 1.4 (b) shows an example architecture of VM technology. VMs share a single physical server. A full OS including Linux kernel is running on top of the virtual machine represented by the hypervisor. Each VM behaves almost as same as a single physical server. Since VMs are fractions of a single physical server, server resources are utilized with finer granularities. Web application providers can start their services with a cluster of VMs, which is smaller than a cluster of physical servers, and hence resulting in lower cost.

Cloud providers generally prepare physical servers and software stacks for VMs before renting it to users, and they also provide an easy to use web user interfaces. As a result, users only need to click a few buttons on web browsers and wait for a few minutes before obtaining up-and-running VMs. This simplicity will bring agility to web application providers when they launch their services. And since computing resources are offered with per-second pay-per-use billings, web application providers can quickly reduce the cost by stopping excessive VMs, when the demand for computing power is scarce. This was impossible when web application providers purchased physical servers and used them as bare metal servers. In short, cloud computing brought users agility, flexibility, and cost-effectiveness.

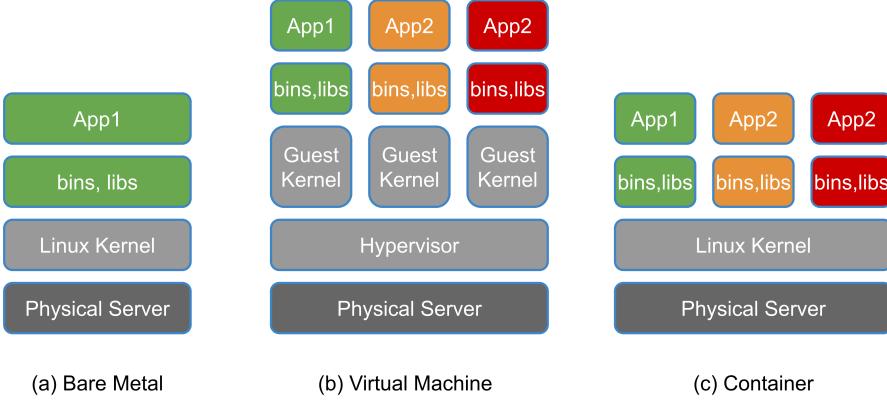


Figure 1.4: The difference in physical server usage between (a) Bare Metal servers, (b) Virtual Machine and (c) Container technology.

(a) Bare Metal servers is a word to describe conventional physical servers in contrast to Virtual Machines. On top of a Bare Metal server, an operating system and application programs are running. (b) Virtual Machine technology utilizes physical server hardware and a hypervisor. The hypervisor provides generic representations of server hardware, which are called virtual machines. A full operating system and applications are running on each of the virtual machines. (c) Container technology separates applications by containing them to their respective namespaces. Applications can not see each other's file systems, networks, users and process IDs unless they belong to the same namespace. Since container technology merely relies on Linux kernel's namespace function and optionally cgroup, a containerized process does not have any additional overhead compared with a process running on a conventional physical server and operating system. Container technology can be also utilized on top of virtual machines.

1.2.3 Container technology

More recently, Linux containers[29] have come to draw a significant amount of attention. Figure 1.4 (c) shows an example architecture of container technology. Linux containers are merely the processes with separate execution environments that are created using the Linux kernel's namespace feature. The namespace feature can isolate visibility of resources on a single Linux server.

Every process in a container is assigned to a certain namespace, and if two processes belong to different namespaces, they can not see each other's resources. Linux kernel implements filesystem namespace, PID namespace, network namespace,

user namespace, IPC namespace, and hostname namespace. For example, every filesystem namespace can have its own root filesystem, and every network namespace can have its own network devices and IP addresses. Therefore, it is possible to configure processes as if they were running in different Linux systems by assigning them to different namespaces, although they share kernel and hardware. While a VM needs to run a full OS on top of a hypervisor and hence imposes extra overhead, a process in Linux container is merely a process with a dedicated namespace and hence imposes no extra overhead.

The Linux container can run on any Linux systems including physical servers and VMs. Due to the widespread usage of Linux systems, the Linux container can run in most of the cloud infrastructures and on-premise data centers, which is beneficial for migrations.

Several management tools are available for Linux containers, including LXC[32], systemd-nspawn[13] and Docker[30]. These tools assign an appropriate namespace to a process upon the launch of itself and make it look like running in its own virtual Linux system. For example, container tools restore a file system from an archive file every time a container is launched. Container tools also set up separate network interfaces with separate IP addresses in the container's namespace.

The fact that each container has its own file system that is restored from a single archive file brings a significant benefit, i.e. a program binary and shared libraries are always exactly the same regardless of the base infrastructure. Therefore a process in a container is guaranteed to behave exactly the same manner, even if totally different data centers or cloud providers are used. This was not easy when there was no container technology. Because there are many flavors of Linux distributions, and even if the same distribution is used, there was always a chance that a slight difference in a program binary version or library versions could have broken the expected behavior.

In addition to that, containers can have own version of libraries in their respective filesystems, in other words, libraries in any container can be independently updated without influencing other containers. In conventional technologies, processes on a single server are dependent on common shared libraries, and hence updates of the library sometimes have caused unexpected side effects. Container tools alleviate these problems by packing necessary libraries into archives.

Thanks to these benefits container technologies are very attractive for web applica-

tions and their migrations. Considerable efforts in utilizing container technologies for web applications are ongoing. And to simplify the deployment of a complex web application that consists of interdependent container clusters, several container orchestrators(container management systems)¹ have been in development.

1.2.4 Container Orchestrator

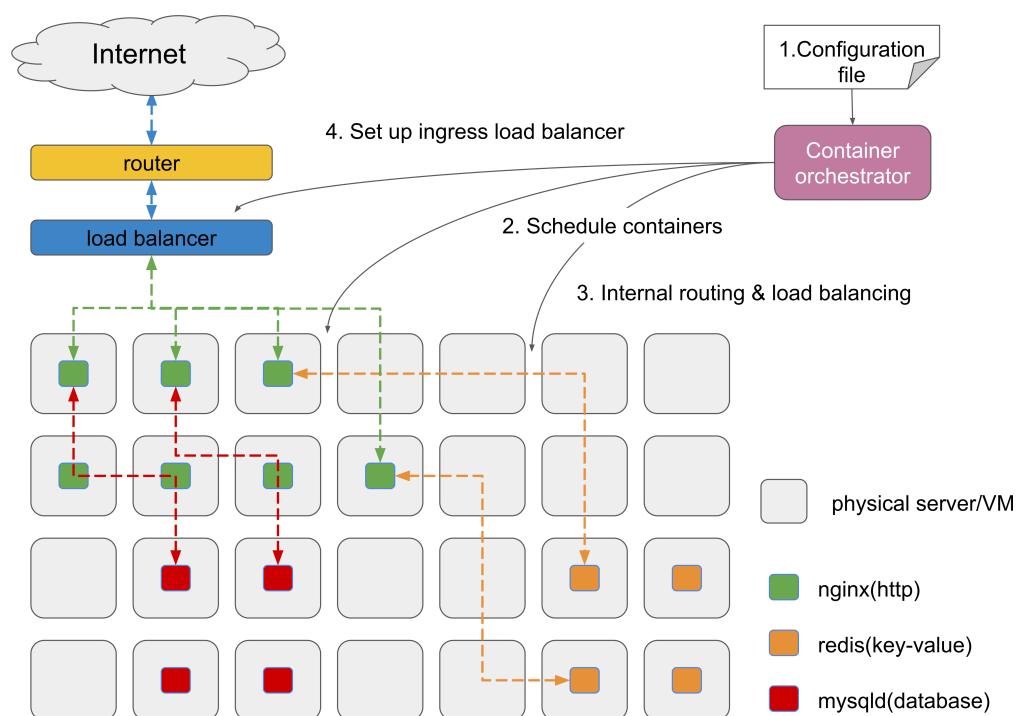


Figure 1.5: A web application cluster and container orchestrator.
A web application cluster consists of nginx(http), redis(key-value store) and mysqld(database) is depicted in this figure. Each of the component consists of a container cluster. Container orchestrator receive configuration file(1), schdule containers(2), set up ingress routing using load balancer(3) and set up internal routing(4).

A container orchestrator (also called container cluster management system) is a tool to simplify the management of a cluster of containers that are launched on

¹The author uses the word 'container orchestrator' and 'container management system' for the same meaning and uses them interchangably

multiple servers. Figure 1.5 shows the important features for the container orchestrator for web applications; 1) **Configuration file:** Orchestrator should manage container clusters based on a configuration file. The configuration file must be able to describe container cluster internals and relationships between interdependent container clusters. 2) **Scheduling:** Depending on the configuration file, the orchestrator must be able to pick servers and launch containers on them. Orchestrator must also maintain the state described in configuration files, for example, the orchestrator may need to maintain the number of running containers. 3) **Ingress routing:** The orchestrator must be able to set up routes for incoming traffic from the internet to multiple containers in a redundant and scalable manner. 4) **Internal routing:** If the web application consists of multiple interdependent container clusters, the orchestrator must be able to set up routes between them in a redundant and scalable manner.

In a configuration file a user can describe how a container cluster should be configured, and also can describe relationships between different clusters. As a result, a user can launch a web application that consists of interdependent container clusters just by supplying the configuration file to the orchestrator. For example, a web cluster in Figure 1.5 consists of three different functionalities, namely http server, key-value store, and database. Each of those consists of a container cluster. In the configuration file, the relationships between http server cluster, key-value store cluster, and database cluster are specified. The configuration file also contains how each cluster should be configured, including the number of the containers and resources assigned to them. Users only need to feed the configuration file to the orchestrator to launch the web application.

Thanks to these features, an orchestrator can be viewed as if it is an Operating System for a server farm in a data center, which not only schedules and launches containers on the server farm but also routes the traffic to the appropriate containers. By using orchestrators, a user can start a complex web application that consists of multiple interdependent container clusters, on multiple servers in a data center, as easily as starting a single process on a single computer. As a result, a user can also easily migrate their web applications at his or her convenience. And migrated web applications are guaranteed to behave exactly the same manner, not only because the same program binary and libraries are used in container, but also because container orchestrators hide difference among the base infrastructures.

Several container orchestrators are available, including Kubernetes, Docker swarm and Mesos/Marathon. Each of the container orchestrators varies in target applications, and thus has the strength and weaknesses.

Kubernetes Kubernetes[5] is an open source container orchestrator, originally developed at Google based on their experience of production container orchestrator, Borg[43]. Since Google runs many of large scale web applications, Kubernetes are considered to be best suited to run web applications.

Docker swarm Docker Swarm is a container orchestrator built in Docker daemon itself. Users can execute regular Docker commands, which are then executed by a swarm manager. The swarm manager is responsible for controlling the deployment and the life cycle of containers.

Mesos/Marathon Mesos[20] is a common resource sharing layer for different type of applications like Hadoop, MPI jobs, and Spark in a Data Center. By using Mesos user does not need to have dedicated physical server cluster for each applications. Marathon is a framework which uses Mesos in order to orchestrate Docker containers. Because of the broader scope of applications, an out of box Mesos might not be particularly suited for web applications.

	Kubernetes	Docker Swarm	Mesos Marathon
Config file	Yaml	Yaml	Json
Scheduling	Yes	Yes	Yes
Ingress routing	Static Cloud load balancer*	Static	Static
Internal routing	iptables DNAT	ipvs	haproxy

Table 1.1: Container orchestrator comparison.

Important aspects of features as web application infrastructures are compared.

*Support for Cloud load balancer is only available in limited infrastructures including GCP, AWS, Azure and Openstack.

Table 1.1 compares these orchestrators based on necessary features as an infrastructure for web applications. Although all of these orchestrators mostly satisfy the requirements, they rely on static routing for ingress traffic. Only Kubernetes

has the functionality to manage cloud load balancer so that ingress traffic from the Internet is routed to containers in a redundant and scalable manner, nevertheless, this functionality is applicable only for a few cloud environments.

To the best knowledge of the author, none of the existing container orchestrators has full support for the redundant and scalable ingress routing feature. The author believes this is an open and important topic for research and development, and therefore intends to pursue it.

1.3 Portable software load balancer

1.3.1 Load balancer for container clusters

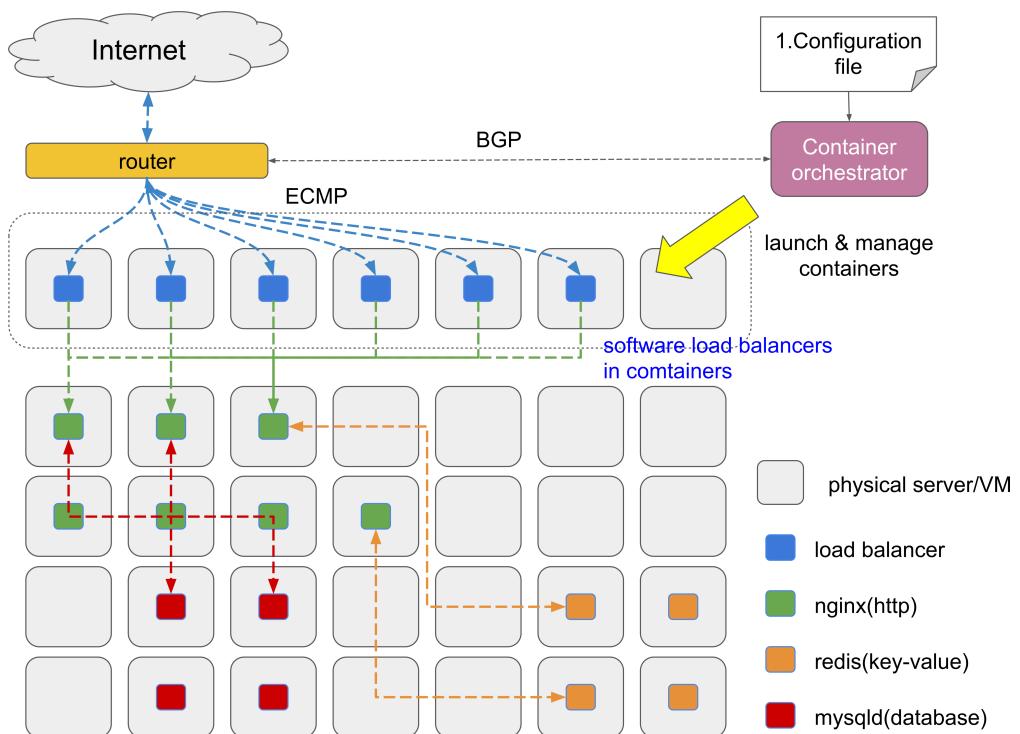


Figure 1.6: Load balancer for container clusters.
In order to distribute the traffic, container orchestrator launches a cluster of software load balancer containers. The container orchestrator also communicates with the upstream router through BGP protocol and the router set up an ECMP routing rule in the routing table.

The purpose of this research is to investigate a generic way to route the traffic into container clusters in a redundant and scalable manner and thereby to facilitate web application migrations. In order to bring this into reality, the author proposes a cluster of containerized software load balancers.

Figure 1.6 shows schematic diagram of an example architecture for such load balancers. A web application that consists of nginx, redis, and mysqld, each being a cluster of containers, is running in the server farm. There is also a cluster of software load balancer containers, running in the same server farm. All of the containers are deployed and managed by the container orchestrator. The orchestrator also communicates with the upstream router using Border Gateway Protocol(BGP), so that the ingress traffic is forwarded according to Equal Cost Multi Path(ECMP) routing table to the load balancer containers in a redundant and scalable manner.

Container orchestrators are good at managing a cluster of containers, and they can scale containers, i.e. change the number of containers depending on the needs. Therefore it seems to be a good idea to deploy load balancers as a cluster of containers. In addition to that, by utilizing ECMP routing, redundancy and scalability are accomplished at the same time.

The author investigates a cluster of containerized software load balancers for Kubernetes as a test case since Kubernetes seems most appropriate for web application clusters at the moment. Nevertheless, the author expects general findings of this investigation can be easily applied to the other container orchestrators as well.

1.3.2 Problems of Kubernetes

As is mentioned in the previous section, none of the existing container orchestrators provide full support for automatic set up of ingress traffic routing. In the case of Kubernetes, the problem is its partial support for external load balancers. Here the author elaborates on the situation.

Load balancers are often used to distribute high volume traffic from the Internet to thousands of web servers. They are implemented as dedicated hardware or as software on commodity hardware. For on-premise data centers, there are a variety of proprietary hardware load balancers. Major cloud providers have developed software load balancers[14, 33] dedicated for their infrastructures. Although software load

balancers for cloud infrastructure have APIs, through which Kubernetes can control the behavior, most of the proprietary hardware load balancers do not have such APIs.

Thanks to the expressive syntax of the configuration file, Kubernetes allows users to easily launch complex web applications that consist of multiple interdependent container clusters as if they were launching a single application program. It also allows users to modify the state of their container clusters, just by modifying the configuration file. Kubernetes always try to make the status of containers to match its desired state, which is written in the configuration file. As a result, Kubernetes is currently considered to be most preferable orchestrators for web applications.

As for the routing of the ingress traffic, the problem becomes evident. In environments where there are supported load balancers, namely cloud environments including Google Cloud Platform (GCP), Amazon Web Applications (AWS), or Openstack, Kubernetes can request the infrastructures to automatically set up the load balancers upon the launch of a web application. The cloud load balancers will distribute ingress traffic to every node(physical servers or VMs) that might host containers. Once the traffic reaches the nodes, Kubernetes nicely route them to containers using iptables Destination Network Address Translation(DNAT) based internal load balancer.

However, this scheme does not work in many environments, since there are many load balancers that are not supported by Kubernetes. In such cases Kubernetes expects users to manually set up a route for the ingress traffic, which generally lacks redundancy and scalabilities. Kubernetes fails to provide uniformity that is essential for easy deployment and migration of complex web applications.

Other container orchestrators, e.g. Docker swarm or Mesos/Marathon, do not even have partial support for load balancers and expect users to manually set up the route for ingress traffic. Therefore this is a generic problem that current container cluster orchestrators possess.

1.3.3 Proposed solution

In order to alleviate this problem, the author proposes a portable and scalable software load balancer that can be used in any environment where there is no load balancer supported by container orchestrators, as is shown in the Figure 1.6. By using a proposed load balancer, users no longer need to manually adjust their services to the base

infrastructures.

As a proof of concept the author implements the proposed software load balancer that works well with Kubernetes using following technologies; 1) To make the load balancer runnable in any environment, Linux kernel's Internet Protocol Virtual Server (ipvs)[45] is containerized using Docker[30]. 2) To make the load balancer redundant and scalable, the author makes it capable of updating the routing table of upstream router with Equal Cost Multi-Path(ECMP) routes[17] using Border Gateway Protocol(BGP), which is a standard routing protocol. 3) The author also extends the research into implementing the novel load balancer using eXpress Data Plane(XDP) technology[4] to enhance the performance level to meet the need for 10Gbps network speed.

1.3.4 Contribution

Contributions of this paper can be summarized as follows: 1) The author addresses the problem of ingress traffic routing that is generic to container orchestrators and proposes software load balancer suitable for container environment. This is one of the most important problems for container orchestrators because without solving this problem migration of a web application will never be easy. 2) The author builds a proof of concept load balancers using OSS, which means that anyone can test drive the proposed load balancers and use them in production for free. 3) The author provides quantitative performance analysis to see the feasibility of proposed load balancer architecture in the 1Gbps network. 4) The author clarifies the remaining problems for future improvement in the 10Gbps network and explores other technology to be used in faster networks.

The outcome of this study will benefit users who want to deploy their web applications 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 application 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. Chapter 6 and Chapter 2 provides related works and the background information necessary to understand following chapters.

Chapter 3 provides discussion of load balancer architecture suitable for container clusters. First the author discusses problems of conventions architecture in Section 3.1.1. Then the author proposes a portable software load balancer in container in Section 3.1.2. After that the author discusses redundancy architecture using ECMP in Section 3.1.3. Chapter 3 also presents implementation of the proof of the concept system for the proposed load balancer architecture in detail. First overall architecure is explained in Section 3.2.1. Then ipvs containerization is explained in detail in Section 3.2.2. Finally implementation of BGP software container is explained in Section 3.2.3.

Chapter ?? discusses portability and performance levels of the proposed load balancer in 1 Gbps network environment. Chapter ?? discusses the redundancy and scalability of the proposed load balancers. Chapter 5 present the performance levels of the proposed load balancer in 10 Gbps network environment and discuss the method to improve the performance of a software load balancer. Chapter ?? discusses the limitation and the future work of this study, which is followed by a conclusion of this work in Chapter 7.1.

2

Background

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

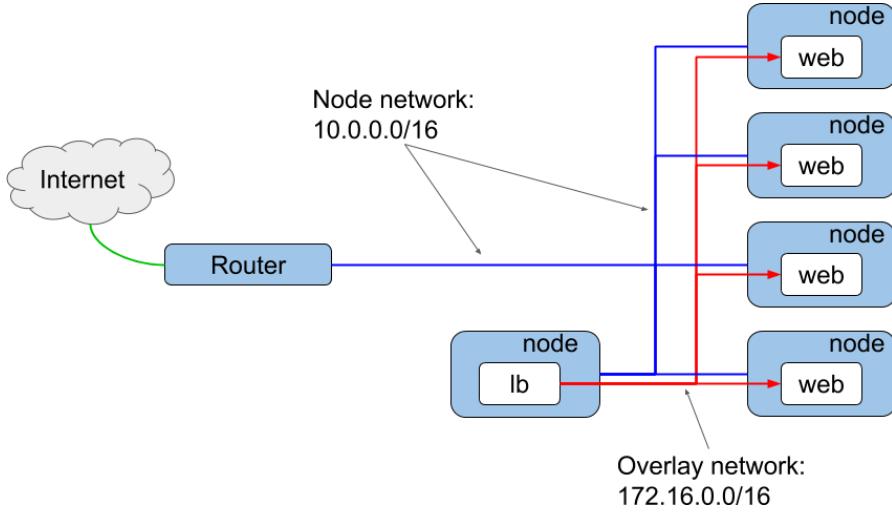


Figure 2.1: The network architecture of an exemplified container cluster system.

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.

2.1 Container Network

2.1.1 bridge+veth

2.1.2 macvlan, ipvlan

2.1.3 hostnetwork

2.2 Overlay Network

2.2.1 General concept

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. 2.1 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. 2.1 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.

2.2.2 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[9].

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,

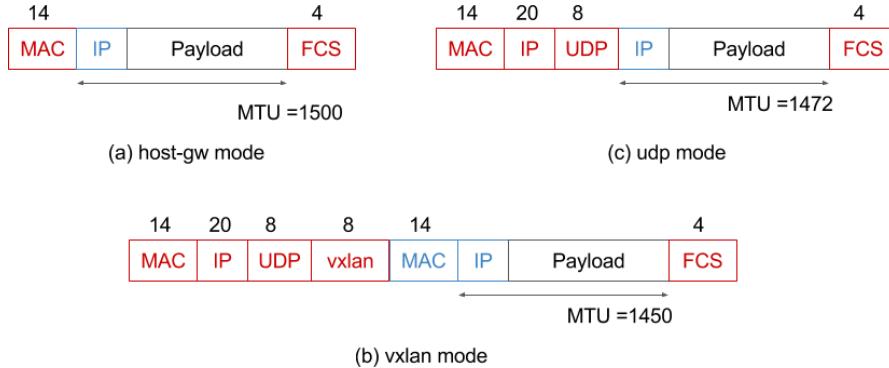


Figure 2.2: Frame diagram

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 2.2 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 is shown in Chapter ???. 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

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 tunneling using vxlan or udp is needed.

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.

2.2.3 Calico

[Filled in later]

2.3 Multicore Packet Processing

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)[42] is a technology to distribute handling of the interrupt from NIC queues to multiple CPU cores. Subsequently, Receive Packet Steering

```

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 2.3: RX/TX queues of the hardware

(RPS)[42] 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 2.3 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 [42], 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
```

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.

2.4 Linux vritual server

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[45]. 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.

¹The term, *real servers* refers to worker servers that will respond to incoming traffic, in the original literature[45]. We will also use this term in the similar way.

2.5 Summary

3

Architecture and Implementation

This chapter provides a discussion of load balancer architecture suitable for container clusters. First the author discusses problems of conventions architecture in Section 3.1.1. Then the author proposes a portable software load balancer in container in Section 3.1.2. After that the author discusses redundancy architecture using ECMP in Section 3.1.3.

This chapter also presents implementation of the proof of the concept system for the proposed load balancer architecture in detail. First overall architecure is explained in Section 3.2.1. Then ipvs containerization is explained in detail in Section 3.2.2. Finally implementation of BGP software container is explained in Section 3.2.3.

3.1 Architecture

In this section the author discusses the architecture of a portable load balancer for container clusters.

3.1.1 Problem of Conventional Architecture

The problem of Kubernetes is its partial support for the ingress traffic routing. Figure 3.1a 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, kubelet and proxy are deployed. The kubelet daemon will run *pods*, depending on the PodSpec (pod specification) information obtained from the apiserver on the master. The proxy daemon will set up iptables DNAT rules that function as the internal load balancer. A *pod* is a group of containers that share the same network namespace and cgroup, and is the basic execution unit in a Kubernetes cluster.

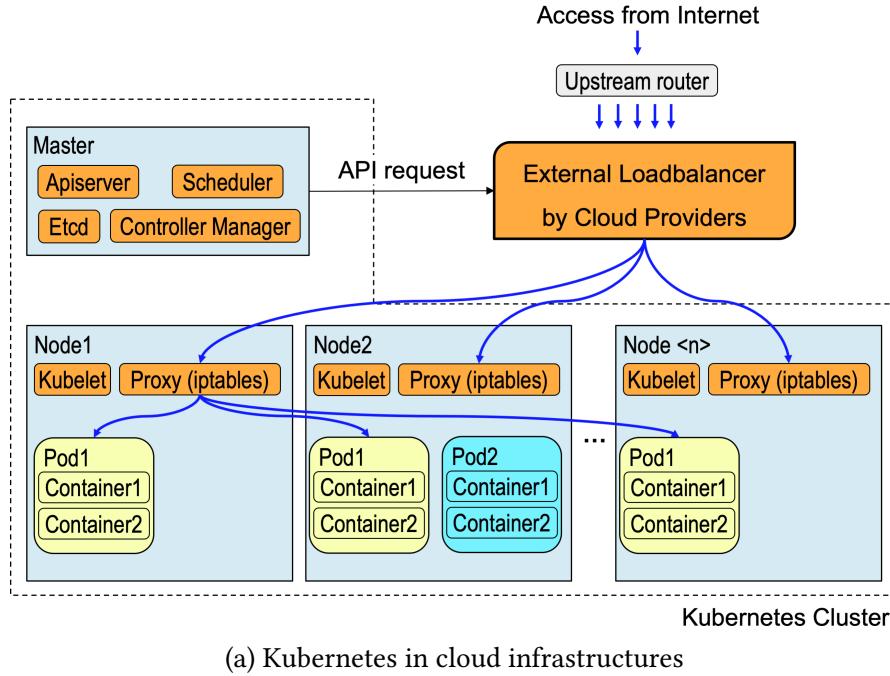
When a service is created, the master schedules where to run *pods*, and kubelets on the nodes launch them accordingly. At the same time, the master sends out requests to cloud provider's API endpoints, asking them to set up external cloud load balancers that distribute ingress traffic to every node in the Kubernetes cluster. The proxy daemon on the nodes also setup iptables DNAT[27] rules. The Ingress traffic will then be evenly distributed by the cloud load balancer to all the existing 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, and hence it would be very difficult to find out which node is malfunctioning. 3) The ingress traffic is distributed to all the existing nodes in the Kubernetes cluster. Suppose there are 1,000 nodes and one of web applications only uses 10 nodes, it seems inefficient and rather complicated to distribute the ingress traffic to all the 1,000 nodes.

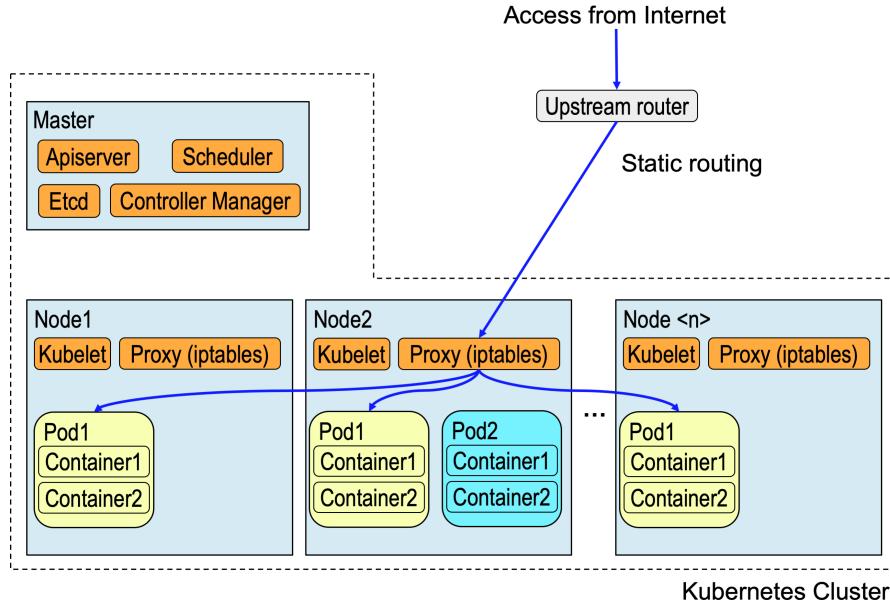
Regarding the first problem, if there is no load balancer that is supported by Kubernetes, users must manually set up the static route on the upstream router,

every time they launch the web application clusters, as is shown in Figure 3.1b. The traffic will be routed to a node then distributed by the DNAT rules on the node to the designated *pods*. In the cases where the upstream router is administered by a data center company, users must always negotiate with them about adding a route to their new application cluster. This approach significantly lacks simplicity, and degrades the portability of container clusters. Furthermore, a static route usually lacks redundancy and scalability.

In short, while Kubernetes is effective in major cloud providers, it fails to provide portability for container clusters in environments where there is no supported load balancer. And the routes incoming traffic follow are very complex and inefficient. In order to address these problems, the author proposes a containerized software load balancer that is deployable in any environment even if there are no external load balancers.



(a) Kubernetes in cloud infrastructures



(b) Kubernetes in on-premise data centers

Figure 3.1: Conventional architecture of Kubernetes clusters in cloud infrastructure and on-premise data center. (a) In supported infrastructures, e.g., major cloud providers, Kubernetes automatically send a request to set up a load balancer, when a container cluster is launched. The load balancer distributes ingress traffic toward an IP address for a service(the author calls this address, service IP) to all the existing nodes. (b) In unsupported infrastructures, e.g., on-premise data centers, web application providers have to manually set up a route to one of the nodes. Packets that reached any of the nodes, having the service IP as the destination IP address, will be distributed to appropriate pods by the iptables DNAT based internal load balancer.

3.1.2 Load balancer in container

The author proposes a load balancer architecture, where a cluster of load balancers are deployed as containers. Figure 3.2 shows the proposed load balancer architecture for Kubernetes, which has the following characteristics; 1) Each load balancer itself is run as a *pod* by Kubernetes. 2) Load balancing rules are dynamically updated based on the information about running *pods*. 3) There exist multiple load balancers for redundancy and scalability. 4) The routing table in the upstream router is updated dynamically using standard network protocol.

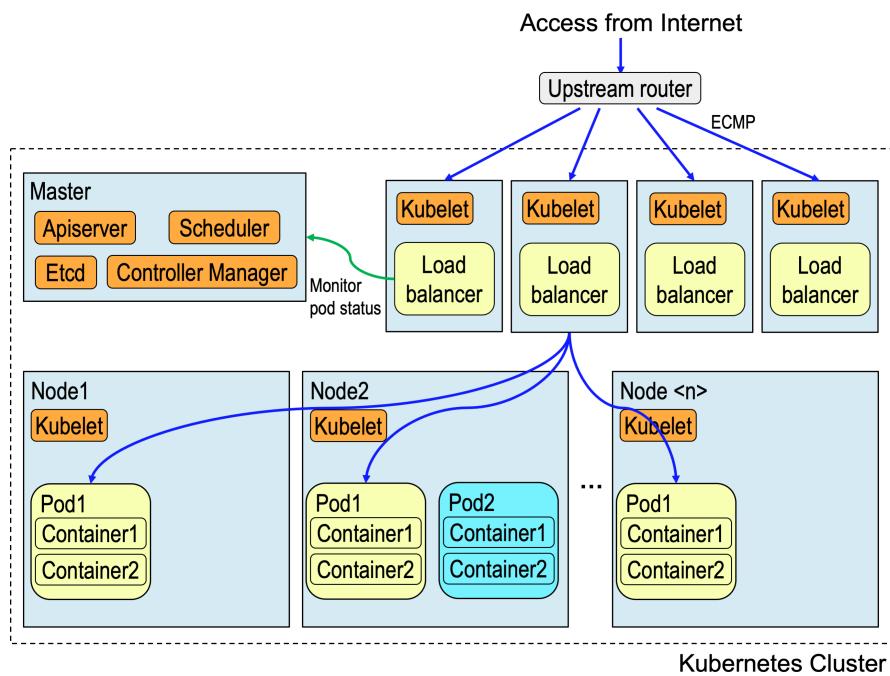


Figure 3.2: Kubernetes cluster with proposed load balancer.

Each load balancer itself is run as a *pod* in the Kubernetes cluster. Load balancing rules are dynamically updated based on the information about running *pods*. There exist multiple load balancers for redundancy and scalability. The routing table in the upstream router is updated dynamically using standard network protocol.

The proposed load balancer can resolve the conventional architecture problems. 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. The incoming traffic is directly distributed to designated *pods* by the load balancer. It makes the administration, e.g. finding malfunctions, easier.

Furthermore, the proposed load balancer has other benefits. Since a software load balancer in a container can run on any Linux system, it can share the server pool with web containers. Users can utilize existing servers rather than buying dedicated hardware.

Because a cluster of load balancer containers is controlled by Kubernetes, it becomes redundant and scalable. Kubernetes always tries to maintain the number of load balancer containers at the number specified by the user. If a single container fails, Kubernetes schedule and launch another one on a different node, which provides the resilience to failures. When there is a huge spike in the traffic, it can quickly scale the size of the cluster depending on the demand.

The routes to the load balancers are automatically updated through the standard protocol, BGP. Therefore users do not need to manually add the route every time new load balancer container is launched, as is the case in the conventional architecture.

3.1.3 Redundancy with ECMP

While containerizing ipvs makes it runnable in any environment, it is essential to discuss how to route the ingress traffic to the ipvs container. The author proposes redundant architecture using ECMP with BGP for proposed load balancer containers.

Fig. 3.3 shows a schematic diagram to explain redundancy architecture with ECMP for the proposed load balancer. The ECMP is a functionality a router supports, where the router has multiple next hops with equal priority(cost) to a destination. And the router generally distributes the traffic to the multiple next hops depending on the hash of five-tuples(source IP, destination IP, source port, destination port, protocol) of the flow. The multiple next hops and their cost are often populated using the BGP protocol. The notable benefit of the ECMP setup is its scalability. 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, after all, limited by performance levels of the router. However, in practice, there are a lot of cases where this architecture is beneficial. For example, if a

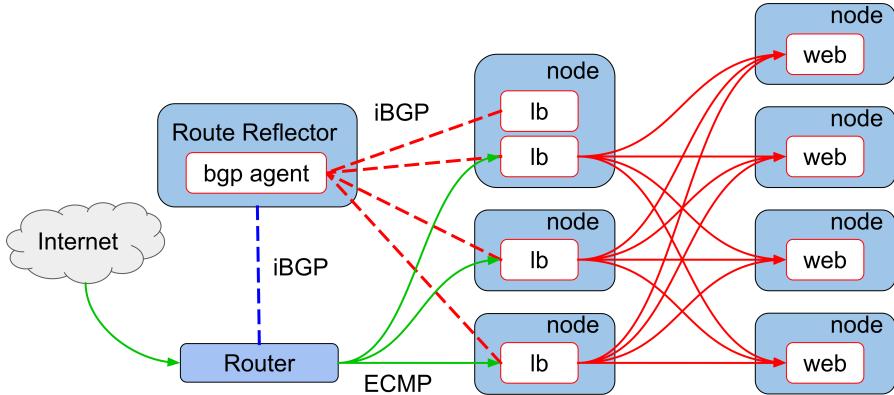


Figure 3.3: The proposed architecture of load balancer redundancy with ECMP
The traffic from the internet is distributed by the upstream router to multiple of load balancer(lb) pods using hash-based ECMP(the solid green line), after which distributed by the lb pods to web pods using Linux kernel's ipvs(the solid red line). The route toward a service IP is advertised to the route reflector(the dotted red line), after which advertised to the upstream router(the blue dotted line) using iBGP.

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.

In the proposed redundant architecture, there exists a node with the knowledge of the overlay network as a route reflector. A route reflector is a network component for BGP to reduce the number of peerings by aggregating the routing information[40]. In the proposed architecture the author uses it as a delegater for load balancer containers towards the upstream router.

The route reflector exists for a practical reason, i.e, to deal with the complexity due to the overlay network. Since the upstream router normally has no knowledge of the overlay network and IP addresses used inside the Kubernetes clusters, a container must rely on SNAT on the node to communicate with the router. The SNAT caused a problem when the author tried a set up without the route reflector, to co-host multiple load balancer containers for different services on a single node. Because of the SNAT, the source IP addresses of multiple connections were translated into a single IP address possessed by the node. The BGP agent on the router was confused by these connections

and could not properly set up ECMP routes for separate services. This was due to the fact that the BGP agent used in the experiment used only the source IP address of the connection to distinguish the BGP peer.

In addition to that, the route reflector brings another benefit. The upstream router does not need to accept BGP sessions from containers with random IP addresses, but only from the route reflector with well known fixed IP address. This is preferable in terms of security especially when a different organization administers 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 in the BGP peer definition.

Since a standard Linux box is used for the route reflector, it can be configured as we like; a) It can be configured to belong to the overlay network so that multiple BGP sessions from containers on a single node can be properly distinguished. b) One can select a BGP agent that supports dynamic neighbor (or dynamic peer), where he only needs to define the IP range as a peer group and does away with specifying every possible IP that load balancers may use. Although not shown in the Fig. 3.3, it is possible to have another route reflector for redundancy purpose.

3.2 Implementation

In this section the author discusses the implementation of the experimental system to prove the concept of our proposed load balancers with ECMP redundancy in detail.

3.2.1 Experimental system architecture

Fig. 3.4 shows the schematic diagram of proof of concept container cluster system with the proposed software load balancers. Each load balancer pod consists of an exabgp container and an ipvs container. The ipvs container is responsible for distributing the traffic toward the service IP to web server(nginx) pods. The IP address for nginx pods

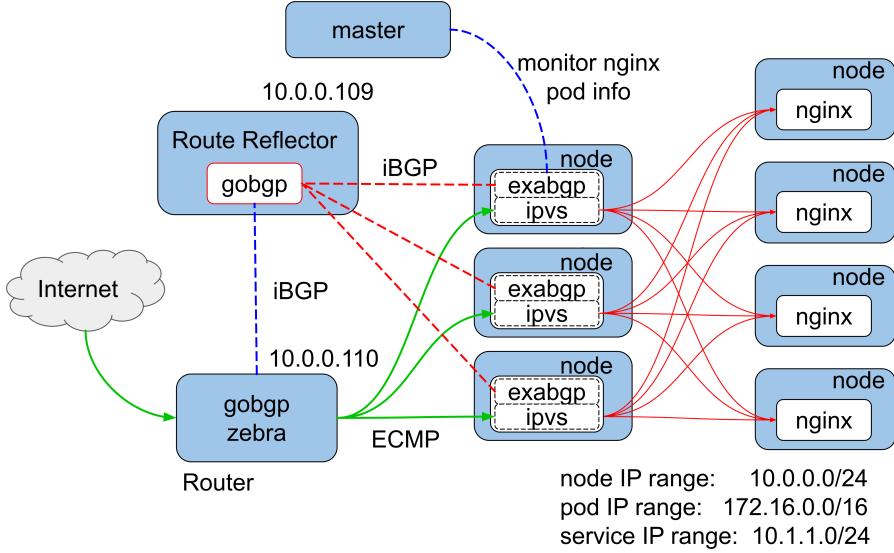


Figure 3.4: An experimental container cluster with proposed redundant software balancers

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. For the green lines, a service IP address is used. The red lines use the IP addresses of the overlay network. The blue line uses the IP addresses of the node network.

and load balancer pods are dynamically assigned upon launch of themselves from 172.16.0.0/16 address range. The ipvs container monitors the availability of web server pods by consulting apiserver on the master node and manages the load balancing rule appropriately. The exabgp container is responsible for advertising the route toward the service IP to the route reflector. The route reflector aggregates the routing information advertised by load balancer pods and advertise them to the upstream router. The upstream router updates its routing table according to the advertisement.

All the nodes and route reflector are configured using Debian 9.5 with self compiled linux-4.16.12 kernel. The author also used conventional Linux box as an upstream router for testing purpose, using the same OS as the nodes and route reflector. The version of Linux kernel needed to be 4.12 or later to support hash based ECMP routing table. The author also needed to enable kernel config option CONFIG_IP_ROUTE_MULTIPATH[23] when compiling, and set the kernel parameter fib_multipath_hash_policy=1 at run time. Although in the actual production environ-

ment, proprietary hardware router with the highest throughput is usually deployed, one can still test some of the advanced features by using a Linux box as the router.

	gobgp	exabgp	bird
Static route advertisement without injector*	No	Yes	No
add-path support**	Yes	No	Yes
FIB manipulation	Yes(through zebra)	No	Yes(Native)
Use case	Router/Route reflector	Load balancer	Not used***

Table 3.1: Comparison of open source BGP agents

Open source BGP agents are compared in terms of the features required in the proposed systems.

* The injector is a program that injects static route, other than the BGP daemon itself.

** The add-path is the feature to support multipath advertisement.

*** Configuration of bird was more complex than other agents.

Table 3.1 compares open source BGP agents in terms of required features for the proposed architecture. Each load balancer *pod* needs to advertise a static route for the service IP to itself. For that purpose, exabgp is preferable due to its simplicity. As for the route reflector, add-path[44] feature is needed for multi-path advertisement, which is supported by gobgp. For the upstream router Forwarding Information Base(FIB) manipulation[16] feature is needed, which is supported by gobgp. As a result, exabgp is used for the load balancer pods, and gobgp is used for the route reflector and the upstream router.

The other features needed for the route reflector are a dynamic-neighbor feature to describe peer group as a range of IP address, and overlay network feature on the Linux box. The configurations for the upstream router is summarised in B.3. The configurations for the route reflector is summarised in B.2.

3.2.2 Ipv4 container

The proposed load balancer needs to dynamically reconfigure the ipvs balancing rules whenever *pods* are created or deleted. Fig. ?? is a schematic diagram of ipvs container to show the dynamic reconfiguration of the ipvs rules. Two daemon programs, controller and keepalived, running in the container are illustrated. The keepalived manages Linux

kernel's ipvs rules depending on the `ipvs.conf` configuration file. It can also periodically health check the liveness of a *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 that *real server* from the ipvs rules immediately. The interval of the health check is typically 1 to several seconds and is arbitrarily determined by users.

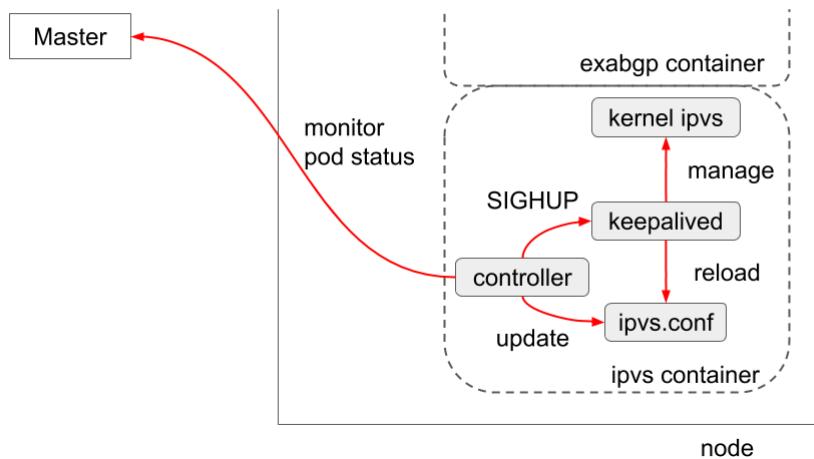


Figure 3.5: Implementation of ipvs container.

The controller checks the pod status every second, by consulting the master node. Upon a change of the status, the controller updates the `ipvs.conf` and sends `SIGHUP` to `keepalived`. The `keepalived` reloads the `ipvs.conf` and updates the load balancing rules in the kernel correctly.

Every second, the controller monitors information concerning the running *pods* of a web application in the Kubernetes cluster by consulting the `apiserver` running in the master through its API. Whenever *pods* are created or deleted, the controller notices the change and automatically regenerate an appropriate `ipvs.conf` and issue `SIGHUP` to `keepalived` within a second. Then, `keepalived` will reload the `ipvs.conf`, and modify the kernel's ipvs rules correctly depending on the result of the health check.

When a pod is terminated, existing connections are reset by the node kernel. The `SYN` packets sent to a pod after termination, but before the ipvs rule update, will be answered with `ICMP unreachable` by the node. In these cases, the client sees

connection errors. In order to avoid the connection errors to be seen by a human, HTTP client programs are required to re-initiate the connection. However, since the load balancer rule update is within a second, these errors can be regarded as the tolerable rare exceptions even without such re-initiations.

The actual controller[25] is implemented using the Kubernetes ingress controller[2] framework. By importing existing Golang package, “k8s.io/ingress/core/pkg/ingress”, the author could simplify the implementation, e.g. 120 lines of code. Keepalived and the controller are placed in the docker image of ipvs container. The ipvs is the kernel function and namespace separation for container has already been supported in the recent Linux kernel.

Configurations for capabilities were needed when deploying the ipvs container: 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, the author also needed to mount the “/lib/module” of the node’s file system on the container’s file system.

Figure 3.6 and Figure 3.7 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[3] is distributed to 172.16.21.2:80 and 172.16.80.2:80 using the masquerade mode(Masq) and the least connection(lc)[45] balancing algorithm.

3.2.3 BGP software container

In order to implement the ECMP redundancy, the author also containerized exabgp using Docker. Figure 3.8 shows a schematic diagram of the network path realized by the exabgp container. As mentioned earlier, the author used exabgp as the BGP advertiser. The ingress traffic from the Internet is forwarded by ECMP routing table on the router to the node that hosts a load balancer pod. And then it is routed to the load balancer pod according to the set of routing rules in Figure 3.8. After that the ipvs forwards them to nginx pods. The IP address of any pod is dynamically assigned from 172.16.0.0/16 when the pod is started.

Table 3.2 summarises some key settings required in the exabgp container to route

```
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 3.6: An example of ipvs.conf

This configuration file is auto-generated by the controller. The controller periodically accesses the apiserver on the master node and constantly monitor the status of running pods.

the traffic to the ipvs container. In BGP announcements the node IP address, 10.0.0.106 is used as the next-hop for the service IPs, 10.1.1.0/24. Then on the node, in order to route the packets toward the service IPs to the ipvs container, a routing rule for 10.1.1.0/24 to the dev docker0 is created in the node net namespace. A routing rule to accept the packets toward the service IPs as local is also required in the container net namespace. A configuration for exabgp is shown in [B.1](#).

```
# 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 3.7: Example of IPVS balancing rules

This shows the load balancing rule in a load balancer container. The packet that has FWM(fwmark)=1 will be forwarded to two real servers using the least connection(lc) balancing algorithm. The fwmark is a parameter that is only available inside a socket buffer in Linux kernel. The fwmark can be put and manipulated by the iptables program, once a socket buffer for a received packet is assigned by the kernel.

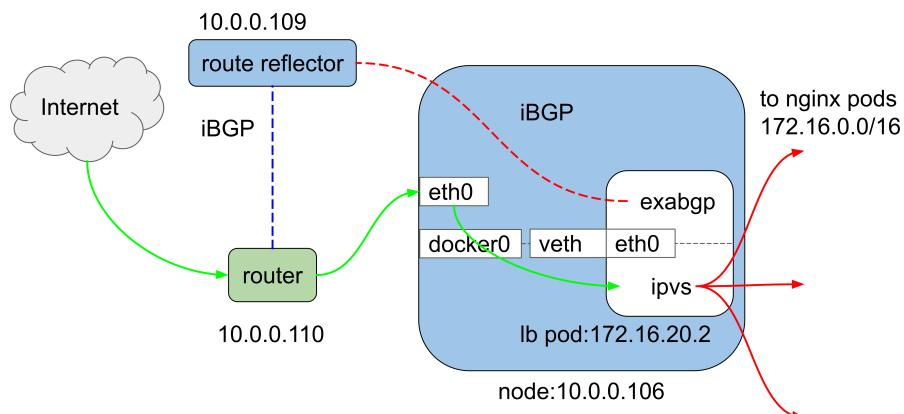


Figure 3.8: Network path by the exabgp container

The packets from Internet to service IPs, 10.1.1.0/24 are routed to the load balancer pod(green arrows) by the set of routing rules shown in Table 3.2. And then the ipvs container forwards them to nginx pods(red arrows). The IP address of any pod is dynamically assigned from 172.16.0.0/16 when the pod is started.

[BGP announcement]

route 10.1.1.0/24 next-hop 10.0.0.106 ... (1)

[Routing in node net namespace]

ip netns exec node ip route replace 10.1.1.0/24 dev docker0 ... (2)

[Accept as local]

ip route add local 10.1.1.0/24 dev eth0 ... (3)

Table 3.2: Required settings in the exabgp container

(1) The node IP address, 10.0.0.106 is used as next-hop for the service IPs, 10.1.1.0/24, in BGP announcement. (2) In order to route the packets destined toward the service IP to the container, a routing rule to the dev docker0 is created in the node net namespace. (3) A routing rule to accept the packets destined toward the service IPs, as local is also required.

3.3 Summary

In this chapter, the author provided a discussion of load balancer architecture and its implementations suitable for container clusters. First, the author discussed the problems of conventional architecture. Since Kubernetes is dependent on external load balancers provided by the cloud infrastructures, it failed to provide portability of a web application in environments where there was no supported load balancer. Furthermore, the routes that ingress traffic from the internet follow were very complex and inefficient.

In order to alleviate these problems, the author proposed a cluster of software load balancers in containers. The proposed load balancers utilized container technology and were managed by Kubernetes. As a result, it is runnable on any environment including cloud infrastructures and on-premise data centers. Furthermore, since Kubernetes manages load balancer containers, it can quickly scale the number of containers depending on the demand. The author also discussed redundant architecture using ECMP with BGP for proposed load balancer containers. By using the ECMP, the upstream router can route the ingress traffic to a cluster of load balancer containers in a redundant and scalable manner. By using BGP, which is the standard protocol, ECMP routing rules in the upstream router are automatically populated, upon the launch of load balancer containers.

As a result users, i.e., web application providers can quickly and automatically set up routes to their web application, upon its launch. This will greatly improve the portability of a web application and thereby enables migrations.

4

Performance Evaluation

In order to verify the feasibility of the proposed load balancer architecture, the author evaluated the performance of the load balancer with the following criteria; (1) Performance analysis: The author evaluated the basic characteristics of the load balancer using physical servers in the on-premise data center, and compared performance level with iptables DNAT and nginx as a load balancer. (2) Portability: The author 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. (3) Redundancy and Scalability: The author evaluated ECMP functionality by monitoring routing table updates on the router when the new load balancer is added or removed. The author also evaluated the performance level by changing the number of load balancers.

4.1 Performance analysis of proposed load balancer

Experimental conditions

Throughput measurements were carried out in order to examine the basic characteristics of the containerized ipvs load balancer in an on-premise data center. Figure 4.1 shows the schematic diagram of the experimental setup for the measurement. A benchmark program called wrk[18] were used. Multiple nginx *pods* are deployed on multiple nodes as web servers in the Kubernetes cluster. In each nginx *pod*, single nginx web server program that returns the IP address of the *pod* itself is running. The author then launched ipvs pod and nginx pod as load balancers on one of the nodes, after that, performed the throughput measurement changing the number of the nginx web server pods. On every Kubernetes node, there are iptables DNAT rules that function as an internal load balancer. The author also measured throughput for the iptables DNAT as a load balancer. The throughput is measured by sending out HTTP requests from the wrk towards a load balancer and by counting the number of responses the benchmark client received.

Table 4.1 shows an example of the command-line for the benchmark program, wrk, and the corresponding output. The command-line in Table 4.1 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 information including per-thread statistics, error counts, throughput in [Request/sec] and data rate in [Transfer/sec].

Table 4.2 shows hardware and software configuration used in the experiment. The author 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 hardware had eight physical CPU cores and a network interface card (NIC) with four rx-queues. The author 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.

The author also investigated the performance level, varying two types of network conditions: The first condition is the setting for multicore packet processing. It is known that distributing handling of interrupts from the NIC and the subsequent IP protocol processing, among multiple cores impact the network performance. To

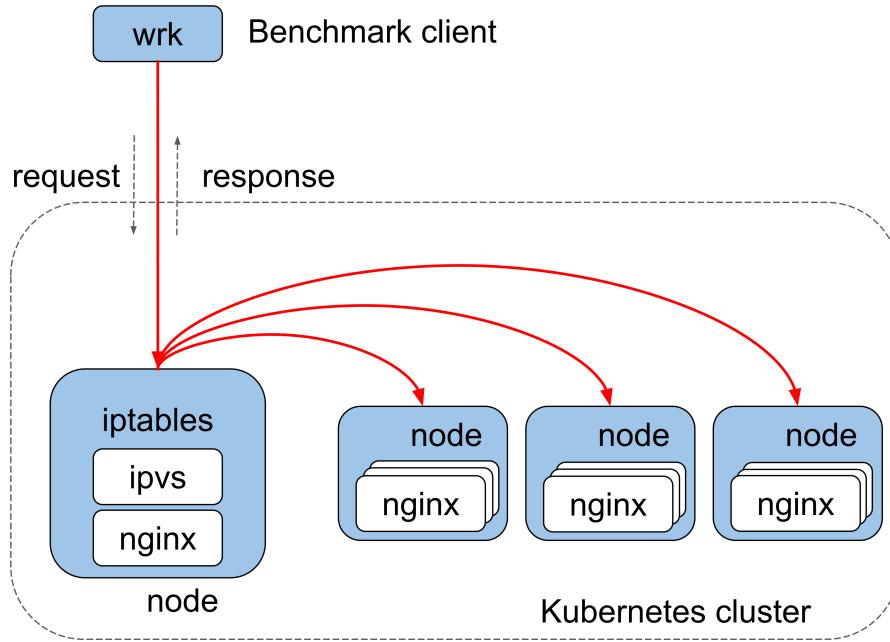


Figure 4.1: Benchmark setup. There are nodes on which nginx pods are deployed as web servers. There is another node where ipvs pod and nginx pod are deployed. Iptables DNAT rules exist on every Kubernetes node as an internal load balancer. Throughput measurements are performed against ipvs pod, nginx pod, and iptables DNAT rules. Each of the load balancers distributes the packets from the wrk to the nginx web servers. The response packets from the nginx web servers follow the same route as the request packets.

derive the best performance from load balancers, the author investigated how this setting would affect their performance levels. The second condition is an overlay network setting[39, 26]. The overlay network is often used to build the Kubernetes cluster and therefore used in the experiment. Flannel[11] is one of the popular overlay network technologies. The author compared the performance levels of three backends settings[9] of flannel, i.e., operating modes, to find the best setting.

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

(a) Command line

```
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
```

(b) Output example

Table 4.1: Benchmark command line and output example. (a) This command line 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. (b) The output example shows information including per-thread statistics, error counts, throughput in [Request/sec] and data rate in [Transfer/sec]. The throughput is 164717.63 [Request/sec] in this example.

[Hardware Specification]

CPU: Xeon E5-2450 2.10GHz (with 8 core, Hyper Threading)

Memory: 32GB

NIC: Broadcom BCM5720 with 4 rx-queues, 1 Gbps
(Node x 6, Load Balancer 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 4.2: Hardware and software specifications. The hardware had eight physical CPU cores and a network interface card (NIC) with four rx-queues.

Effect of multicore processing

Figure 4.2 shows the result of the throughput measurement for the ipvs load balancer with different multicore processing settings. Since hardware used in the experiment has a NIC with four rx-queues, and a CPU with eight cores, the setting “(RSS, RPS) = (on, off)” uses 4 cores and “(RSS, RPS) = (off, on)” uses 8 cores for packet processing. For the setting “(RSS, RPS) = (off, off)”, only one core is used for the packet processing.

There is a general trend in the throughput result, where the throughput linearly increases as the number of nginx *pods* increases and then it eventually saturates. For example, if we look at the red line in the Figure 4.2, the throughput increases almost linearly with the increasing number of the nginx pods(web servers) from 1 to around 14, and then saturated. The increase indicates that the load balancer functions properly, as the load balancer increased throughput by distributing HTTP requests to multiple of the web servers. The saturated throughput indicates the maximum performance level of the load balancer, which could be determined either by network bandwidth between the benchmark client machine and the load balancer node, or CPU performance levels of these machines. The maximum performance levels are dependent on the number of cores used for packet processing. Throughput is highest for the setting with eight cores (rps is on), followed by four cores (rss is on), then single core (none). This indicates that the more cores are used, the better the throughput is.

Figure 4.3 shows the result of the throughput measurement for iptables DNAT as a load balancer, with different multicore processing settings. As is the case for the ipvs result, there is a general trend where the throughput increases as the number of nginx *pods* increases and then it eventually saturates. Also, throughput is highest for the setting with eight cores (rps is on), followed by four cores (rss is on), then single core (none). The saturated performance levels of iptables DNAT and ipvs are the same for the condition with eight cores (rss is on). The saturated performance levels of iptables DNAT are higher than those of ipvs, for the conditions with four cores (rss is on) and single core (none).

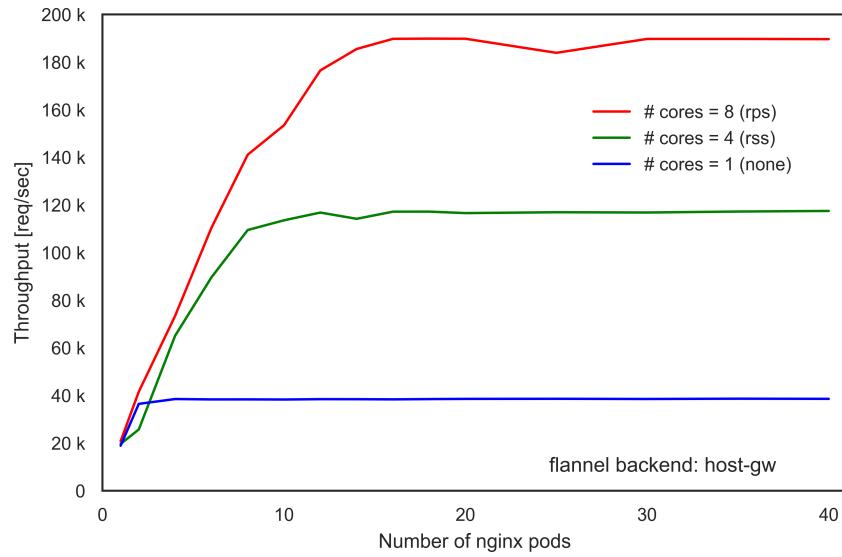


Figure 4.2: Effect of multicore packet processing on ipvs throughput. Throughput linearly increases as the number of nginx *pods* increases and then it eventually saturates. The throughput is highest for the setting with eight cores (rps is on), followed by four cores (rss is on), then single core (none).

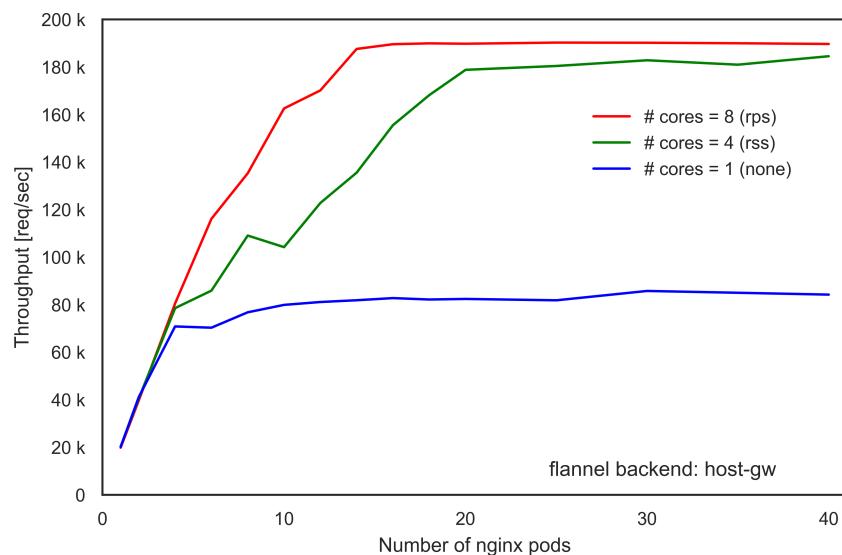


Figure 4.3: Effect of multicore packet processing iptables DNAT throughput. Throughput linearly increases as the number of nginx *pods* increases and then it eventually saturates. The throughput is highest for the setting with eight cores (rps is on), followed by four cores (rss is on), then single core (none).

Effect of overlay network

Figure 4.4 shows the ipvs throughput results for different overlay network settings. The author used the flannel for the overlay network. Flannel has three backend modes, host-gw, vxlan and udp, and the throughput for each setting are compared.

Except for the udp backend mode case, a general trend can be clearly seen, i.e., the throughput linearly increases as the number of nginx *pod* increases, and then it eventually saturates. Among the flannel backend mode types, the host-gw mode where no encapsulation is conducted shows the highest performance level, followed by the vxlan mode where the Linux kernel encapsulates the Ethernet frame. The udp mode where flanneld itself encapsulates the IP packet shows significantly lower performances levels.

Figure 4.5 shows throughput results of the iptables DNAT as a load balancer for different overlay network settings. The same characteristics can be seen for the iptables DNAT, although the performance level of the iptables DNAT for udp mode is slightly better than that of ipvs.

As is shown here, overlay network settings greatly affect the performance level. The author considers the host-gw mode is the best, the vxlan tunnel the second best and the udp tunnel mode unusable. In environments where containers need to communicate with each other via a gateway that has no knowledge of overlay network, the host-gw mode cannot be used, which is often the case in cloud environments. The author used host-gw mode for the rest of the experiments conducted in on-premise data centers and vxlan mode for the experiments conducted in cloud environments.

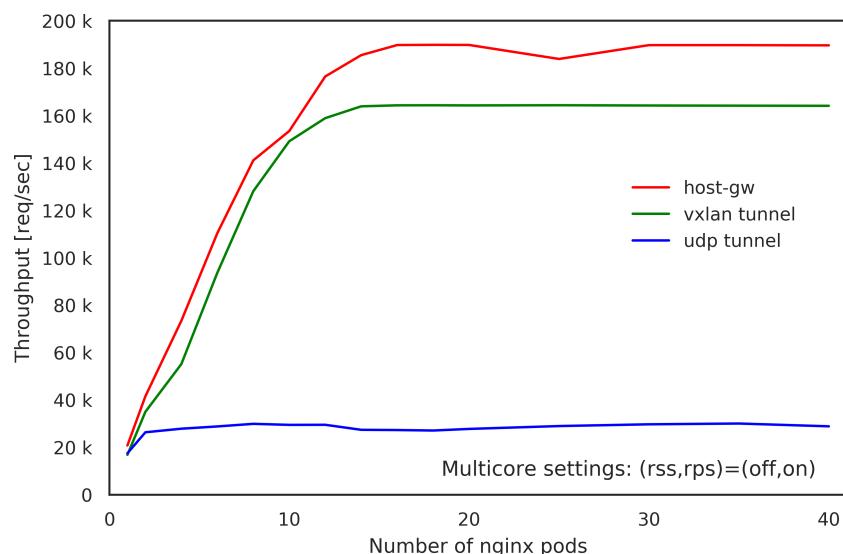


Figure 4.4: Effect of flannel backend modes on ipvs throughput. The host-gw mode shows the highest performance level, followed by the vxlan mode. The udp mode shows significantly lower performances levels.

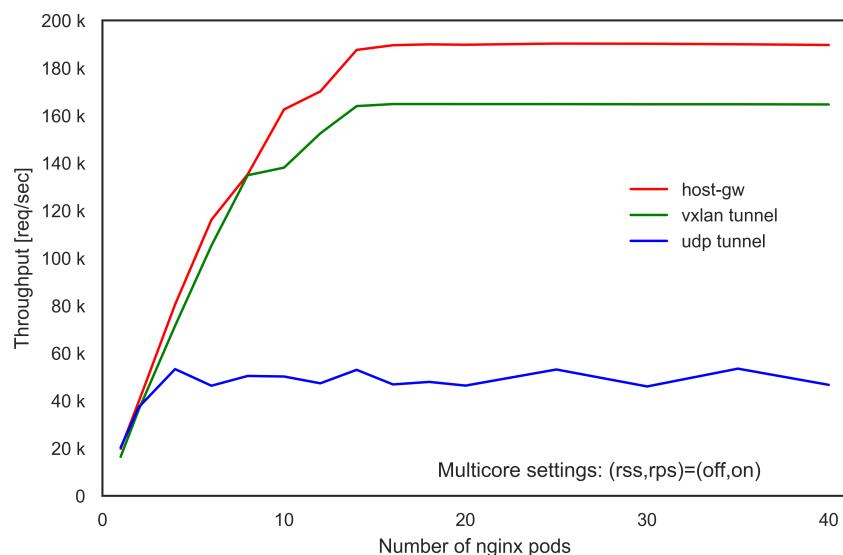


Figure 4.5: Effect of flannel backend modes on iptables DNAT throughput. the host-gw mode shows the highest performance level, followed by the vxlan mode. The udp mode shows significantly lower performances levels.

Comparison of different load balancer

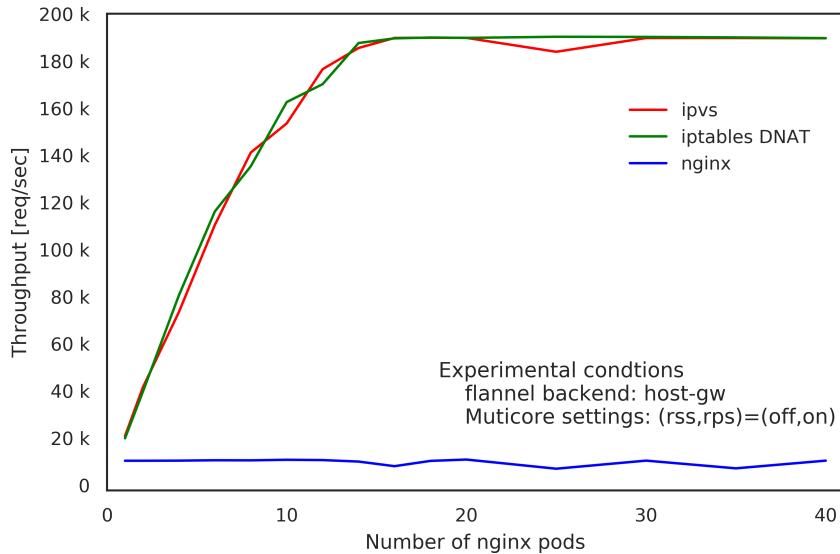


Figure 4.6: Throughput of ipvs, iptables DNAT and nginx. The performance levels of ipvs and iptables DNAT are almost the same. The nginx as a load balancer does not perform well in the experiment.

Figure 4.6 presents the throughput results of the different load balancers. The performance levels of ipvs, iptables DNAT and nginx as the load balancers are compared. The throughput of the ipvs and iptables DNAT increases almost linearly with the increasing number of the nginx pods(web servers) from 1 to around 14, and then saturated. The proposed ipvs load balancer exhibits almost equivalent performance levels as the iptables DNAT as a load balancer.

The saturated throughput indicates the maximum performance level of the load balancer, which could be determined either by network bandwidth between the benchmark client machine and the load balancer node, or CPU performance levels of these machines. In this specific experiment, the performance level was limited by the 1 Gbps bandwidth of the network¹, which is revealed by packet level analysis using tcpdump[41]. On average the data size of each request and the corresponding response was about 636 [byte/req] in total, including TCP/IP headers, Ethernet header, and

¹ All of the nodes use a single interface for communication. At the load balancer node, the bandwidth for each direction of Full duplex Ethernet is consumed by the sum of request and response packets.

interframe gaps. Multiplying that with 190K [req/sec] and 8 [bit/byte] will result in 966.72 Mbps. Therefore the throughput of about 190K [req/sec] is a reasonable number as the maximum performance level in 1Gbps network environment.

While nginx did not show any benefit as the load balancer, the performance of the ipvs load balancer container showed equivalent performance level as iptables DNAT. This means that the proposed ipvs container load balancer is at least as good as the internal load balancer for Kubernetes in the 1 Gbps network.

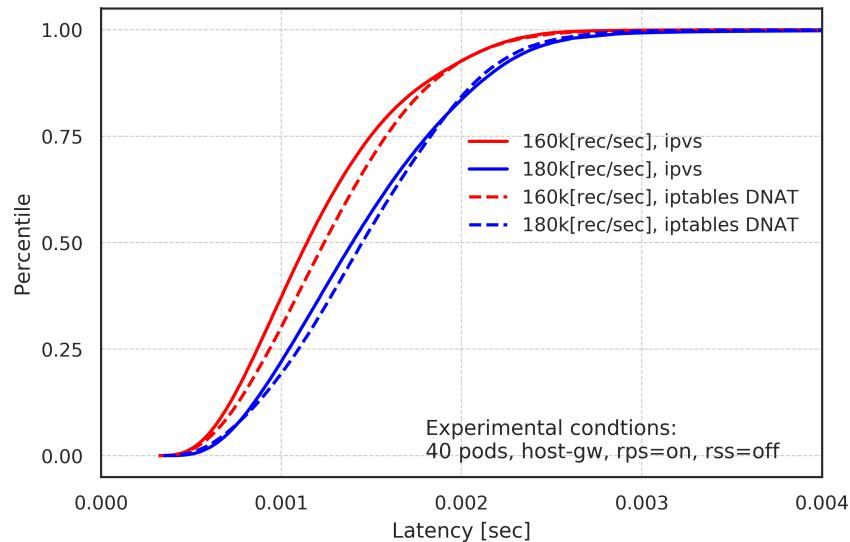


Figure 4.7: Latency for ipvs and iptables DNAT. Cumulative Distribution Function(CDF) of the latency for ipvs and iptables DNAT are compared, at the two constant loads, 160K[req/sec] and 180K[req/sec] . Smaller latencies are observed for ipvs.

Figure 4.7 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. It is seen that the latencies 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. While these may be considered a subtle difference, however, this indicates that proposed load balancer is at least as good as iptables DNAT.

Fig. 4.8 compares the CPU usage for the proposed ipvs load balancer in container and iptables DNAT at the time of the throughput measurement in the on-premise data

center. Since the CPU usage was higher for the ipvs in a container, the proposed load balancer may be less efficient compared with the iptables DNAT. However, since single hardware can accommodate 1 Gbps traffic with CPU usage of about 60%, the authors regard this as a tolerable overhead.

The authors plan to improve the efficiency of the proposed load balancer by developing a software load balancer using eXpress data path(XDP) technology[21] in the future work and thereby improving the performance levels of the portable load balancer.

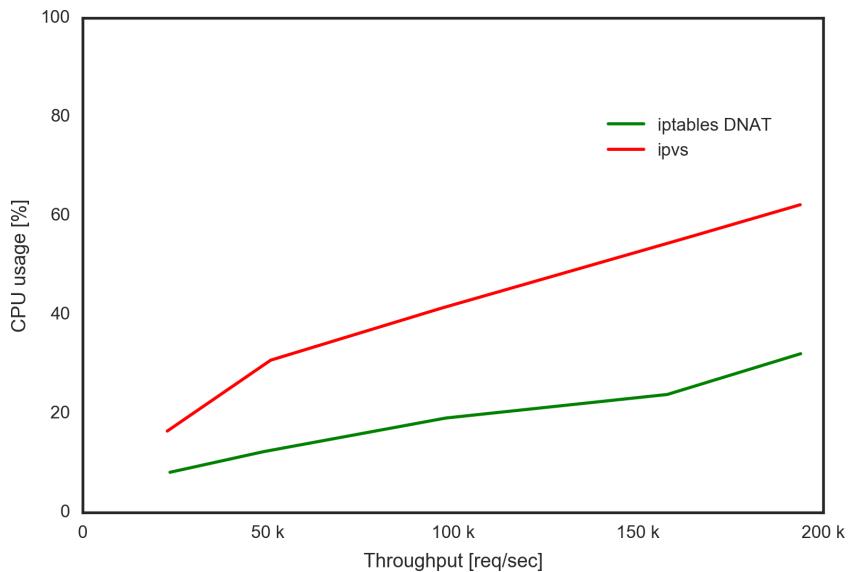


Figure 4.8: CPU usage of the ipvs and iptables DNAT. CPU usage of ipvs is higher than that of iptables DNAT.

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 4.9 shows the schematic diagram illustrating the packet flow of the L3DSR experiment. The red arrows indicate the route HTTP request packets follow and the blue arrows indicate the route response packets follow. Since the response packets directly return to the client, the load balance is expected to perform better.

The ipvs has a 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. Since iptables DNAT does not have the functions that enable L3DSR settings, this is one of the benefits only available for the ipvs load balancer.

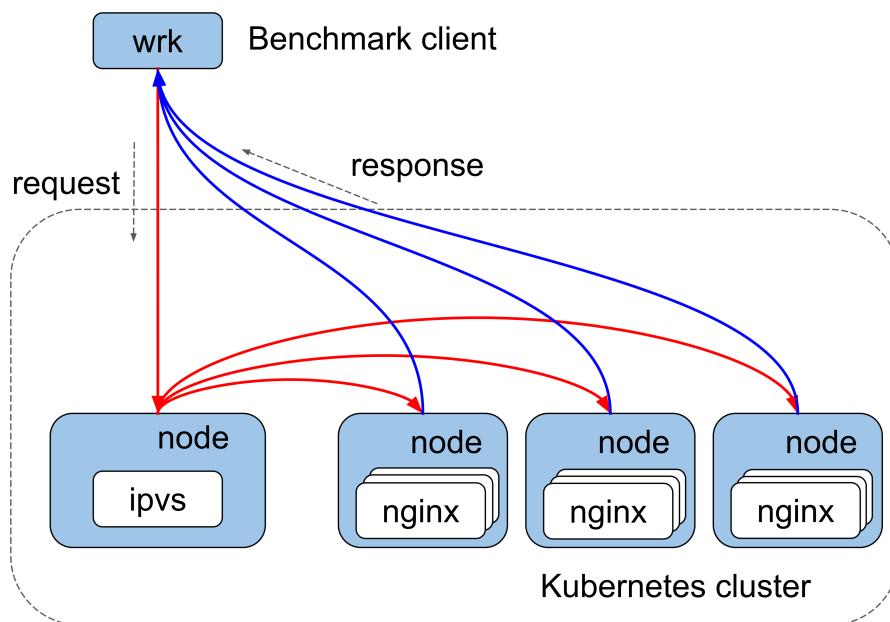


Figure 4.9: Experimental setup for L3DSR throughput measurement. The red arrows indicate the route HTTP request packets follow and the blue arrows indicate the route response packets follow. Since the response packets directly return to the client, the load balance is expected to perform better.

The author carried out throughput measurement using the experimental setup shown in Figure 4.9. Figure 4.10 compares the throughput of the ipvs-tun, conventional ipvs and iptables DNAT. As can be seen in the figure, while the performance levels for ipvs and iptables DNAT exactly match, the performance level of ipvs-tun is much higher than those. For example, the throughput of ipvs-tun is 1.5 times higher than

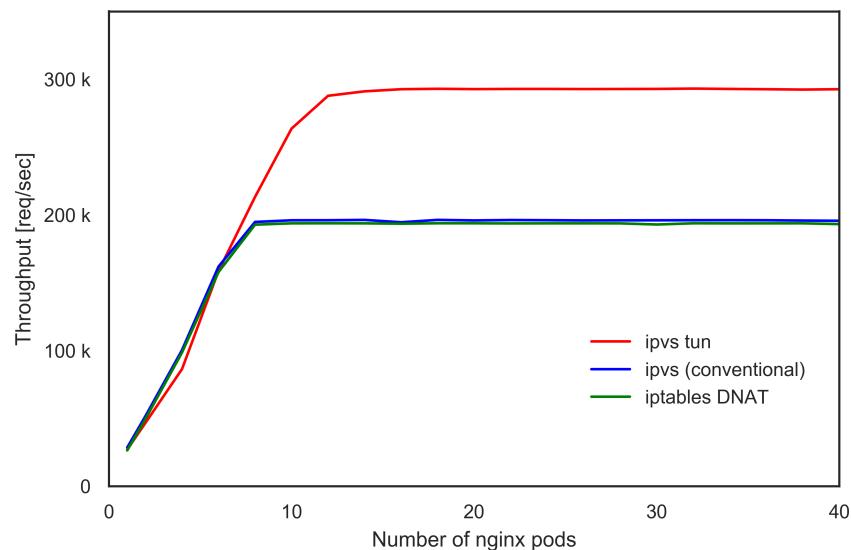


Figure 4.10: Throughput of L3DSR using ipvs-tun. The throughput of ipvs-tun is 1.5 times higher than those of conventional ipvs and iptables DNAT.

those of conventional ipvs and iptables DNAT.

4.2 Cloud experiment

The throughput measurements were also carried in GCP and AWS to show the containerized ipvs load balancer is runnable even in the cloud environment. The specifications of virtual machines used for the experiment in both environment are summarized in Table 4.3 and Table 4.4. In the cases of cloud environments, it is easy to change the machine specifications, especially CPU counts. Therefore, the author measured throughput with several conditions of them. In the case of GCP, custom instance with 32Gbyte memory and with 8, 16, and 32 CPU are used. And in the case of AWS instance type of c4.2xlarge, c4.4xlarge, and c4.8xlarge are used. The vxlan mode of the flannel is used for the overlay network in both of the cloud environment. As for multicore packet processing, different settings depending on the number of prepared queues for VMs. The setting “(RSS, RPS) = (on, off)” is used in GCP and the setting “(RSS, RPS) = (off, on)” is used in AWS.

[GCP VM Instance Specification for Client and Web Server Nodes]

Instance type: custom instance
CPU: Xeon 2.2GHz, 16 cpus
Memory: 16GB
NIC: virtio_net /w 16 rx-queues
(Node x 6, Client x 1)

[GCP VM Instance Specification for Load balancer Node]

Instance type: custom instance
CPU: Xeon 2.2GHz, 8, 16, 32 cpus
Memory: 16GB
NIC: virtio_net /w 8, 16, 32 rx-queues
(Load balancer x 1)

Table 4.3: Virtual Machine specifications in GCP experiment. The author measured throughputs using load balancer nodes with 8 CPUs, 16 CPUs, and 32 cups. The number of rx-queues of each node was 8, 16 and 32, respectively. Since the same number of rx-queues as the number CPU is prepared, the setting with “(rss, rps) = (on, off)” is used.

Figure 4.11 and Figure 4.12 show the load balancer performance levels that are

[AWS VM Instance Specification for Client and Web Server Nodes]

Instance type: m4.4xlarge

CPU: Xeon E5-2686 v4 2.30GHz, 16 cpus

Memory: 64GB

NIC: ixgbevf /w 2 rx-queues

(Node x 6, Client x 1)

[AWS VM Instance Specification for Load balancer Node]

Instance type: c4.2xlarge, c4.4xlarge, c4.8xlarge

CPU: Xeon E5-2666 v3 2.90GHz, 8, 16, 36 cpus

Memory: 15GB, 30GB, 60GB

NIC: ixgbevf /w 2 rx-queues

(Load balancer x 1)

Table 4.4: Virtual Machine specifications in AWS experiment. The author measured throughputs using load balancer nodes with 8 CPUs, 16 CPUs, and 36 CPUs. Since there are only two rx-queues, the setting with “(rss, rps) = (off, on)” is used.

measured in GCP and AWS, respectively. Both results show similar characteristics as the experiment in an on-premise data center in Figure 4.6, where throughput increased linearly to a certain saturation level that is determined by either network speed or machine specifications. It is also seen that the performance levels are higher for load balancer nodes with more CPUs in both GCP and AWS. These results indicate that the proposed ipvs load balancers can be run in GCP and AWS, and function properly.

In general, the throughput results in the cloud environments are inferior to those in the on-premise data center, even though much more powerful CPUs are used for VMs in the cloud environment than the experiment in the on-premise data center. For example, while the maximum throughput of the ipvs load balancer with eight physical cores was 190K[req/sec] in the on-premise data center(Figure 4.2), the maximum throughput in GCP and AWS are 160K[req/sec] and 100K[req/sec], respectively, even with 32 and 36 virtual CPUs. Although the author suspects that this is probably due to the inefficiency of the virtual machines as opposed to Bare Metal servers, it is not clear what limits the maximum performance levels of cloud environments. The author leaves a detailed analysis for future work.

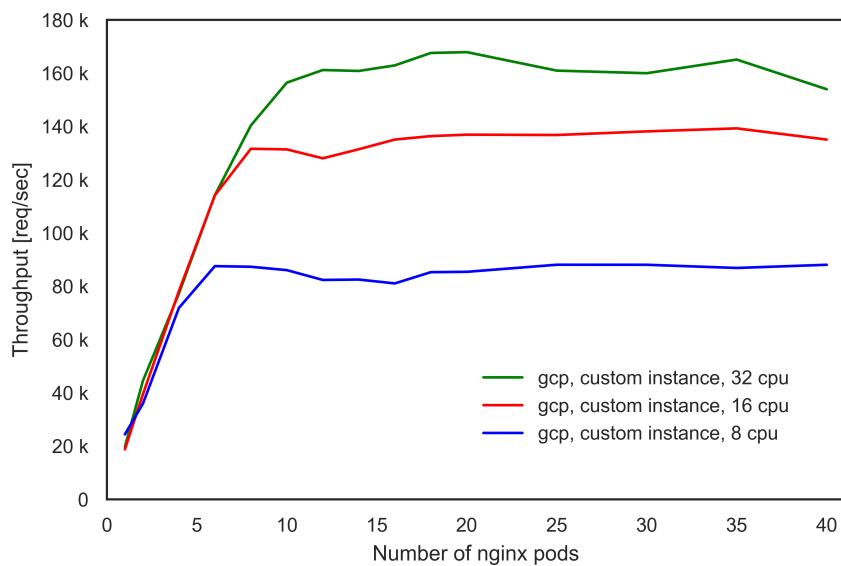


Figure 4.11: Throughput measurement results in GCP. The throughput increases linearly as the number of nginx *pods* increases until it reaches the saturation level. The maximum throughput is higher for instances with more virtual CPUs.

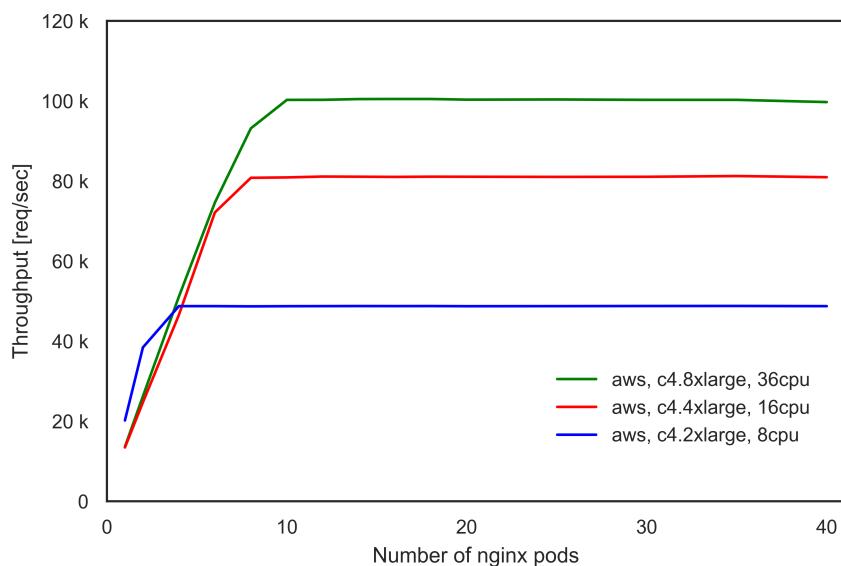
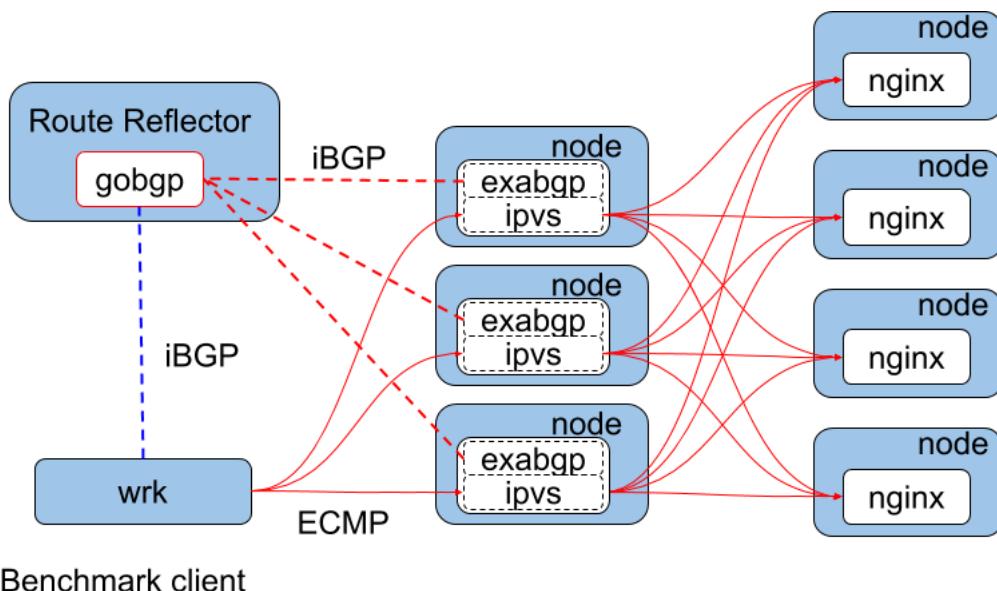


Figure 4.12: Throughput measurement results in AWS. The throughput increases linearly as the number of nginx *pods* increases until it reaches the saturation level. The maximum throughput is higher for instances with more virtual CPUs.

4.3 Redundancy with ECMP

While containerizing ipvs makes it runnable in any environment, it is essential to discuss how to route the traffic to the ipvs container in a redundant and scalable manner. The ECMP technique is expected to make the load balancers redundant and scalable since all the load balancer containers act as active. The author 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, the author also measured the throughput from a benchmark client with ECMP routes when multiple of the ipvs container load balancers are deployed.



Benchmark client

Figure 4.13: Benchmark setup for ECMP experiment. Each load balancer pods 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.

Figure 4.13 shows the schematic diagram of the experimental setup. Table 4.5 summarizes hardware and software specifications. Notable differences from the previous throughput experiment in Figure 4.1 and Table 4.2 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.

[Hardware Specification]

CPU: Xeon E5-2450 2.10GHz x 8 (with Hyper Threading)

Memory: 32GB

NIC: Broadcom BCM5720 with 4 rx-queues, 1 Gbps

(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 4.5: Hardware and software specifications for ECMP experiment. The NIC of the benchmark client is 10 Gbps card to measure the aggregated throughput of 1Gbps load balancers.

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 *pods*.

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 *pods* and a service with two load balancer *pods*.

Table 4.6: ECMP routing tables. All the routing rules are updated by zebra. (a) According to this entry, packets toward 10.1.1.0/24 are forwarded to 10.0.0.106. (b) There is a routing rule with three next hops towards 10.1.1.0/24, each of which is the node where the load balancer pods are running. The weights of the three next-hops are all 1. (c) There are two routing rules regarding the services with different service IPs, one with three load balancers and the other with two load balancers. These load balancers share the same group of nodes, i.e., (10.0.0.105, 10.0.0.106, 10.0.0.107).

First, the author examined ECMP functionality by monitoring the routing table on the benchmark client. Table 4.6 (a) shows the routing table entry on the router when a single load balancer pod exists. From this entry, it is seen 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 updated by zebra.

The routing table entry in Table 4.6 (b) is seen when the number of the load balancer pods is increased to three. There are three next hops towards 10.1.1.0/24, each of

which is 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 the author increased the number of the load balancers.

Table 4.6 (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 services with different service IPs on the same group of nodes(10.0.0.105, 10.0.0.106, 10.0.0.107). The one has three load balancers and the other has two load balancers. The update of the routing entry was almost instant as the author started the load balancers for the second service.

As far as the route withdrawal is concerned, if an exabgp is killed by SIGKILL or SIGTERM the kernel of the node close the BGP connection by sending out a packet with FIN flag to the peer gobgpd on the route reflector, and thus the route is withdrawn immediately. The gobgp on the route reflector also periodically check the BGP connection, and if the peer exabgp container is unresponsive for more than the specified duration, “hold-time” setting in gobgpd, it will also terminate the connection and withdraw the route. The packets arriving within that duration will be dropped. However, it is possible to set up the “hold-time” short enough so that the retransmitted TCP packets from the client will be forwarded correctly to functioning load balancers.

The author also carried out throughput measurement to show that the proposed architecture increases the throughput as the number of the load balancers is increased. Figure ?? shows the results of the measurements. There are four solid lines in the figure, each corresponding to 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). The performance level increases linearly as the number of the load balancers increases up to four of the ipvs load balancers, but does not scale further. This is because the benchmark program uses up CPU power of the benchmark client, i.e., the CPU usage is 100% when there are more than four load balancers. The author expects 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.

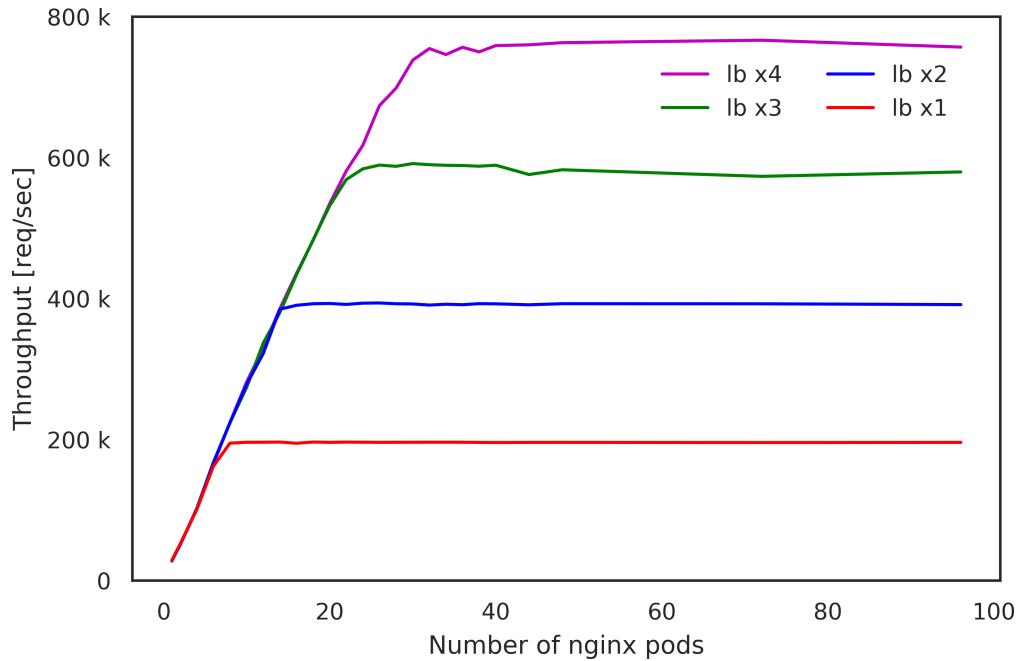


Figure 4.14: Throughput of ECMP redundant load balancer. The throughputs are measured for a single load balancer(lb x1), two(lb x2), three(lb x3) and four(lb x4) load balancers.

Figure ?? shows the throughput measurement results when the author periodically changed the number of the load balancers. The red line in the figure shows the number of ipvs load balancer pods, which is changed randomly every 60 seconds. The blue line corresponds to the resulting throughput. As can be seen 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.

Figure ?? shows a histogram of the ECMP update delay, where the author measured the delays until the number of running ipvs pods is reflected in the routing table on the benchmark client, as the number of the ipvs pods is changed randomly every 60 seconds for 20 hours. As can be seen from the figure, most of the delays are within 6 seconds, and the largest delay during the 20 hours experiment was 10 seconds. The author concludes that ECMP routing update in the proposed architecture is quick enough.

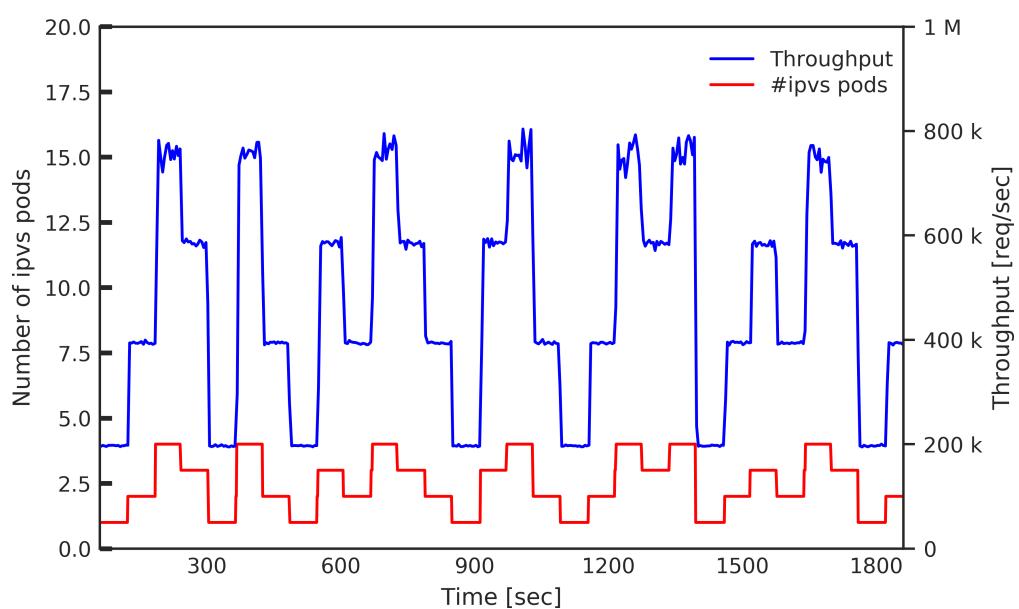


Figure 4.15: Throughput responsiveness. Throughput responsiveness when the number of load balancers is changed randomly in every 60 seconds is shown.

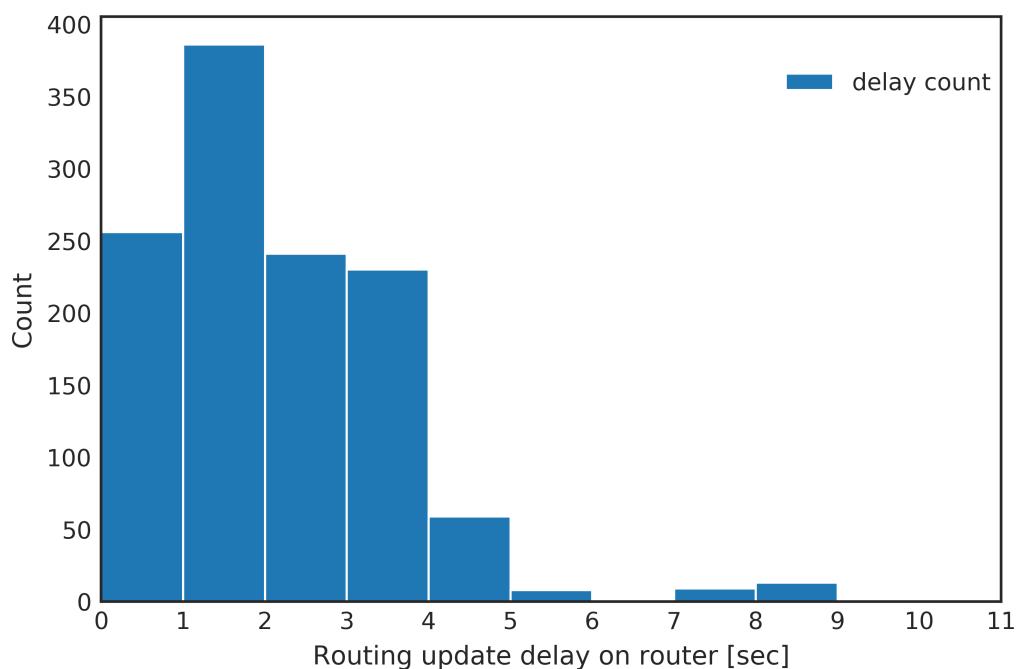


Figure 4.16: ECMP update delay histogram. This shows the delays until the number of running ipvs pods is reflected in the routing table on the benchmark client, when the number of the ipvs pods is changed randomly every 60 seconds for 20 hours.

4.4 Summary

In this chapter, the results of the performance evaluation for the proposed load balancer. The author carried out throughput measurement to verify the feasibility of the load balancer. The author confirmed that the proposed load balancer is portable, and the ECMP routing is redundant and scalable.

From the throughput measurement of the proposed load balancer using physical servers in the on-premise data center, the author confirmed the followings; The throughput of the proposed load balancer linearly increases as the number of nginx *pods* increases, and then it eventually saturates, indicating the load balancer functions properly. The throughput depends on the multicore packet processing settings and overlay network settings. The throughput increases as the number of CPU cores that are utilized for packet processing. The throughput of the host-gw mode without any encapsulation is higher than vxlan mode and udp mode, where the packet is encapsulated. The throughput of the ipvs in a container is equivalent to that of the iptables DNAT as a load balancer, in 1Gbps network environment. This is due to the bandwidth limitation of the experimental network. By utilizing ipvs-tun mode, i.e., L3DSR, the throughput of the ipvs container can be further improved, up to 1.5 times.

The author also carried out the same throughput measurement in GCP and AWS to show the containerized ipvs load balancer is runnable even in the cloud environment. While the performance levels of the proposed load balancers in cloud environments are inferior to those in on-premise data centers, the ipvs container can run and function properly, in both GCP and AWS.

Finally, the author evaluated ECMP functionality by monitoring routing table updates on the router when the new load balancer is added or removed. The author also measured throughput by changing the number of load balancers. The linear scalability of the ECMP throughput has been confirmed up to 4x load balancers. The ECMP routing update in the proposed architecture is properly functioning and quick enough.

5

Further Improvement

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.

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

Figure 5.1 shows the packet flow for ipvs-nat and iptables DNAT, and Figure 5.2 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, a better performance level is expected for ipvs-tun.

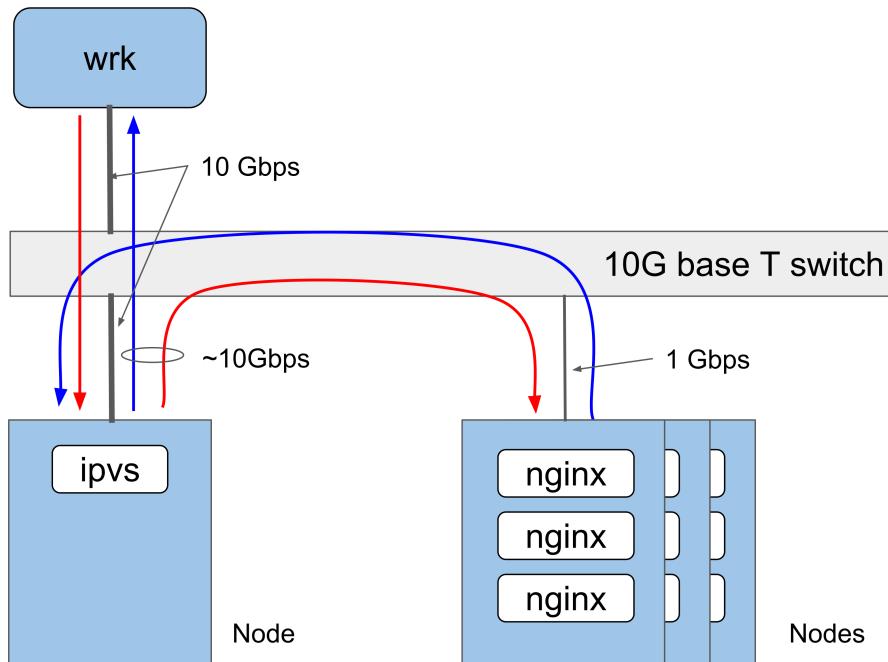


Figure 5.1: Packet flow of ipvs-nat and iptables DNAT.

Figure 5.3 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 5.1 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

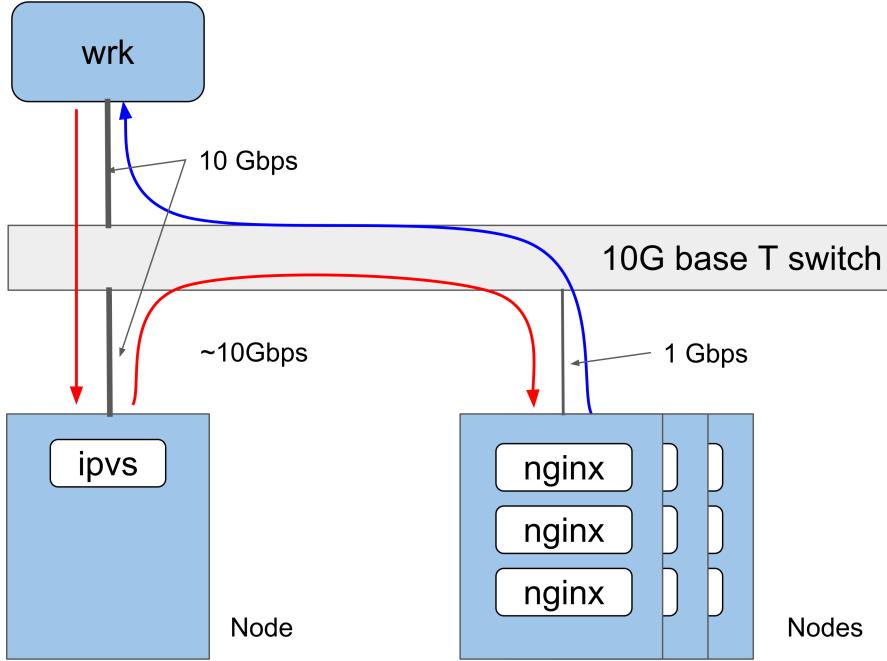


Figure 5.2: Packet flow of ipvs-tun.

[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)[46, 37] 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 5.1: Performance levels in 1Gbps and in 10Gbps.
Throughput results of the load balancers at 40 nginx pods from the data for the Figures 4.10 and 5.3 are shown.

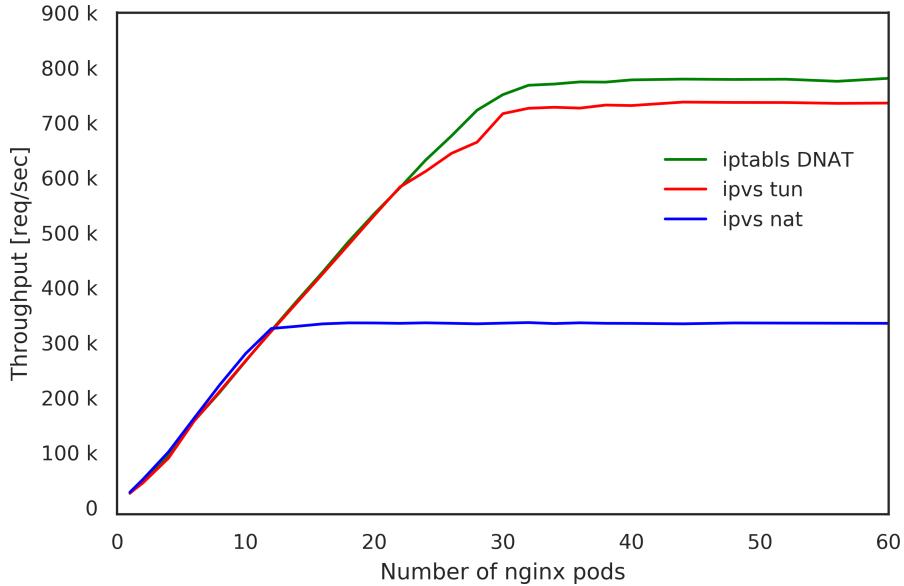


Figure 5.3: Throughput of load balancers in 10 Gbps.

5.2 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 5.4 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 5.3.

Table 5.2 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 5.3 and 5.4. 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.

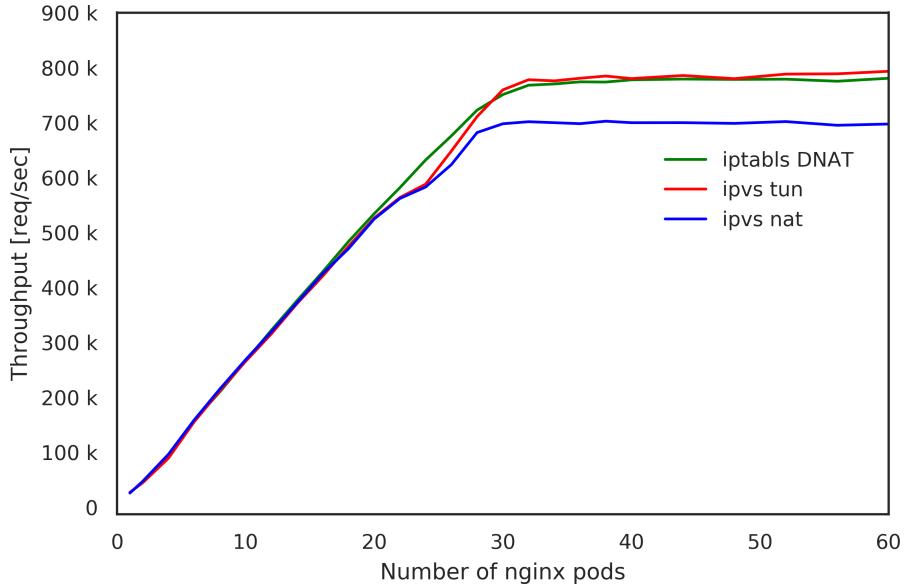


Figure 5.4: Throughput of load balancers in node name space.

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

Table 5.2: 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 5.3 and 5.4 are shown.

5.3 XDP load balancer

[Filled in later]

5.4 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 observed. 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.

[Filled in later]

6

Related Work

This Chapter provides related works and the background information of this study.

6.1 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[1] capability for handling situations where multiple Kubernetes clusters¹ 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

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

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[35, 22] utilizes the ingress[2] 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[36] project is trying to use Linux kernel's ipvs[45] load balancer capabilities by containerizing the keepalived[6]. 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[15, 12] 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[38] and clusterf[28] 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[10] and HashiCorp's Consul[19]. Although these were usable to implement a containerized load balancer in our proposal, we did not use them, since 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[14] 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[33] is another software load balancer implementation using ECMP and windows network stack. Ananta can be solely used in Microsoft's Azure cloud infrastructure[33]. 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.

7

Conclusion and future work

7.1 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 proposed load balancer architecture utilizes software load balancers with container technology to make the load balancers runnable in any base infrastructure. It also utilizes ECMP technology to make multiple load balancers active, and thereby to provide redundancy and scalability.

The author implemented a containerized software load balancer that is run by Kubernetes as a part of container cluster, using Linux kernel's IPVS. In order to discuss the feasibility of the proposed load balancer, performance measurements are conducted in 1 Gbps network environment. It was shown that the proposed load balancers are runnable in an on-premise data center, GCP and AWS. Therefore the proposed load balancers can be said to be portable. The throughput levels of a load balancer are dependent on settings for multi-core packet processing. It was shown that better to use

as many CPU cores as possible for packet processing. The throughput levels are also very dependent on the overlay network backend mode. The host-gw mode where no tunneling is used resulted in the best performance level, and the vxlan mode resulted in the second best. In the experiment in 1 Gbps network environment, the ipvs-nat load balancer in the container had the same performance level as load balancing function of iptables DNAT on the node. Furthermore, the performance level of ipvs-tun load balancer in a container with the L3DSR setup was about 1.5 times larger than that of iptables DNAT. Therefore in 1 Gbps network environment, the proposed load balancer is portable while it has the 1.5 times better performance level or the same performance level depending on the mode of operation.

Also implemented is the ECMP setups where multiple of the load balancer containers are deployed, each advertising the route to the service VIP. The ECMP technique makes 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 by utilizing multiple of load balancers simultaneously, the throughput of the total system is increased significantly. These characteristics are evaluated by checking the routing table of the upstream router and by throughput measurement. 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 maximum performance levels of the cluster of load balancers scaled linearly as the number of the load balancer pods was increased up to four of them. 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 author also extended the throughput measurement into 10 Gbps network environment. It was revealed that ipvs-nat and ipvs-tun load balancers in containers had lower performance levels compared with the iptables DNAT. This has been suspected to be due to the overhead of the container network, i.e., veth+bridge. By setting up the load balancing table in node net namespaces, the performance levels of ipvs-nat and ipvs-tun became closer to that of the iptables DNAT. Although the overheads of the container network were invisible in 1 Gbps network environment, they were no longer invisible in 10 Gbps network environment. The author is currently implementing and evaluating a novel software load balancer using XDP technology to

provide a better alternative to ipvs as a portable load balancer.

The outcome of this 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 this 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.

[Filled in later]

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.

Bibliography

- [1] The Kubernetes Authors. *Federation*. 2017. URL: <https://kubernetes.io/docs/concepts/cluster-administration/federation/>.
- [2] The Kubernetes Authors. *Ingress Resources / Kubernetes*. 2017. URL: <https://kubernetes.io/docs/concepts/services-networking/ingress/>.
- [3] Bert Hubert et al. *Linux Advanced Routing & Traffic Control HOWTO*. 2002. URL: <http://www.tldp.org/HOWTO/Adv-Routing-HOWTO/index.html> (visited on 07/14/2017).
- [4] Gilberto Bertin. “XDP in practice: integrating XDP into our DDoS mitigation pipeline”. In: *Technical Conference on Linux Networking, Netdev*. Vol. 2. 2017.
- [5] Brendan Burns et al. “Borg, omega, and kubernetes”. In: (2016).
- [6] Alexandre Cassen. *Keepalived for Linux*. URL: <http://www.keepalived.org/>.
- [7] Brian F. Cooper. “Spanner: Google’s Globally-distributed Database”. In: *Proceedings of the 6th International Systems and Storage Conference*. SYSTOR ’13. Haifa, Israel: ACM, 2013, 9:1–9:1. ISBN: 978-1-4503-2116-7. DOI: [10.1145/2485732.2485756](https://doi.acm.org/10.1145/2485732.2485756). URL: <http://doi.acm.org/10.1145/2485732.2485756>.
- [8] James C. Corbett et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013), 8:1–8:22. ISSN: 0734-2071. DOI: [10.1145/2491245](https://doi.acm.org/10.1145/2491245). URL: <http://doi.acm.org/10.1145/2491245>.
- [9] Inc CoreOS. *Backend*. URL: <https://github.com/coreos/flannel/blob/master/Documentation/backends.md> (visited on 07/14/2017).
- [10] Inc CoreOS. *etcd | etcd Cluster by CoreOS*. URL: <https://coreos.com/etcd> (visited on 07/14/2017).

- [11] Inc CoreOS. *flannel*. URL: <https://github.com/coreos/flannel> (visited on 07/14/2017).
- [12] Docker Inc. *Use swarm mode routing mesh / Docker Documentation*. 2017. URL: <https://docs.docker.com/engine/swarm/ingress/> (visited on 07/14/2017).
- [13] Jake Edge. *Creating containers with systemd-nspawn*. 2013. URL: <https://lwn.net/Articles/572957/>.
- [14] Daniel E Eisenbud et al. “Maglev: A Fast and Reliable Software Network Load Balancer.” In: *NSDI*. 2016, pp. 523–535.
- [15] Docker Core Engineering. *Docker 1.12: Now with Built-in Orchestration! - Docker Blog*. 2016. URL: <https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/>.
- [16] Exa-Networks. *Exa-Networks/exabgp*. July 2018. URL: <https://github.com/Exa-Networks/exabgp>.
- [17] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. “A scalable, commodity data center network architecture”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 38. 4. ACM. 2008, pp. 63–74.
- [18] Will Glozer. *wrk - a HTTP benchmarking tool*. 2012. URL: <https://github.com/wg/wrk>.
- [19] HashiCorp. *Consul by HashiCorp*. URL: <https://www.consul.io/> (visited on 07/14/2017).
- [20] Benjamin Hindman et al. “Mesos: A platform for fine-grained resource sharing in the data center.” In: *NSDI*. Vol. 11. 2011, pp. 22–22.
- [21] Toke Høiland-Jørgensen et al. “The eXpress data path: fast programmable packet processing in the operating system kernel”. In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM. 2018, pp. 54–66.
- [22] NGINX Inc. *NGINX Ingress Controller*. 2017. URL: <https://github.com/nginxinc/kubernetes-ingress>.
- [23] *ip-sysctl.txt*. URL: <https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt>.

- [24] Van Jacobson, Craig Leres, and S McCanne. “The tcpdump manual page”. In: *Lawrence Berkeley Laboratory, Berkeley, CA* 143 (1989).
- [25] ktaka-ccmp. *ktaka-ccmp/iptvs-ingress: Initial Release*. July 2017. doi: [10.5281/zenodo.826894](https://doi.org/10.5281/zenodo.826894). URL: <https://doi.org/10.5281/zenodo.826894>.
- [26] Victor Marmol, Rohit Jnagal, and Tim Hockin. “Networking in Containers and Container Clusters”. In: *Netdev* (2015).
- [27] Martin A. Brown. *Guide to IP Layer Network Administration with Linux*. 2007. URL: <http://linux-ip.net/html/index.html> (visited on 07/14/2017).
- [28] Tero Marttila. “Design and Implementation of the clusterf Load Balancer for Docker Clusters”. en. Master’s Thesis, Aalto University. 2016-10-27, pp. 97+7. URL: <http://urn.fi/URN:NBN:fi:aalto-201611025433>.
- [29] Paul B Menage. “Adding generic process containers to the linux kernel”. In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 45–57.
- [30] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.
- [31] Walter Milliken, Trevor Mendez, and Dr. Craig Partridge. *Host Anycasting Service*. RFC 1546. Nov. 1993. doi: [10.17487/RFC1546](https://doi.org/10.17487/RFC1546). URL: <https://rfc-editor.org/rfc/rfc1546.txt>.
- [32] Vivian Noronha et al. “Performance Evaluation of Container Based Virtualization on Embedded Microprocessors”. In: *2018 30th International Teletraffic Congress (ITC 30)*. Vol. 1. IEEE. 2018, pp. 79–84.
- [33] Parveen Patel et al. “Ananta: Cloud scale load balancing”. In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 207–218.
- [34] Andrew Pavlo and Matthew Aslett. “What’s really new with NewSQL?” In: *ACM Sigmod Record* 45.2 (2016), pp. 45–55.
- [35] Michael Pleshakov. *NGINX and NGINX Plus Ingress Controllers for Kubernetes Load Balancing*. Dec. 2016. URL: <https://www.nginx.com/blog/nginx-plus-ingress-controller-kubernetes-load-balancing/>.
- [36] Bowei Du Prashanth B Mike Danese. *kube-keepalived-vip*. 2016. URL: <https://github.com/kubernetes/contrib/tree/master/keepalived-vip>.

- [37] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. “Performance evaluation of containers for HPC”. In: *European Conference on Parallel Processing*. Springer. 2015, pp. 813–824.
- [38] Andrey Sibiryov. *GORB Go Routing and Balancing*. 2015. URL: <https://github.com/kobolog/gorb>.
- [39] Alan Sill. “Standards Underlying Cloud Networking”. In: *IEEE Cloud Computing* 3.3 (2016), pp. 76–80. ISSN: 23256095. DOI: [10.1109/MCC.2016.55](https://doi.org/10.1109/MCC.2016.55).
- [40] E. Chen T. Bates and R. Chandra. *BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP)*. RFC 4456. RFC Editor, Apr. 2006, pp. 1–12. URL: <https://www.rfc-editor.org/rfc/rfc4456.txt>.
- [41] Kimitoshi Takahashi et al. “A Portable Load Balancer for Kubernetes Cluster”. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM. 2018, pp. 222–231.
- [42] Tom Herbert and Willem de Bruijn. *Scaling in the Linux Networking Stack*. URL: <https://www.kernel.org/doc/Documentation/networking/scaling.txt> (visited on 07/14/2017).
- [43] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys ’15* (2015), pp. 1–17. DOI: [10.1145/2741948.2741964](https://doi.org/10.1145/2741948.2741964). URL: <http://dl.acm.org/citation.cfm?doid=2741948.2741964>.
- [44] Daniel Walton et al. *Advertisement of multiple paths in BGP*. RFC 7911. RFC Editor, July 2016, pp. 1–8. URL: <https://www.rfc-editor.org/rfc/rfc7911.txt>.
- [45] Wensong Zhang. “Linux virtual server for scalable network services”. In: *Ottawa Linux Symposium* (2000).
- [46] Yang Zhao et al. “Performance of Container Networking Technologies”. In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM. 2017.

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 &IPVSCController{}
}

type IPVSCController struct {}

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

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

```

```
}
```

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

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

func (ipvs IPVController) Info() *ingress.BackendInfo {
    return &ingress.BackendInfo{
        Name:      "dummy",
        Release:   "0.0.0",
        Build:     "git-00000000",
        Repository: "git://foo.bar.com",
    }
}

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

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

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

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 Gobgp 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
```

C

Analysis of the performance limit

The maximum throughput 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[24] 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 4.1. The worker sends out 100 requests to the web server within each connection, and closes it either if all of the responses are received 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 receives 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;

$$\begin{aligned} \text{throughput[req/sec]} &= \text{band width[byte/sec]}/(\text{request} + \text{response}) \\ &= 1\text{e}9/8/(\text{data}+665.36) \end{aligned}$$

Figure ?? 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 tcpdump(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.

D

VRRP

D.1 VRRP

Fig. D.1 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 switches 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

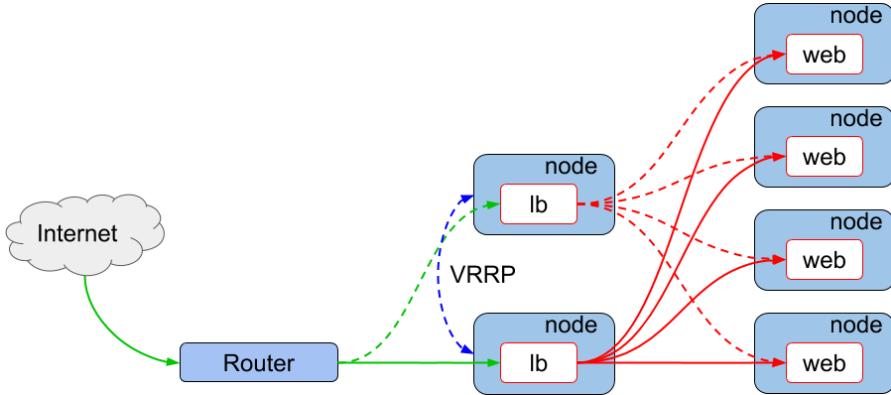


Figure D.1: An alternative redundant load balancer architecture using VRRP. The traffic from the internet is forwarded by the upstream router to a active lb node and then distributed by the lb pods to web pods using Linux kernel’s ipvs. The active lb pod is selected using VRRP protocol.

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.