Vol.2017-OS-141 No.14
2017/7/27

Kubernetes クラスタ用ポータブルロードバランサの検討

髙橋 公俊[†1]　合田 憲人[†1, †2]　丹生 智也[†2]　孫 静涛[†2]

概要：Linux コンテナのクラスタからなる WEB サービスが注目を集めている。BCP(Business continuity planning)や DR(Disaster Recovery)の目的でこのような WEB サービスを異なるクラウドプロバイダへマイグレーションする必要性が出ている。既存のコンテナ管理システム Kubernetes を調査したところ、異なるクラウドプロバイダ間で WEB サービスのマイグレーションを実現するためには、ロードバランサの仕組みに課題があることがわかった。
Kubernetes は外部のロードバランサに依存しており、コンテナクラスタの起動と同時にクラウドプロバイダが提供する API を通してロードバランサのセットアップを行うが、ロードバランサの仕組みや API はクラウドプロバイダによって異なるため、クラウドプロバイダ毎に実装が必要であり利用可能なクラウドプロバイダが限定されてしまう。また、オンプレミスデータセンタ用のロードバランサについてもセットアップ方法が製品によって異なるため、全ての製品に対応することは困難である。
本稿ではこの課題を解決するために、ソフトウェアロードバランサをコンテナとして実装することで外部ロードバランサへの依存を排除することを試みる。具体的には、Linux カーネルの L4 ロードバランサ機能である ipvs を、Kubernetes のコンテナとして稼働させ、既存の iptables による負荷分散機能との性能比較を行う。また適切な実装方法、ネットワーク構成についても検討する。

キーワード：コンテナ，ロードバランサ，Kubernetes

Portable Load Balancer for Kubernetes Cluster System

KIMITOSHI TAKAHASHI[†1] KENTO AIDA[†1, †2]
TOMOYA TANJO[†2] JINGTAO SUN[†2]

1. はじめに

Linux コンテナのクラスタからなる WEB システムが注目を集めている。Linux コンテナは可搬性が高く、BCP(Business continuity planning)や DR(Disaster Recovery)の目的で WEB システムを異なるクラウドプロバイダのクラウド基盤間でマイグレーションすることが容易であるという利点がある。
しかし既存のコンテナ管理システム Kubernetes[1]を調査したところ、異なるクラウドプロバイダ間でマイグレーションを実現するためには、ロードバランサの仕組みに課題があることがわかった。具体的には、Kubernetes は外部のロードバランサに依存しており、コンテナクラスタの起動と同時にクラウドプロバイダが提供する API を通してロードバランサのセットアップを行う。ところが、ロードバランサの仕組みや API はクラウドプロバイダによって異なり、今の所一部の主要なクラウドプロバイダでしかロードバランサを使うことができない。また、オンプレミスデータセンタ用のロードバランサ製品は多岐に渡り、その多くを Kubernetes から API で制御することは非常に困難である。本稿ではこの課題を解決するためにロードバランサ自体をコンテナとして実装する方法について検討する。

2. 関連研究

以下に本研究に関連する研究について述べる。Kuberenetes は Federation モードをサポートしており Google Cloud, AWS, and on-premises (e.g. on OpenStack)といった Hybrid で利用可能であるとされている[3] しかしながらこの場合も、外部からのトラフィックのルーティングは外部ロードバランサに依存しており、本稿で提案する様々なクラウドプロバイダ、オンプレミスデータセンタ上でで使えるジェネリックなロードバランサとは一線を画するものである。

ロードバランサのコンテナ化については、以下のような関連研究がある。

nginx ingress[5,6]は Kubernetes の Ingress Controller[12] の仕組みを利用して nginx をポータブルなロードバランサとして実装している。nginx は高速な WEB サーバプログラムとして人気が高いが、Layer 7 のロードバランサ機能も併せ持っている。SSL 対応可能であり、URI 毎に分散先を決めたりすることができフレキシブルであるが、Layer7 であるが故にパフォーマンスに問題がある。ロードバランサとしての性能は、WEB サーバとしての性能を超えることはなく、万能ではない。

kube-keepalived-vip[7]は Kubernetes の daemonset の仕組み

†1 総合研究大学院大学
The Graduate University for Advanced Studies
†2 国立情報学研究所
National Institute of Informatics

ⓒ 2017 Information Processing Society of Japan

を利用し、コンテナ化された keepalived[8]により ipvs の設定を行う。本稿で目指す実装に非常に近いが、公開されている情報に性能面での評価データや Overlay Network との関連性の評価などは示されていない。

Docker の Swarm モード[11,12]は ipvs を用いて、クラスタ内部の負荷分散を行なっている。Kubernetes において iptables の利用しているのと同様の位置付けであると考えられ、本稿で提案するポータブルなロードバランサとは異なる。

gorb[9], clusterf[10]は Docker のコンテナ内で使用可能な、ipvs セットアップ用デーモンである。Consul[14], etcd[15]などのキーバリューストレージにストアされたコンテナ情報を元に ipvs の設定を行う。本稿で実装する Ingress Controller と keepalived の機能に近いが Kuberenetes で使用可能であるかどうかは不明である。

本稿では Linux カーネルのロードバランサ機能 ipvs をコンテナ化することで、Kuberenetes のユーザーがどのクラウドプロバイダ、オンプレミスデータセンタでも利用可能なポータブルなロードバランサを提案し、定量的な評価行なっている点が新しい。

3. 課題と解決方法

3.1 Kubernetes におけるロードバランサの課題

図 1 に Kubernetes システムの構成図を示す。Kubernetes システムは Master と複数の Node で構成される。Master が kubectl コマンドによるリクエストを受けると、各ノードで Pod が作成される。Pod は IP アドレスなどのリソースを共有する複数のコンテナをまとめたもので、Kubernetes では Pod 単位でコンテナが実行される。

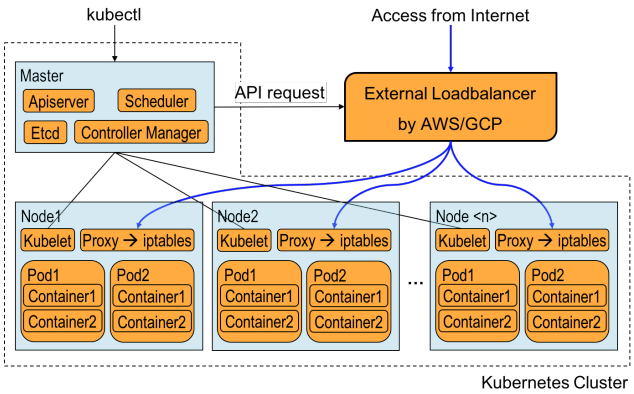

図 1 Kubernetes システム構成図

Kubernetes でのロードバランサのセットアップは、Master からの API リクエストを通して行われる。セットアップされたロードバランサはインターネットからのアクセスを全ての Node に均等に振り分け、各 Node でそれらのアクセスをさらに適切な Pod に DNAT[16]を用いて接続している。以上が Kubernetes における外部ロードバランサの利用形態であるが、この方法には次のような問題がある。(1)ロードバランサの仕組みや API はクラウドプロバイダによって異なるため、Kubernetes にサポートされた非常に限られたクラウドプロバイダ以外はこの仕組みを使うことができない。(2)オンプレミスデータセンタで使われるロードバランサ製品は多岐に渡り、そもそもそれらの多くをサポートすることは非常に困難である。(3)Pod への DNAT は iptables を利用しているため、同時にデプロイされる WEB システムの数が増えると、設定が複雑になりデバッグが困難になる。(4)一旦全ての Node にアクセスを均等に分散し、そこから目的の Pod に DNAT を用いて接続しているため、アクセス経路が複雑になり障害ポイントの把握が困難である。

3.2 本研究での提案手法

図 2 に本稿が提案するロードバランサを用いた場合のシステム構成を示す。提案のロードバランサは次のような特徴を持っている。 (1)ロードバランサ自体が Kubernetes の Pod として起動される。(2)ロードバランサ用コンテナは冗長化されており、Active-Backup 構成をとる。(3)負荷分散先の Pod の情報を Master から取得し、動的に分散先に加えたり分散先から外したりすることができる。

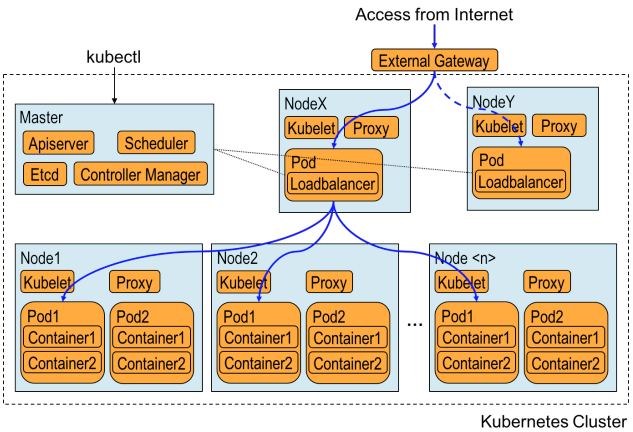

図 2 ポータブルロードバランサシステム構成図

このような構成をとることで、既存の Kubernetes システムでのロードバランサの問題を解決することができる。すなわち、ロードバランサ自体をコンテナ化しているので、Kubernetes による API サポートの如何に関わらず、オンプレミス環境でもクラウドプロバイダでも同じ仕組みのロードバランサを利用することができる。また iptables DNAT に依存しないためデバッグの複雑さが無く、外部からのアクセス経路も単純で、通信障害発生時にも、障害箇所を容易に特定することができる。

ⓒ 2017 Information Processing Society of Japan

本稿では、様々なクラウドプロバイダ、オンプレミスデータセンタ環境でも利用可能なポータブルなロードバランサとして、コンテナ化されたソフトウェアロードバランサを提案する。具体的には Linux カーネルのロードバランサ機能である ipvs[17]をコンテナ内で利用可能なように実装し、性能評価、課題の洗い出しなどを行う。

以降、実装方法ついて 4 章で述べ、5 章でその性能評価方法、6 章で評価結果について考察する。

4. 実装方法

図 3 に今回我々が実装したロードバランサの模式図を示す。

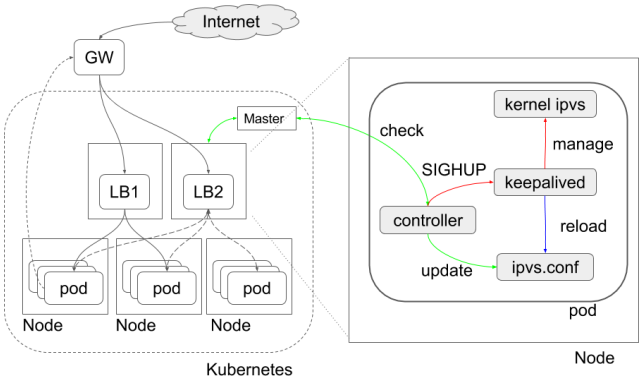

図 3 ロードバランサコンテナ内部の模式図

ロードバランサ用の Docker コンテナ内部では kubernetes の Ingress[13]の仕組みを利用した controller を稼働させている。controller[17]はサンプルを元に go 言語で実装したものである。既存のパッケージ ”k8s.io/ingress/core/pkg/ingress” を import することで、わずか 120 行で動作させることができた。controller は master と通信し pod の稼働状態を監視する。pod の増減などの変化があると keepalived の設定ファイルを再生成し、keepalived に SIGHUP を発行する。keepalived は新たな設定ファイルを読み込み、設定に基づいてカーネル内部の ipvs の分散テーブル上で分散先の追加削除を行う。

コンテナ内で Linux カーネルのロードバランサ機能を利用するためには、ip_vs.ko などの必要なカーネルモジュールをロードできること、ネットワーク設定ができることが必要である。そのためにロードバランサ用のコンテナ起動時に、CAP_ SYS_MODULE と CAP_ NET_ADMIN のケイパビリティを与えることが必要である。

図 4 に controller によって生成された、keepalived の設定ファイルの一例、図 5 にその際の ipvs のテーブルの様子を示す。fwmark[18]=1 のパケットを 172.16.21.2:80、172.16.80.2:80 へ Masq モードでフォワードする動作設定がなされていることがわかる。

```
1/etc/keepalived/ipvs.d/default-tea-svc-81.conf
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
        }
    }
}
```

図 4 生成された設定ファイル一例

```
# 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
```

図 5 生成された設定ファイル一例

5. ロードバランサの評価方法

wrk[19]というベンチマークツールを用いて、本稿で提案する ipvs をコンテナ化したロードバランサの評価を行う。比較対象として、Kubernetes にもともと備わる iptables ベースの負荷分散機構、nginx ingress に対してもベンチマークを行う。Kubernetes のクラスタを構築する際に flannel[20,21]と呼ばれる Overlay Network の仕組みが利用されることが多い。ロードバランのベンチマークの際には、flannel の 3 通りの動作モードそれぞれについて測定を行なった。また、マルチコア CPU をもつサーバで高速なルーターロードバランサを構築しようとする際に割り込み、プロトコル処理の分散処理が必要になることが多い。今回の評価ではできるだけ性能を引き出すために、これについても

ⓒ 2017 Information Processing Society of Japan

条件を変えて測定を行なった。以降で、以上についての詳細を述べる。

5.1 ベンチマーク手法

ロードバランサの性能評価を行うために wrk[19]を用いて外部より http リクエストを送信し正常応答が得られた数をカウントし、単位時間あたりの処理性能(Request/sec)のデータを取得した。
図 6 にベンチマーク実行時の構成模式図を示す。

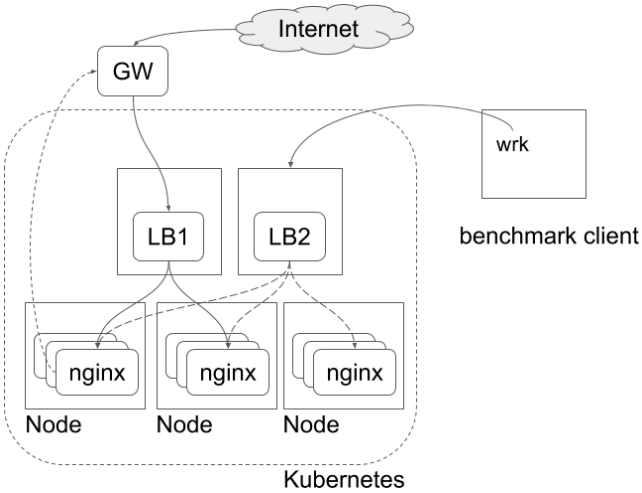

図 6　ベンチマーク構成図

kubernetes クラスタ内に用意されたロードバランサ LB2 に対して、外部のベンチマーククライアント上の wrk から HTTP GET リクエストを送り応答パケットをカウントすることで、Request/sec を測定する。測定対象には、本稿で提案するコンテナ化された ipvs の他に、Kubernetes に元々備わっている iptable による負荷分散機構、コンテナ化された L7 ロードバランサである nginx を含めた。

図 7 に評価に用いた wrk の書式と、測定結果のサンプルを示す。この書式でプログラムを実行すると wrk はスレッドを 40 個生成し、同時接続数 800 で 172.16.72.2 の 8888 番ポートに" GET / HTTP/1.0"リクエスト送信する。測定結果のサンプルを見るとスレッド単位での統計値、エラーカウント、Requests/sec、Transfer/sec などトータルなどの値が表示されていることがわかる。我々は、Requests/sec に着目し、複数の条件での測定結果について比較を行なった。
図 8 に本稿での評価に用いた、クラスタシステムのハードウェア構成、および主なソフトウェアの構成を示す。ロードバランサがアクセスを分散する先の http サーバとしては nginx を利用し、GET /リクエストに対しては、pod の IP アドレスを返すように設定した。レスポンスパケットがヘッダを除くと高々15byte の文字列であ、MTU に近いペイロードを使う場合に比べ、ロードバランサの性能をより限界近くまで測定することができる。

```
Command line:
  wrk -c800 -t40 -d30s http://172.16.72.2:8888/
        -c: concurrency, -t: # of thread, -d: duration
Result sample:
  Running 30s test @ http://10.254.0.10:81/
    40 threads and 800 connections
    Thread Stats    Avg        Stdev      Max     +/- Stdev
      Latency      15.82ms    41.45ms    1.90s     91.90%
      Req/Sec       4.14k     342.26     6.45k     69.24%
    4958000 requests in 30.10s, 1.14GB read
    Socket errors: connect 0, read 0, write 0, timeout 1
Requests/sec: 164717.63
Transfer/sec:      38.86MB
```

図 7　ベンチマークコマンドの書式と出力サンプル

```
Number of Physical Machines: Master: 1, Node: 7, Client: 1
  CPU: Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz
  Memory: 32GB
  NIC: Broadcom BCM5720 Gigabit Ethernet PCIe
  OS: Debian 8.7 (Jessie)
  Kernel: 3.16.0-4-amd64 #1 SMP Debian 3.16.39-1 (2016-12-30)
Node Software version:
      Kubernetes v1.5.2
      flannel v0.7.0
      etcd version: 3.0.15
Load balancer Software version:
      Keepalived: v1.3.2 (12/03,2016)
      nginx: nginx/1.11.1
Worker Pod Software vesion:
      nginx : nginx/1.13.0 (GET /で Pod IP を返すように設定)
```

図 8　ハードウェアおよび主なソフトウェア構成

5.2 Overlay Network について

Kubernetes においては様々な、Overlay Network が利用可能である。今回、我々は flannel[20,21]を利用して Kubernetes のシステムを構築し、実験を行なった。flannel には、host-gw、vxlan、udp という 3 つの動作モードが用意されている。host-gw モード では、flanneld デーモンがキーバリューストアを参照して、pod にアクセスするために必要なルーティングテーブルを生成する。異なる Node 上で稼働している pod に対してパケットを送信する場合には、その pod が稼働している Node の IP アドレスをルーティングテーブルから割り出しその IP に対する MAC アドレス宛にフレームを送信する。この時 Destination IP は pod の IP アドレスである。

vxlan モードでは、flanneld は vxlan デバイス flannel.1 の

作成とルーティングテーブルの生成を行う。異なる Node 上の pod が通信する場合には、linux カーネルの vxlan 機能によりイーサネットフレームがカプセリングされ、pod が稼働する Node にフレームを送信される。パケットを受け取った Node はカプセリングしたヘッダを取り除いたのち、自身のルーティングテーブルに基づいて、pod との通信を成立させる。

udp モードにおいて、flanneld は tun デバイス flannel0 を作成と、ルーティングテーブルを生成する。tun デバイス flannel0 は flanneld デーモンと繋がっており、flannel0 に送られたパケットは flanneld によりカプセリングされ、適切なホストに送信される。カプセリングは IP パケットに対して行われる。

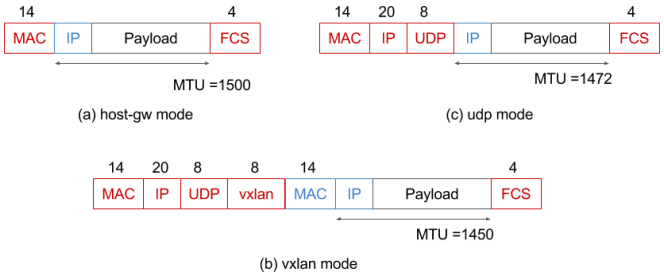

図 9　flannel 各モードでのフレームフォーマット

図 9 に、flannel のそれぞれの動作モードでのフレームフォーマットをまとめている。カプセリングを行わない時の MTU が 1500byte であると仮定し、カプセリングのヘッダが消費する byte 数も模式的に示している。

host-gw ではカプセリングを行わないのでイーサネットフレームのペイロードは 1500byte のままである。vxlan では vxlan のヘッダによりカプセリングを行うのでペイロードは 1450byte、udp では udp データグラムの中にパケットがカプセリングされるので、udp ヘッダのによる消費を差し引いた、1472byte がペイロードとなる。カプセリングの負荷、カプセリングがカーネル空間で行われるのかユーザー空間のプログラムによるものなのかなどの違いにより、flannel の 3 つのモードでは、Node のパケット転送性能が異なると予想される。

カプセリングを行わない host-gw モードは性能面で有利あるが、次の欠点がある。すなわち、Node 間に flannel のことを関知しないルータ、ゲートウェイが存在する場合には、pod 間で通信できないということである。

これは、クラウドプロバイダの場合に問題になることがわかっている。Table 1 に flannel のそれぞれの動作モードの利用可否について調査した結果を示す。

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

Table　1　flannel 動作モード、利用可否

GCP の場合には、VM（仮想マシン）には/32 のアドレスが付与され他の VM との通信はすべて、GCP のゲートウェイを通して行われる。AWS においては、同一サブネット内であれば VM 同士が直接通信することができるが、別サブネットにまたがる場合には、AWS のゲートウェイを通して通信することになる。これらのゲートウェイでは flannel により生成されルーティングルールを自動的に適用することは困難であり、トンネリング無しの host-gw モードを利用することは困難である。

我々はロードバランサの性能評価時に、これらの flannel の 3 つのモードの性能差についても比較を行う。

5.3　ネットワーク割り込み処理のハンドリング

近年 CPU がマルチコア化され計算機の性能がめざましく向上しているが、ネットワークのパケット処理がマルチコア化の恩恵を受けるためには、rss、rps[22,23]と呼ばれる技術を利用し、物理 NIC でのパケット受信時の割り込み処理とそれに続くプロトコル処理を複数のコアに分散させる必要があることが知られている。

ロードバランサの処理性のを比較するには、これらの技術を利用しすることでボトルネックを解消した上でベンチマークを行うことが必要である。我々は rss、rps を利用する場合と利用しない場合での、ロードバランサの性能比較を行いつつ一番いい条件で、ロードバランサ実装の優劣比較を行いたい。

以下では、今回の実験での rss と rps の設定方法についてまとめる。

実験に用いたハードウェアは Broadcom BCM5720 という NIC が搭載されており、この NIC には 4 つの rx-queue と一つの tx-queu が用意されている。図 10 に IRQ と NIC の Queue との対応関係を示す。

```
# cat /proc/interrupts |egrep eth|awk '{print $1,$19}'
81: eth0-tx-0
82: eth0-rx-1
83: eth0-rx-2
84: eth0-rx-3
85: eth0-rx-4
```

図 10　IRQ と NIC queue の関係

ⓒ 2017 Information Processing Society of Japan

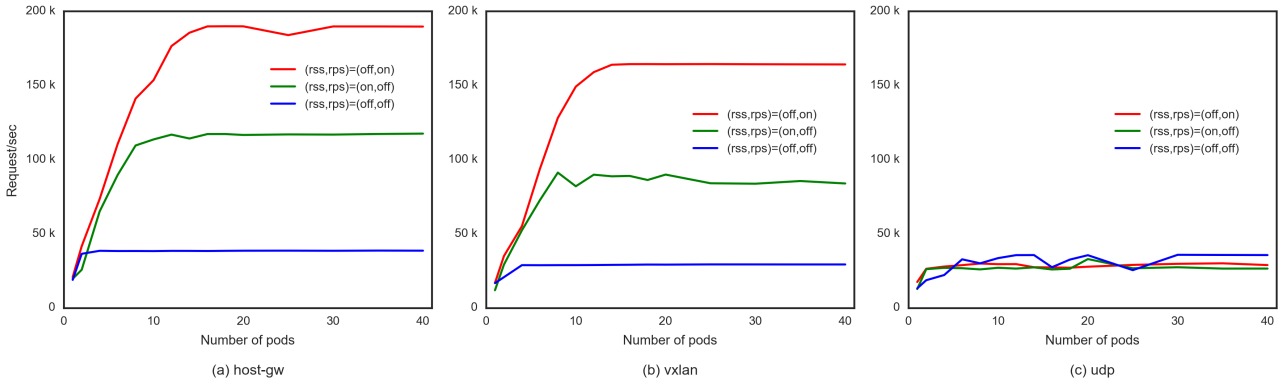

図 11 ipvs のベンチマーク結果

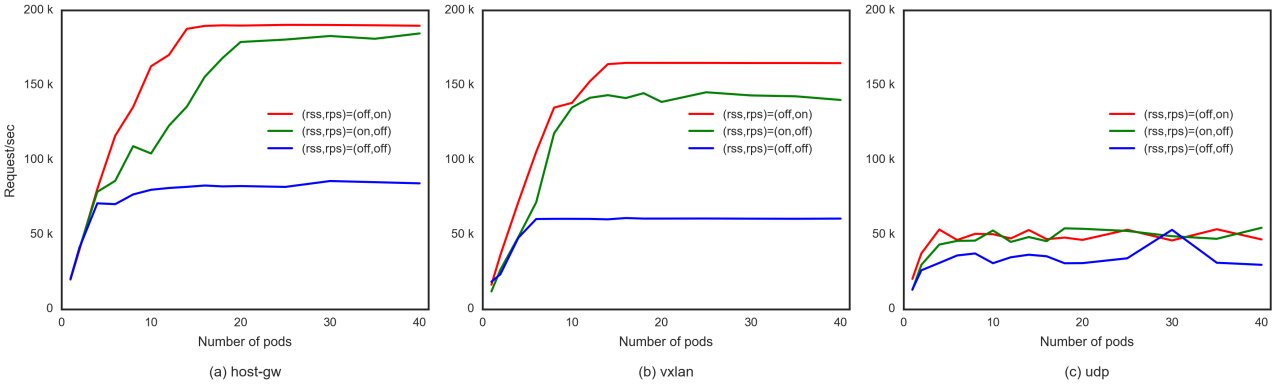

図 11 iptables のベンチマーク結果

これらの rx-queue にパケットが入って来ると、CPU のいずれかのコアに割り込みが発生し、その CPU が割り込みへの対応、 それに続くプロトコル処理を行う。[22]によると、どの CPU コアに割り込みを許可するかは、各 IRQ 番号に割り込み先として許可される CPU 番号のビットマップ値を設定することで指定することができる。例えば rx-queu: eth0-rx-1,2,3,4 からの割り込み先 CPU を CPU0,1,2,3 にするには、0001、0010、0100、100 に相当する数を以下のように設定すれば良い。

rss=on

```
echo 1 > /proc/irq/82/smp_affinity   # 0001 ->CPU0
echo 2 > /proc/irq/83/smp_affinity   # 0010 ->CPU1
echo 4 > /proc/irq/84/smp_affinity   # 0100 ->CPU2
echo 8 > /proc/irq/85/smp_affinity   # 1000 ->CPU3
```

本稿では、この設定を行った状態を、rss=on と呼ぶことにする。一方 rss=off は以下の設定を行った状態である。

rss=on

```
echo 1 > /proc/irq/82/smp_affinity   # 0001 ->CPU0
echo 1 > /proc/irq/83/smp_affinity   # 0001 ->CPU0
echo 1 > /proc/irq/84/smp_affinity   # 0001 ->CPU0
echo 1 > /proc/irq/85/smp_affinity   # 0001 ->CPU0
```

この場合、全ての割り込みは、CPU0 に対して行われる。

rps[22]は、ハードウェア割り込みが入った CPU からさらに別の CPU にソフトウェア割り込みをかけ、パケットの処理を別の CPU コアに肩代わりしてもらう仕組みである。

rps は NIC の割り込みキューが複数用意されていない場合や、今回のハードウェアのように CPU の物理コア数よりも少ない場合の性能向上に有効である。今回の実験では、次のように rps 設定を行った。

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
```

rps=on

```
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
```

fefe は 2 進数 1111 1110 1111 1110 であるから CPU0 と CPU8

以外の全ての論理 CPU に対してソフトウェア割り込み可能なことを意味している。

6. ベンチマーク結果とディスカッション

図 11 に本稿で提案する ipvs ベースのポータブルなロードバランサに対してのベンチマーク結果を示す。各グラフにおいて nginx の pod 数を横軸に、Request/sec を縦軸にプロットしている。Overlay Network に関しては flannel の 3 つの動作モード host-gw (図 11(a))、vxlan (図 11(b))、udp (図 11(c))に対して行なっており、それぞれの場合で、(rss, rps) = (off, off) , (on, off) , (off, on) の各条件で測定を行なっている。flannel udp モードを除いて、pod 数が 1-12,3 までの領域では pod 数の増加に従いロードバランサの性能がリニアに増加し、それ以降は飽和するという傾向が全体として見て取れる。図 11(a,b)の結果を見ると、(rss,rps)=(off, off)の場合には、著しく性能が悪く、ネットワークカードからホストがパケットを受けとる際には、マルチコアで分散処理を行うことが有効であるあることがわかる。(rss,rps)=(off, on)の場合には、intrrupt を CPU0 で受けた後、プロトコル処理を CPU1-7, CPU9-15 の 14 個の論理 CPU で分散処理しているのに対して、(rss,rps)=(on, off)の場合には intrrupt 処理とプロトコル処理を、CPU0,1,2,3 の 4 個の論理 CPU で行なっているので、後者では性能が低くなっていると考えられる。

Flannel の動作モードに関しては、カプセリングの無い host-gw モードの場合に性能が一番高く、Linux のカーネルの機能によりカプセリングを行う、vxlan モードがそれに続く。Flanneld がカプセリングを行う udp モードは著しく性能が低い。

図 12 に、既存の iptables ベースのロードバランス機能に対するベンチマーク結果を示している。Request/sec の値は worker pod の数が増えるに従いリニアに上昇して、ある時点で飽和するという傾向は、ipvs の場合と同様である。(rss,rps)の設定に性能が依存することも、その傾向も ipvs の場合と同様である。また flannel の動作モードに関しても、host-gw モードが一番高性能で vxlan モードがそれに続く。udp モードは著しく性能が低い。

図 13 に iptables、ipvs、nginx の性能を比較したグラフを示す。本稿で提案する ipvs ベースのポータブルなロードバランサは、既存の iptablesn によるロードバランス機能と比べ性能差がほとんどない。nginx ベースのポータブルなロードバランサは、これらと比べて著しく性能が低く、ワーカーpod が増えても全く Request/sec が向上しないので、今回の実験の範囲内では、ロードバランサとして利用する意義さえ疑われる。

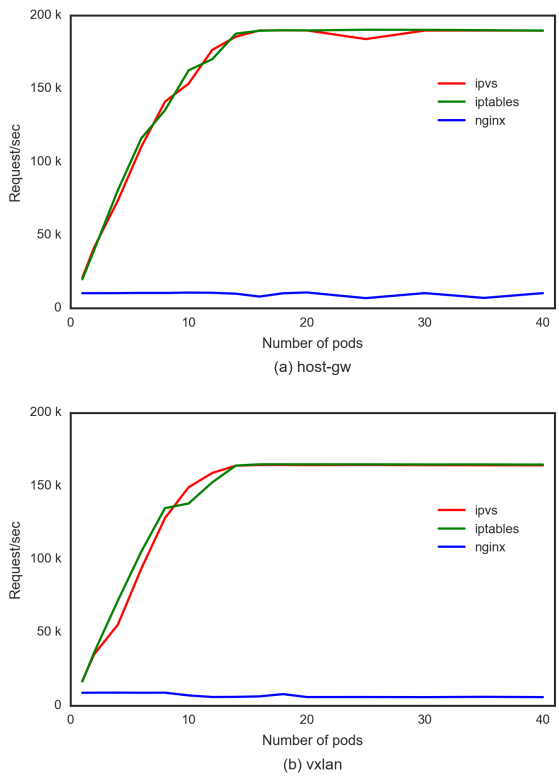

図 12 ipvs, iptables, nginx の性能比較
(rss,rps)=(off,on)

7. 結論

本稿では、異なるクラウドプロバイダ間で Kubernetes システムをマイグレーションするためのロードバランサの実装方式を提案した。提案方式の有効性をオンプレミスデータセンタ環境状に構築し、ロードバランサの性能評価を行なったところ、既存の Kubernetes に用意されている iptables による負荷分散性能と比べ遜色のない性能を維持しつつ、コンテナ化された、ポータブルなロードバランサが実現できることが明らかになった。また、オンプレミスデータセンタ、マルチクラウドで flannel を使う場合、vxlan モードが良いこと、ロードバランサの性能を引き出すには、マルチコア CPU での分散処理に対して気をつける必要があることなどがわかった。

参考文献

[1] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015, April). Large-scale cluster management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems (p. 18). ACM.
[2] “Create an External Load Balancer”. https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/
[3] “Federation”. https://kubernetes.io/docs/concepts/cluster-administration/federation/
[4] Zhang, W. (2000, July). Linux virtual server for scalable network

ⓒ 2017 Information Processing Society of Japan

services. In Ottawa Linux Symposium (Vol. 2000).
[5] “NGINX and NGINX Plus Ingress Controllers for Kubernetes Load Balancing”. https://www.nginx.com/blog/nginx-plus-ingress-controller-kubernetes-load-balancing/
[6] “NGINX Ingress Controllers”. https://github.com/nginxinc/kubernetes-ingress
[7] “kube-keepalived-vip” . hztps://github.com/kubernetes/contrib/tree/master/keepalived-vip
[8] http://www.keepalived.org/
[9] “GORB, Go Routing and Balancing”. https://github.com/kobolog/gorb
[10] “Cluster Frontend”. https://github.com/qmsk/clusterf
[11] “DOCKER 1.12: NOW WITH BUILT-IN ORCHESTRATION!”. https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/
[12] “Use swarm mode routing mesh”. https://docs.docker.com/engine/swarm/ingress/
[13] “Ingress Resources”. https://kubernetes.io/docs/concepts/services-networking/ingress/
[14] https://www.consul.io/
[15] https://coreos.com/etcd
[16] Linux-ip.net. (2016). 5.5.Destination NAT with netfilter (DNAT). http://linux-ip.net/html/nat-dnat.html
[17] https://github.com/ktaka-ccmp/ipvs-ingress
[18] “Linux Advanced Routing & Traffic Control HOWTO”. Chapter 11, http://www.tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.netfilter.html
[19] Will Glozer , “wrk - a HTTP benchmarking tool”. https://github.com/wg/wrk
[20] “flannel”. https://github.com/coreos/flannel
[21] “Backends”. https://github.com/coreos/flannel/blob/master/Documentation/backends.md
[22] Tom Herbert and Willem de Bruijn, “Scaling in the Linux Networking Stack”. https://www.kernel.org/doc/Documentation/networking/scaling.txt
[23]Tom Herbert (therWillem de Bruijn, “Scaling in the Linux Networking Stack”. https://www.kernel.org/doc/Documentation/networking/scaling.txt

ⓒ 2017 Information Processing Society of Japan