課題研究報告書

# パケットフィルタリングの iptables から BPF への移行妥当性調査

花家 研一

主指導教員 篠田 陽一 教授

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

2022 年 9 月

Abstract

There has been an explosive increase in the number of private IP addresses that a single web application can contain due to the proliferation of container technologies like Docker and container orchestration systems such as Kubernetes, an open-source software. The application layer in the Web 3-tier architecture consists of multiple nodes linked to load balancers, with multiple containers riding on each node. Containers communicate across nodes via virtual network interfaces assigned IP addresses in CIDR-allocated IP ranges in the virtual network space built across nodes. Container orchestration systems manage private IP addresses for tens of thousands of containers in an extensive application.

In this background, a virtual network space within physical nodes is divided using cgroups, a Linux kernel function. To prevent network threats, packet filtering must be implemented between physical nodes and containers—some Container Network Interfaces (CNIs) support packet filtering between containers. There is a long precedent for using iptables for packet filtering in Linux. Still, it is a performance bottleneck in cases with many filtering rules (tens of thousands), as in the current clustering situation.

The Berkeley Packet Filter (BPF) tool has newly begun to be used for packet filtering as a substitute. Though BPF has been used as a packet filtering tool for a long time, it has achieved significant progress recently. Its versatility and ease of implementation have attracted attention for packet filtering and in various other areas. This study investigates the validity of the shift in packet filtering technology by examining iptables and BPF from various perspectives.

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

単一の Web アプリケーションが内包する プライベート IP アドレスの数は Docker[1] を代表とするコンテナ技術と オープンソースのソフトウェアである Kubernetes[2] を代表とするコンテナオーケストレーションシステムの普及により爆発的に増えた。Web 3 層アーキテクチャにおけるアプリケーション層には複数のノードがロードバランサーに紐付いており、ノード内には複数のコンテナが乗り、コンテナはノードを跨いで構築された仮想ネットワーク空間にて CIDR によって区切られた IP レンジの IP アドレスが割り当てられた仮想ネットワークインターフェースを介してノードを跨いだコンテナ間の通信を行っている。巨大なアプリケーションともなるとコンテナの数は何万という数となり、その分の プライベート IP アドレスをコンテナオーケストレーションシステムは管理している。

このような背景から物理ノード内には Linux のカーネル機能である cgroups[3] によって分断された仮想ネットワーク空間が存在しており、ネットワーク脅威防止のためのパケットフィルタリングは物理ノード間のみならずコンテナ間においても実装が求められ、いくつかのコンテナネットワークインターフェース（CNI）[4] はコンテナ間のパケットフィルタリングに対応している。一方で Linux におけるパケットフィルタリングの技術は古くから iptables[5] が用いられてきたが、昨今のクラスタ事情のようにフィルタリングのルールが何万という膨大になるケースにおいてはパフォーマンスのボトルネックとなることがわかった。

そこで代替技術として活用され始めたのが Berkeley Packet Filter（BPF）[7] によるパケットフィルタリングである。BPF 自体はパケットフィルタリング技術として古くから使われてきた技術ではあるが、昨今目覚ましい進化を遂げており、その汎用性、実装容易性からパケットフィルタリングのみならず様々な面で活用の場を広げている昨今注目されている技術の一つである。本研究では iptables と BPF を多角的に考察することでパケットフィルタリング技術の移行妥当性の調査を行う。

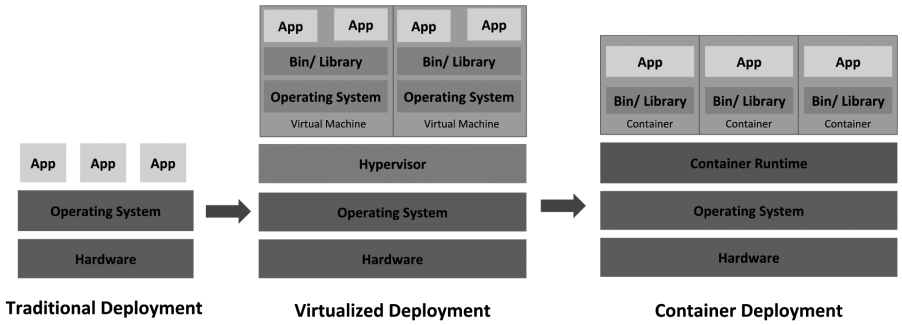
# 第2章　研究の背景と目的

本章では、デプロイ技術の変化とそれに伴い顕在化してきた iptables の問題点を上げた上で本研究の目的を述べる。

## 2.1　研究の背景

Linux のパケットフィルタリング技術として長く活用されてきた iptables であるが、図 2.1 のように OS 上に直接アプリケーションが稼働している形から Container Runtime 上にコンテナとしてアプリケーションが稼働している昨今の Web アプリケーション構成においてはパフォーマンスにおけるボルトネックとなるケースが散見され始めている。BPF は iptables の代替となりうるツールであり、Kubernetes の CNI である Cilium[9] では BPF をネットワークの制御に活用している。

一方で Kubernetes のコンポーネントである kube-proxy[10] ではプロキシ及び、ロードバランサーの機能を iptables で実現しており、Kubernetes を活用している多くの web アプリケーションが未だに iptables に依存しているが、BPF 自体に対する知見が広く広まっておらず、BPF へ移行することのメリットが広く浸透していないという背景が存在している。



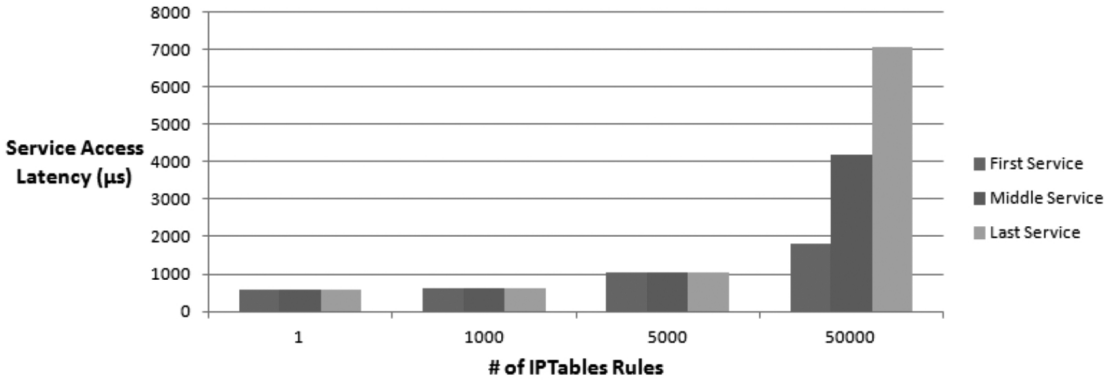

Source: Kubernetes とは何か？[8]

図 2.1: デプロイ技術の推移

### 2.1.1 現代における iptables の問題点

iptables はインターネットの速度が現在と比較して非常に低速だった時代に開発されたため、受信したパケットがリストにマッチするかを逐次判断するため、リストが大きくなればなるほどオーバーヘッドがリニアに増加するという課題を抱えている。現代における web アプリケーションは 20 年前と比較して非常に多くの IP アドレスを内包するため、リストの大きさに伴い、リニアにオーバーヘッドが増加する iptables のデメリットが顕著になってきた。

こうした問題に対応するため、IP を集合で管理することのできる ipset[11] というツールが開発されたが、すべての問題を解決することはできていない。例えば、kube-proxy では、service に対するロードバランサーの機能を iptables の DNAT(Destination Network Address Translation) を用いて実現しているが、複数の iptables rules はそれぞれの service にインストールされ、記述される rule は service を横断した rule になるため、全体で管理しなければならない iptables rules のリストの数は service の数に応じて指数関数的に増える。KubeCon Europe 2017 での Scale Kubernetes to support 50000 services [12] によると 2 万の service に対して、16 万の iptables rules を適用した場合に変更完了まで 5 時間かかり、図 2.2 のように Rules 数に伴い Latency が増加すると報告がされている。




Source: Scale Kubernetes to support 50000 services[12]

図 2.2: IPTables Service Routing Perormance

## 2.2 目的

本研究ではiptablesとBPFの差異を多角的に調査し、BPFの知識の浸透を図るとともにBPFに移行することのメリット・デメリットを示すことで、実際に現場でリファランスとして活用されることを目的としている。

## 2.3 本研究の社会的な意義

BPFやKubernetes単体を扱ったリファランスは数多く存在するが、iptablesとKubernetesの関係性や、なぜ現代においてBPFが活用されはじめているかを包括的に扱った日本語の文書は存在せず、本研究の成果は複雑かつ大規模なwebアプリケーションを運用する上で社会的な意義がある研究と考える。

# 第3章　関連技術

本章では、本研究に関連する技術について説明を行う。

## 3.1　iptables

iptables とは Linux におけるパケットフィルタリング及び NAT の管理ツールである。実際の機能は Netfilter[13] にて実装されており、iptables は Netfilter を操作するツールにあたる。（Firewalld[14] も同様に Netfilter を操作するツールにあたる）iptables は長い間 Linux のパケットフィルタリングツールとして使われている。Linux カーネル 2.4 より Netfilter を操作するツールとして組み込まれ、それ以前にパケットフィルタリングとして使われていた ipchains[15] を置き換えるツールである。

### 3.1.1　iptables の構成

iptables では 図 3.1 のように Iptables, Tables, Chains, Rules のような階層構造を採用しており、テーブルには Filter Table, Nat Table, Mangle Table, Raw Table がある。

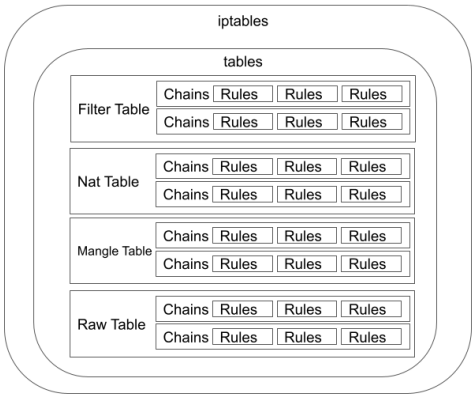



図 3.1: iptables の階層

**Filter Table**

Filter Table は INPUT chain, OUTPUT chain, Forward chain にて構成されており、受信・送信・転送といったパケットのルールを制御している。

**Nat Table**

Nat Table は、PREROUTING chain, POSTROUTING chain, OUTPUT chain にて構成されており、IP アドレスの変換ルールを制御している。

**Mangle Table**

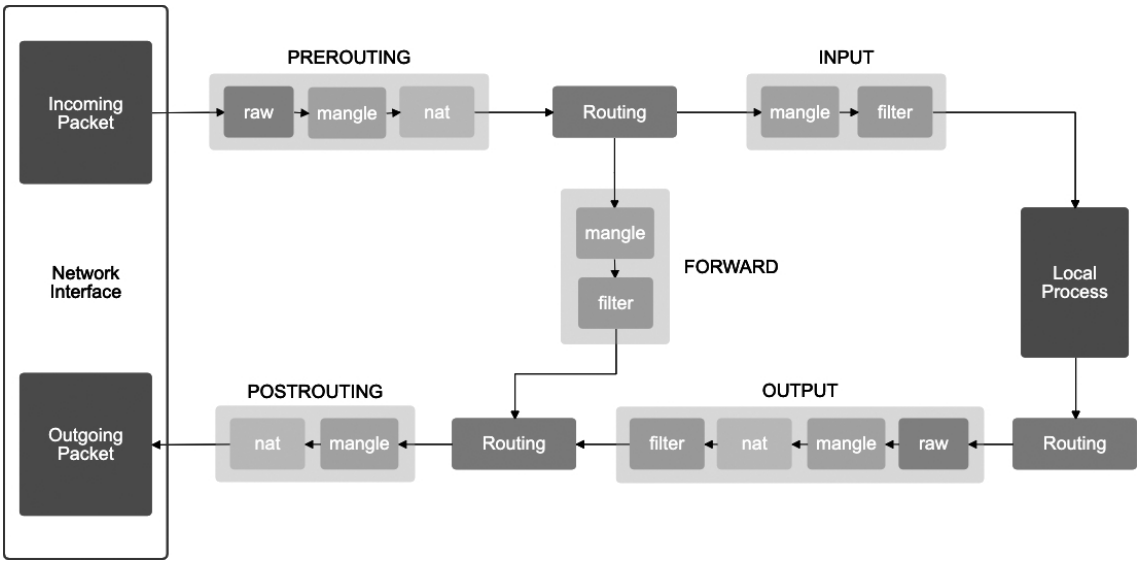
Mangle Table は、PREROUTING chain、INPUT chain、FORWARD chain, OUTPUT chain、POSTROUTING chain にて構成されており、QoS（Quality of Service) を実現するために用いられる。

**Raw Table**

Raw Table は、PREROUTING chain, OUTPUT chain にて構成されており、パケットがコネクショントラッキングシステムに追跡されないようにマークをつけるために用いられる。

### 3.1.2 処理フロー

Chains は処理の順番が決まっており、図 3.2 のように受信したパケットは最初に PREROUTING chain で処理され、ルーティングの後にローカルホスト向けではない場合は FORWARD chain で処理がされ、最後に POSTROUTING chain が処理されるというフローとなっている。

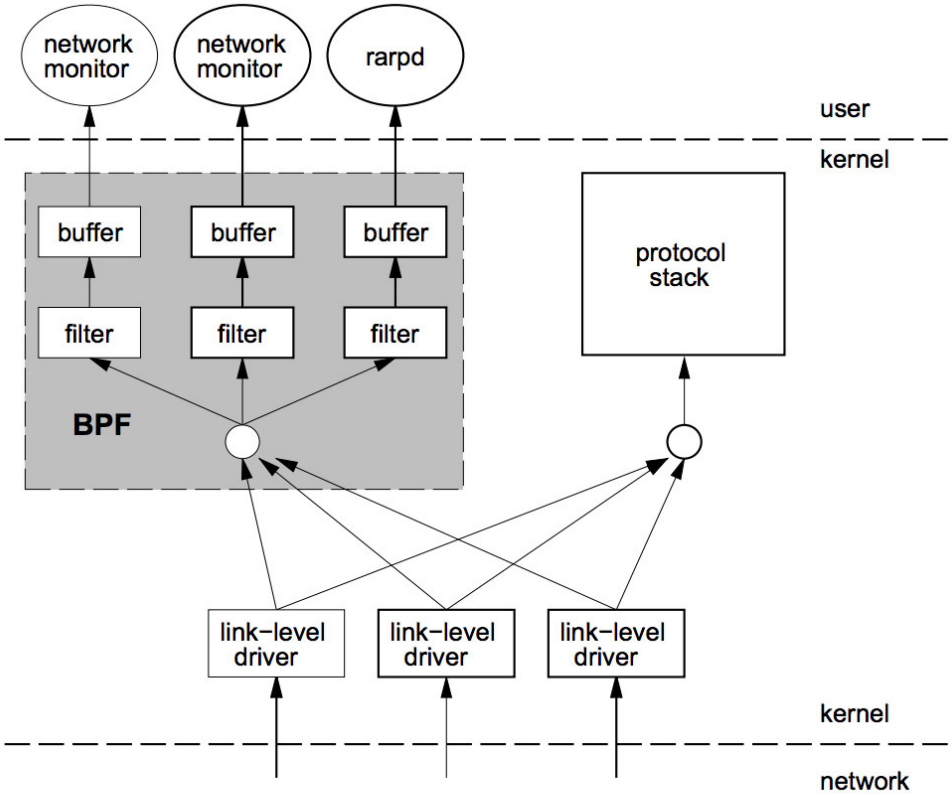



Source: iptables の仕組みを図解 [6]

図 3.2: iptables の処理フロー

## 3.2 BPF

BPF の歴史は古く、Steven McCanne と Van Jacobson による The BSD Packet Filter:A New Architecture for User-level Packet Capture [16] にて 1992 年に提唱された。BPF の当初の利用用途はその名の通り、パケットフィルタリングであるが、ユニークな実装方法で実現をしている。BPF を用いない場合においてパケットフィルタリングを行うためには、カーネル空間にてネットワークデバイスから受け取ったパケットをユーザー空間にて稼働するフィルタリングプログラムに渡すためにコピーが行われ、コピーされたパケットを元にユーザー空間にてフィルタリングが行われるが、BPF を用いた場合は、図 3.3 のようにユーザー空間にて動作するプログラムからインストラクションを BPF 仮想マシンに送り、インタープリターを介してカーネル空間にてパケットのフィルタリングが行われるため、空間を跨いだパケットのコピーが発生せずパフォーマンスの向上を図ることができる。




Source: The BSD Packet Filter: A New Architecture for User-level Packet Capture[16]

図 3.3: BPF Overview

### 3.2.1 BPF の仮想マシン

当初の BPF 仮想マシンは 32bit の レジスタが２つと 16 x 32 bit のメモリ領域、プログラムカウンタが１つで構成され、フィルタリングという限定的な用途での利用が想定されていた。

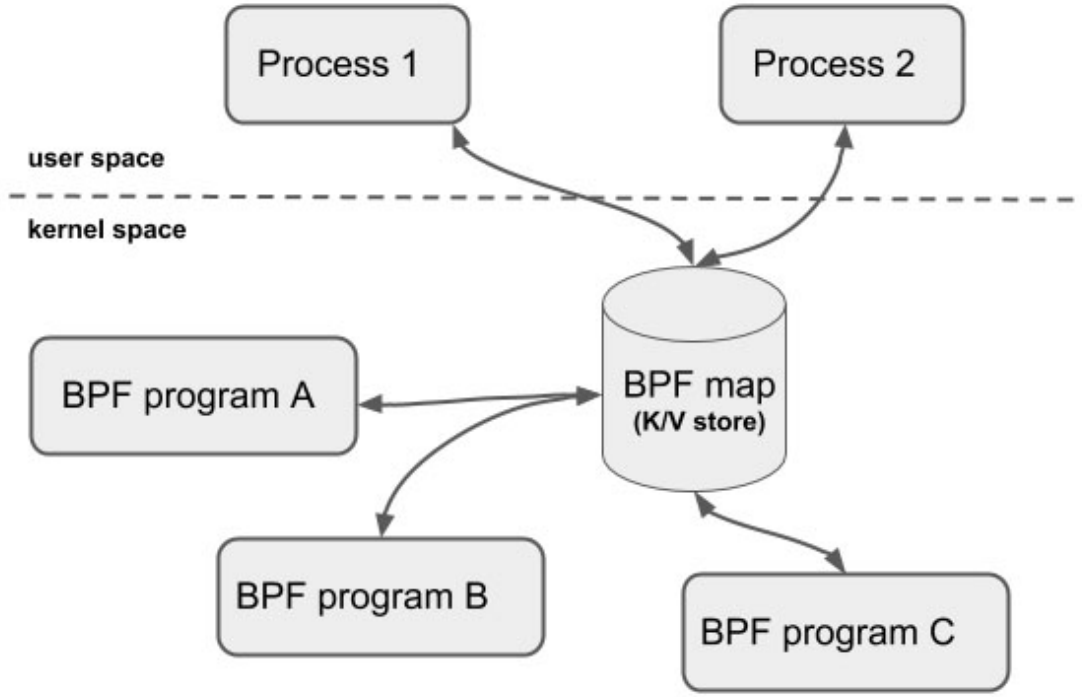
表 3.1: BPF 仮想マシンの構成

| Element | Description |
|---|---|
| A | 32 bit wide accumulator |
| X | 32 bit wide x register |
| M[] | 16 x 32 bit wide misc registers aka ”scratch memory store” |

大きな変化があったのは 2013 年に Alexei Starovoitov より提唱された extended BPF [17] に端を発する。Alexei の提唱した extended BPF ではレジスタの数は 2 から 10 に、レジスタ幅は 32bit から 64bit に、メモリ領域の代わりに 512byte のスタック領域と BPF maps と呼ばれるストレージを使い、制限付きでカーネル機能を使用可能にするなど、従来の BPF) から大幅に変更が加えられ、2014 年に Linux 3.15 にてリリースされている。この変更により BPF はネットワーキング、オブザーバビリティ、セキュリティといった汎用的な用途で利用可能な仮想マシンへと変貌を遂げた。

### 3.2.2 BPF maps

BPF maps とは、カーネル空間に存在するキーバリューストアであり、空間を跨いだ状態を維持するために活用される。BPF maps は 図 3.4 のように BPF プログラム、ユーザー空間のファイルディスクリプタからアクセスすることが可能であり、他の BPF プログラムやユーザースペースのアプリケーションと共有することも可能である。map は用途に応じて様々な種類がカーネルにて用意されており、v5.18 では Listing3.1 が用意されている。




Source: BPF and XDP Reference Guide[18]

図 3.4: BPF Map

Listing 3.1: BPF map type

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
        BPF_MAP_TYPE_LPM_TRIE,
        BPF_MAP_TYPE_ARRAY_OF_MAPS,
        BPF_MAP_TYPE_HASH_OF_MAPS,
        BPF_MAP_TYPE_DEVMAP,
        BPF_MAP_TYPE_SOCKMAP,
        BPF_MAP_TYPE_CPUMAP,
        BPF_MAP_TYPE_XSKMAP,
        BPF_MAP_TYPE_SOCKHASH,
        BPF_MAP_TYPE_CGROUP_STORAGE,
        BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
        BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
        BPF_MAP_TYPE_QUEUE,
        BPF_MAP_TYPE_STACK,
        BPF_MAP_TYPE_SK_STORAGE,
        BPF_MAP_TYPE_DEVMAP_HASH,
        BPF_MAP_TYPE_STRUCT_OPS,
        BPF_MAP_TYPE_RINGBUF,
        BPF_MAP_TYPE_INODE_STORAGE,
        BPF_MAP_TYPE_TASK_STORAGE,
        BPF_MAP_TYPE_BLOOM_FILTER,
};
```

### 3.2.3 BPF の利点

BPF を活用することの利点としては大きく以下の 5 点が挙げられる。

- カーネル空間で動作するプログラムをカーネル空間、ユーザー空間の境界を意識することなく記述することができる。例として、ロードバランシングやパケットフィルタリングといったネットワークに関するプログラムは BPF maps で状態を保持することで、空間を跨いだパケットの移動が発生せず、パフォーマンスの向上を図ることができる。
- 求めるユースケースに不必要なコンパイルなどをすることなく、実行が可能である。例として、コンテナが IPv6 のみを受け付ける状態にしたい場合などにおいても、BPF を活用することで IPv4 を排除するためだけのプログラムを記述することができる。
- eXpress Data Path(XDP)[19] といったネットワーキングの場合、BPF はカーネルやコンテナの再起動といったことをすることなく、アトミックに更新することが可能である。
- BPF はユーザー空間に対して stable な Application Binary Interface(ABI) を 提供しており、システムコールと同様に動作が保証されているため、ポータビリティに優れている。
- BPF のプログラムは既存のカーネルインフラストラクチャやツールを活用して動作しており、安全性が保証されている。またカーネルモジュールと違い、カーネルがクラッシュしないための検査機構が備わっている。

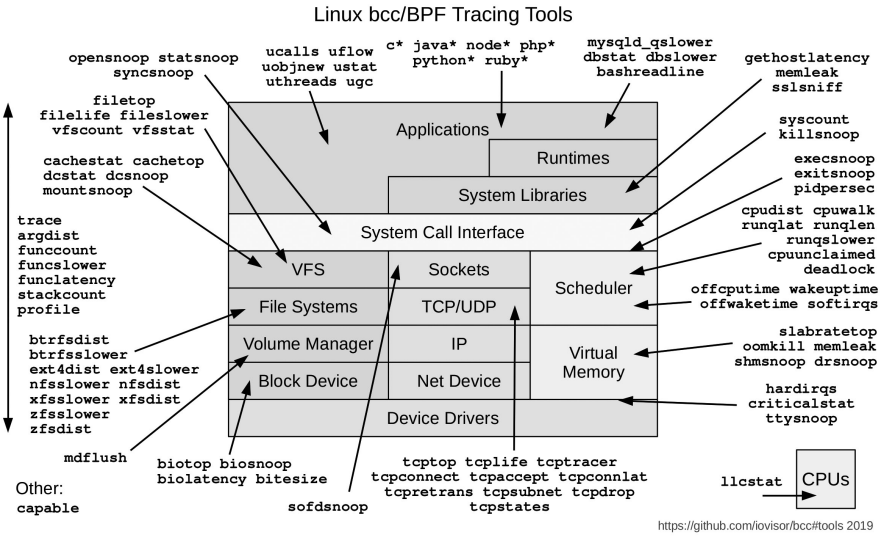
### 3.2.4 BPF が活用可能なツール

BPF には BPF Compiler Collection(BCC) [20] と呼ばれるオープンソースのツールキットが存在し、BPF をプログラミングする上で有益な多数のサンプルやツールが内包されている。BCC はパッケージマネジャーからインストールすることができ、Ubuntu の場合だと以下のコマンドでインストールすることができる。

Listing 3.2: BCC install command for Ubuntu

```
1 sudo apt-get install bpfcc-tools linux-headers-$(uname -r)
```

BCC には図 3.5 のように多数のツールがサンプルとして内包されている。




Source: GitHub - iovisor/bcc: BCC[20]

図 3.5: BPF Tracing

例えば system call である execve[21] のトレースを行うためには、以下の例のように execsnoop を用いることでトレースすることができる。

Listing 3.3: execsnoop

```
1 root@ubuntu:/sbin# execsnoop-bpfcc
2
3 PCOMM PID PPID RET ARGS
4 ps 55931 52711 0 /usr/bin/ps
5 ls 55932 52711 0 /usr/bin/ls --color=auto
```

## 3.3 コンテナ

Linux におけるコンテナとはカーネルの機能である cgroups や namespace[22] を使い、システムリソースを論理的にプロセスに分配し、名前空間 (ユーザー ID、プロセスツリーなど) をプロセスに独自に設定することでホストマシーンとは隔離された仮想的な空間を作り上げる技術のことである。仮想化には 図 3.6 のようなハイパーバイザー型と　図 3.7 コンテナ型の技術が存在しており、ハイパーバイザー型による仮想化の仕組みではホストマシーン上のホスト OS にハイパーバイザーが稼働しており、仮想的なハードウェアをエミュレートすることで仮想化を行うが、コンテナ型による仮想化では、ハイパーバイザーの代わりにコンテナエンジンを介して論理的に隔離されたリソースを用いて仮想化を行う。また Linux カーネルをホスト OS と共有するため OS をすべてエミュレートするハイパーバイザー型の仮想化と比べてオーバーヘッドが少ないという利点がある。一方で Linux カーネルを用いない OS の仮想化は出来ない点や一部の機能に制限が加わるなどの欠点も存在する。


図 3.6: HyperViser

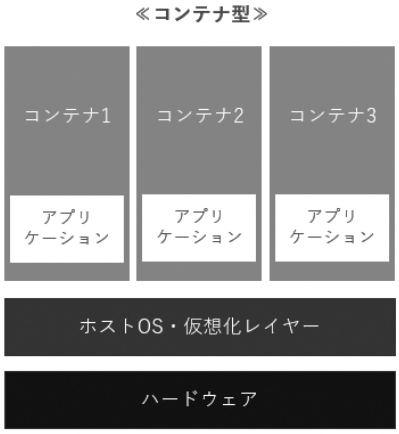



図 3.7: Container

Source: GMO クラウドアカデミー [23]

コンテナの実行はコンテナランタイムが用いられ、コンテナランタイムは大きくハイレベルとローレベルに機能が分けられている。ローレベルではコンテナの作成と起動を担い、代表的なものとして lxc[24], runc[25], containerd[26] などが存在し、Linux Foundation プロジェクトの一つである、Open Container Initiative（OCI）[27] というイニシアチブにて、業界標準策定がなされている。ハイレベルではコンテナイメージのフォーマット定義やイメージの作成・管理などを担い、代表的なものとして rkt[28], Docker などが存在する。

ハイレベルランタイムである Docker では Dockerfile と呼ばれるコンテナの雛形を用いてコンテナ環境を作り出す。Listing3.4 は Node.js[29] をコンテナで動かすための Dockerfile であり、Docker Hub というレポジトリからベースとなる node:12-alpine を Pull し、Debian 系のパッケージマネジャーである apt を利用し、追加のパッケージをインストールした後に yarn install –prodcution コマンドにて依存パッケージの fetch し、index.js に記述されたソースコードを元に Node をコンテナ環境で起動する。Docker がインストールされている環境であればホストマシーンの環境に依存せずに、Docker Hub という Docker image レポジトリを介して 記述通りのコンテナ環境を作り出せる点は大きな利点と言えるだろう。

Listing 3.4: dockerfile

```
1 # syntax=docker/dockerfile:1
2 FROM node:12-alpine
3 RUN apk add --no-cache python2 g++ make
4 WORKDIR /app
5 COPY . .
6 RUN yarn install --production
7 CMD ["node", "src/index.js"]
8 EXPOSE 3000
```

## 3.4 Kubernetes

Kubernetes とは可搬性、拡張性に優れたコンテナオーケストレーションシステムであり、オープンソースでの開発が行われている。Kubernetes は 2014 年に Google から発表されたが、Google の内部プラットフォームであった Borg から強い影響をうけている。コンテナ技術の普及によりアプリケーションのデプロイは OS 上にプロセスを動作させる形からホスト OS 上のコンテナランタイムにてホスト OS から分断された環境でのデプロイへと移行が進んだ。

一方で、Kubernetes のデフォルトプロキシである kube-proxy ではパケットフィルタリングや DNAT を iptables に依存する形で提供しているため、前述のルール数増加による性能低下の影響を受けやすく、大規模なアプリケーションを運用する上でボトルネックとなっている。

Kubernetes ドキュメント [8] によるとコンテナをつかったデプロイは以下のメリットがあると説明されている。

- アジャイルアプリケーションの作成とデプロイ: VM イメージの利用時と比較して、コンテナイメージ作成の容易さと効率性が向上する。
- 継続的な開発、インテグレーションとデプロイ: 信頼できる頻繁なコンテナイメージのビルドと、素早く簡単にロールバックすることが可能なデプロイを提供する。(イメージが不変であれば)
- 開発者と運用者の関心を分離: アプリケーションコンテナイメージの作成は、デプロイ時ではなく、ビルド/リリース時に行う。それによって、インフラストラクチャとアプリケーションを分離する。
- 可観測性は OS レベルの情報とメトリクスだけではなく、アプリケーションの稼働状態やその他の警告も表示する。
- 開発、テスト、本番環境を越えた環境の一貫性: クラウドで実行させるのと同じようにノート PC でも実行させる事ができる。
- クラウドと OS ディストリビューションの可搬性: Ubuntu、RHEL、CoreOS 上でも、オンプレミスも、主要なパブリッククラウドでも、それ以外のどんな環境でも、実行できる。
- アプリケーション中心の管理: 仮想マシン上で OS を実行するから、論理リソースを使用して OS 上でアプリケーションを実行するへと抽象度のレベルを向上させる。
- 疎結合、分散化、拡張性、柔軟性のあるマイクロサービス: アプリケーションを小さく、同時にデプロイと管理が可能な独立した部品に分割される。1 台の大きな単一目的のマシン上に実行するモノリシックなスタックではない。
- リソースの分割: アプリケーションのパフォーマンスが予測可能。
- リソースの効率的な利用: 高い効率性と集約性が可能。

### 3.4.1 CNI

CNIとは正式名称をContainer Network Interfaceといい、CNCF [30] のプロジェクトの一つであり、Kubernetesのコンポーネントであるkubelet[31]はCNIに則ったAPIを利用してPluggableなネットワークプラグインを呼び出す。Kubernetesにおける Node 間の通信はCNI Pluginによって実現されており、その手段はCNIによって実装方針が異なる。代表的なCNIとしてflannel[32]、Calico[33]、Ciliumが存在するが、Ciliumは当初からBPFを用いることでパケットフィルタリング、ビジビリティといった多彩な機能を備えており、Calicoも後続ではあるが、BPFを活用を積極的に進めている。

### 3.4.2 Kubernetesオブジェクト

Kubernetesにはコンテナを管理する上でいくつかの抽象化されたオブジェクトが用意されている。本章では代表的なKubernetesのオブジェクトを説明する。

#### Pod

Podとはコンテナの集合体であり、Kubernetesで管理できる詳細のデプロイ可能ユニットとなる。Pod上には複数コンテナを配置することが可能となっており、例えばアプリケーション実行コンテナとサイドカーコンテナとしてログ収集コンテナを1 Podに乗せるといった構成も可能となっている。また、Podは後述するDeploymentやStatefulSetといったワークロードリソースにて作成が行われる。

#### ReplicaSet

ReplicaSetとは宣言したPod数を維持するためのワークロードリソースである。ReplicaSetの対象となるセレクターやレプリカ数などを定義することで使用が可能となる。 Listling 3.5 の例では replicas にて必要なPod数が定義されており、spec にて実際に必要なコンテナの定義（ここではgcr[34]が指定されているため、imageが存在しない場合は自動的にgcrからダウンロードされる）、requests にてPodに割り当てるリソースが定義されている。ReplicaSetの適用は Listling 3.6 のようにkubectlコマンド経由で行うことができる。

Listing 3.5: ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

Listing 3.6: ReplicaSetApply

```
kubectl apply -f http://k8s.io/examples/controllers/frontend.yaml
```

### Deployment

Deployment とは Pod と ReplicaSet の管理を行うワークロードリソースである。Deployment は ReplicaSet を管理することでロールアウトやロールバックといったリビジョン管理が可能なリソースとなっている。 Listling 3.7 の例は nginx[35] コンテナを 3pod 宣言しており、apply することによって Deployment、ReplicaSet、Pod が作成される。 Deployment のリビジョンを確認する場合は、 Listling 3.8 のように kubectl rollout history コマンドで確認することができ、任意のバージョンにロールバックすることが可能となっている。

Listing 3.7: Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Listing 3.8: DeploymentHistory

```
kubectl rollout history deployment.v1.apps/nginx-deployment

deployments "nginx-deployment"
REVISION CHANGE-CAUSE
1 kubectl apply --filename=https://k8s.io/examples/controllers/
    nginx-deployment.yaml
2 kubectl set image deployment.v1.apps/nginx-deployment nginx=
    nginx:1.16.1
3 kubectl set image deployment.v1.apps/nginx-deployment nginx=
    nginx:1.161
```

# 第4章　iptablesからBPFへの移行

BPF は他のツールを組み合わせることで iptables の抱える問題点を解決可能な代替技術としてのみならずオブザーバビリティ、セキュリティといった様々な分野での活躍が期待される技術である。故に本章では iptables から BPF へのパケットフィルタリング技術移行を前提として iptables との比較を示す。

## 4.1　BPFによるパケットフィルタリング

現時点において iptables のすべての機能を代替することが可能な BPF ツールは存在しないが、XDP に BPF を Hook させることで INPUT chain の代用とする方法や Traffic Control(tc)[36] のレイヤーに BPF を Hook させることで OUTPUT chain の代用とするといった活用が広まっている。現に Cilium ではパケットの処理に XDP と tc を BPF に Hook させており、Daniel Borkmann と Alexei Starovoitov によって開発中の次世代 Linux firewall として期待されている bpfilter [37] においても Hook ポイントとして INPUT に XDP、 OUTPUT に tc を採用している。

### 4.1.1　XDPでのINBOUNDフィルタリング

XDP とは BPF ベースのデータパスであり、sk_buff 生成前のネットワーキングドライバに限りなく近い場所で BPF のプログラムを走らせることでパケットの転送や破棄といった処理を高速で行うことができる。XDP は受信したパケットに対して BPF プログラムを実行することができるため、iptables における INPUT chain の代替として使用することができ、パケット処理までのオーバーヘッドの少なさから DDoS 攻撃やロードバランサーとして活躍が期待されている。一方で XDP は NIC もしくはドライバのサポートが必須となるため、XDP に対応していない NIC およびドライバでは動作しないというデメリットも存在する（Generic XDP と呼ばれる sk_buff 生成後に BPF プログラムを実行する機能においてはこの制約はなくなるがパフォーマンスは落ちる）。

Listing4.1 は ICMP を XDP で破棄するコードの例である。このコードではプロトコル番号が１のパケットのみ破棄を行っており、破棄と共に BPF Map の値を

インクリメントすることで破棄した回数をカウントしている。ユーザースペースでは１秒毎にBPF Mapから値を取得し、出力している。

**Listing4.1 コード詳細**

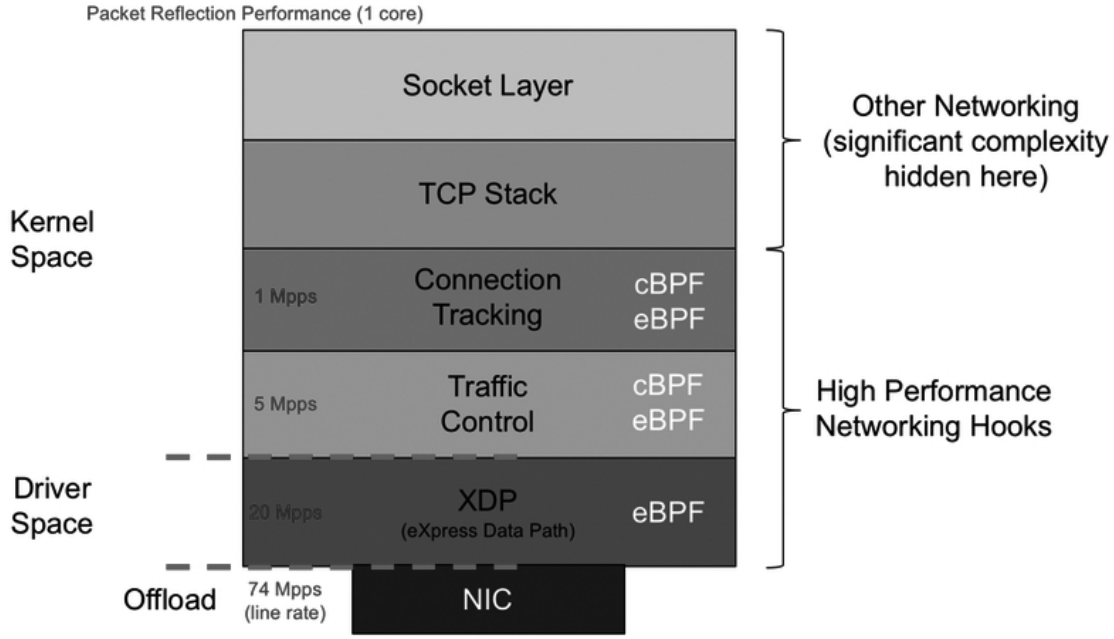
- Line 1-24: BPF コード
- Line 19-23: プロトコル番号が１ (ICMP) のパケットを破棄 (XDP_DROP) し、map の値をインクリメント
- Line 34: XDP にアタッチ
- Line 35: BPF Maps からテーブルオブジェクトを取得
- Line 36-43: １秒間隔で破棄されたパケットの数を標準出力

Listing 4.1: samplexdp

```
bpf_text = """
int xdp_drop_icmp(struct xdp_md *ctx) {
  void* data_end = (void*)(long)ctx->data_end;
  void* data = (void*)(long)ctx->data;
  struct ethhdr *eth = data;
  u32 value = 0, *vp;
  u32 protocol;
  u64 nh_off = sizeof(*eth);

  if (data + nh_off > data_end)
    return XDP_PASS;

  if (eth->h_proto == htons(ETH_P_IP)) {
    struct iphdr *iph = data + nh_off;
    if ((void*)&iph[1] > data_end)
      return XDP_PASS;

    protocol = iph->protocol;
    if (protocol == 1) {
      vp = dropcnt.lookup_or_init(&protocol, &value);
      *vp += 1;
      return XDP_DROP;
    }
  }

  return XDP_PASS;
}
"""

b = BPF(text=bpf_text)
fn = b.load_func("xdp_drop_icmp", BPF.XDP)

device = sys.argv[1]
b.attach_xdp(device, fn)
dropcnt = b.get_table("dropcnt")
while True:
  try:
    dropcnt.clear()
    time.sleep(1)
    for k, v in dropcnt.items():
      print("{} {}: {} pkt/s".format(time.strftime("%H:%M:%S"),
          k.value, v.value))
  except KeyboardInterrupt:
    break

b.remove_xdp(device)
```

### 4.1.2 tc での OUTBOUND フィルタリング

BPF は XDP 以外にもカーネルの tc(traffic control) レイヤーに Hook をしかけることもできる。XDP では受信したパケットのみに BPF プログラムを走らせることができたが tc では受信、送信両方のパケットに BPF プログラムを走らせることが可能である。一方で tc は XDP と異なり、 sk_buff に対して処理を行うため XDP と比較して実行速度が劣るというデメリットが存在する。また図 4.1 のように XDP で処理が走った後に処理がなされる。Listing4.2 は ICMP を tc で破棄するコードの例である。基本的には Listing4.1 における XDP の例と同様の構造となっている。相違点としては、XDP の場合パケットの破棄では XDP_DROP を返却していたのに対して、tc では TC_ACT_SHOT となっている点や BPF にアタッチする部分が XDP では attach_xdp 関数を呼び出していたのに対して、IPRoute 経由でのアタッチとなっている（ここで Inbound か Outboud かの指定も行う）。また引数として扱うデータ型も XDP の場合は xdp_md であったのに対して sk_buff となっており扱えるフィールドの数が増えている。




Source: Netronome Blog[38]

図 4.1: パケットフィルタリング階層

**Listing4.2 コード詳細**

- Line 1-27: BPF コード
- Line 18-23: プロトコル番号が１ (ICMP) のパケットを破棄 (TC_ACT_SHOT) し、map の値をインクリメント
- Line 34: tc にアタッチ
- Line 36: BPF Maps からテーブルオブジェクトを取得
- Line 37-44: １秒間隔で破棄されたパケットの数を標準出力

Listing 4.2: sampletc

```
bpf_text = """
BPF_HASH(dropcnt, u32, u32);
int tc_drop_icmp(struct __sk_buff *skb) {
  void* data_end = (void*)(long)skb->data_end;
  void* data = (void*)(long)skb->data;
  struct ethhdr *eth = data;
  u64 nh_off = sizeof(*eth);

  if (data + nh_off > data_end)
    return TC_ACT_OK;

  if (eth->h_proto == htons(ETH_P_IP)) {
    struct iphdr *iph = data + nh_off;
    if ((void*)&iph[1] > data_end)
      return TC_ACT_OK;
    u32 protocol;
    protocol = iph->protocol;
    if (protocol == 1) {
      u32 value = 0, *vp;
      vp = dropcnt.lookup_or_init(&protocol, &value);
      *vp += 1;
      return TC_ACT_SHOT;
    }
  }
  return TC_ACT_OK;
}
"""
try:
    b = BPF(text=bpf_text, debug=0)
    fn = b.load_func("tc_drop_icmp", BPF.SCHED_CLS)
    idx = ipr.link_lookup(ifname=device)[0]

    ipr.tc("add", "clsact", idx)
    ipr.tc("add-filter", "bpf", idx, ":1", fd=fn.fd, name=fn.
       name, parent=EGRESS, classid=1,direct_action=True)

    dropcnt = b.get_table("dropcnt")
    while True:
      try:
        dropcnt.clear()
        time.sleep(1)
        for k, v in dropcnt.items():
          print("{} {}: {} pkt/s".format(time.strftime("%H:%M:%S
              "), k.value, v.value))
      except KeyboardInterrupt:
        break
```

### 4.1.3 Kubernetes でのフィルタリング

Kubernetes における BPF を用いたパケットフィルタリングは Cilium を kube-proxy の代わりに利用することで可能となる。kube-proxy の代わりに Cilium を使用する場合は Listing4.3 のように Kubernetes 用パッケージマネージャーである helm [39] 経由でインストールすることが可能である。

Listing 4.3: cilium

```
kubeadm init --skip-phases=addon/kube-proxy

helm repo add cilium https://helm.cilium.io/

helm install cilium cilium/cilium --version 1.11.6 \
    --namespace kube-system \
    --set kubeProxyReplacement=strict \
    --set k8sServiceHost=REPLACE_WITH_API_SERVER_IP \
    --set k8sServicePort=REPLACE_WITH_API_SERVER_PORT
```

#### Cilium での INBOUND フィルタリング

Cilium では Kubernetes のリソースである NetworkPolicy を使うことでパケットフィルタリングを行うことができる。Listing4.4 の例では myService という label にマッチする pod は 20.1.1.1/32 及び 10.96.0.0/12　を除いた 10.0.0.0/8 からのアクセスは許可するという記述がなされている。

Listing 4.4: networkpolicyinbound

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "cidr-rule"
spec:
  endpointSelector:
    matchLabels:
      app: myService
  ingress:
  - toCIDR:
    - 20.1.1.1/32
  - toCIDRSet:
    - cidr: 10.0.0.0/8
      except:
      - 10.96.0.0/12
```

### Cilium での OUTBOUND フィルタリング

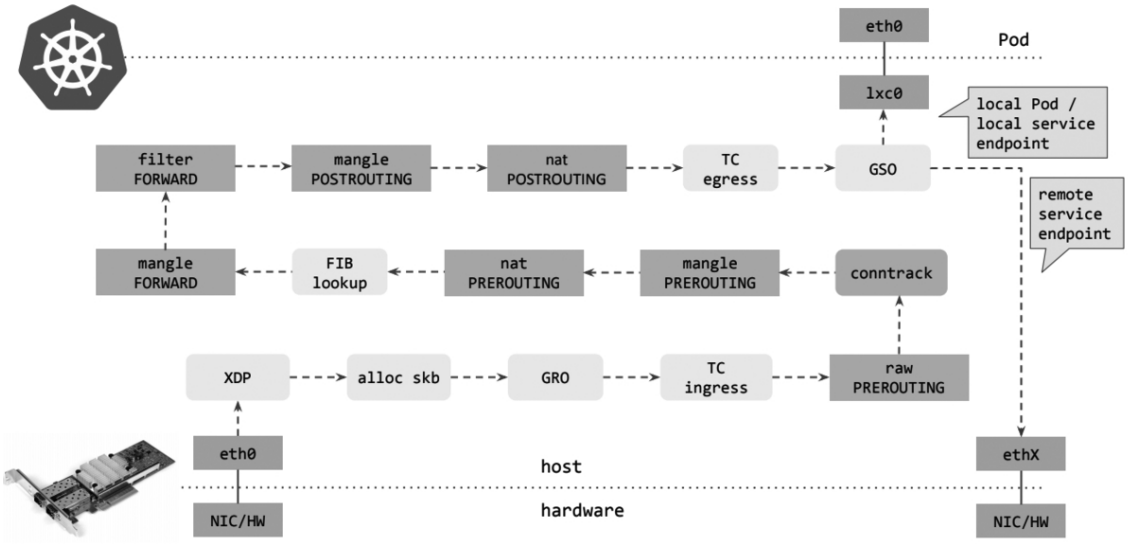
OUTBOUND においても NetworkPolicy でほぼ INBOUND と同様に記述することができる。Listing4.5 の例では myService という label にマッチする pod は 20.1.1.1/32 及び 10.96.0.0/12　を除いた 10.0.0.0/8 へのアクセスは許可するという記述がなされている。

Listing 4.5: networkpolicyoutbound

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "cidr-rule"
spec:
  endpointSelector:
    matchLabels:
      app: myService
  egress:
  - toCIDR:
    - 20.1.1.1/32
  - toCIDRSet:
    - cidr: 10.0.0.0/8
      except:
      - 10.96.0.0/12
```

### kube-proxy との相違点

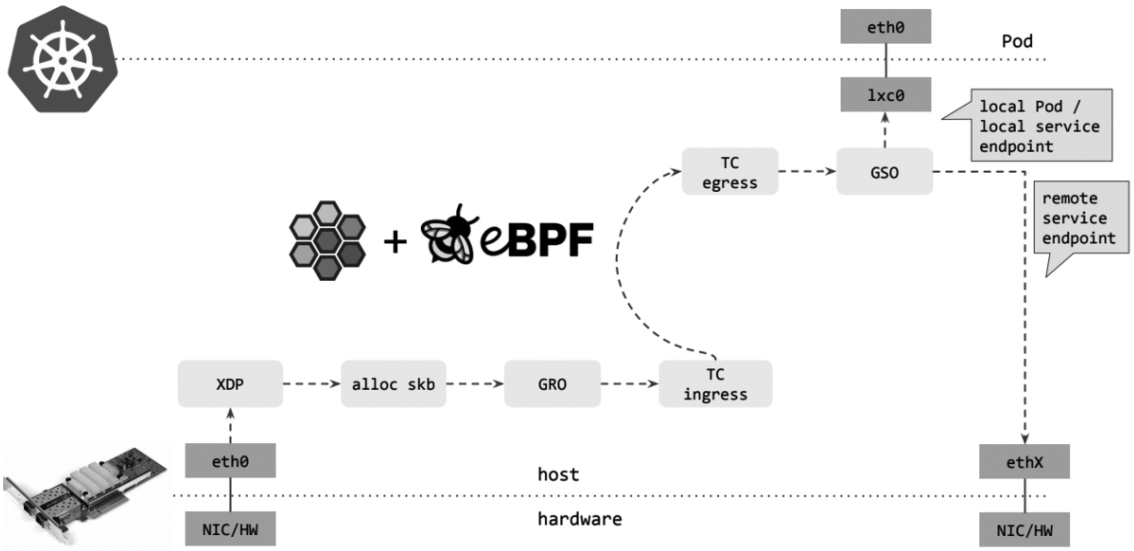
Kubernetes ではデフォルトで kube-proxy を用いているが、kube-proxy での pod 間通信は図 4.2 のように iptables における Tables を複数経由して pod との通信が可能となっている。




Source: KubeCon + CloudNativeCon Europe 2020: eBPF and Kubernetes[40]

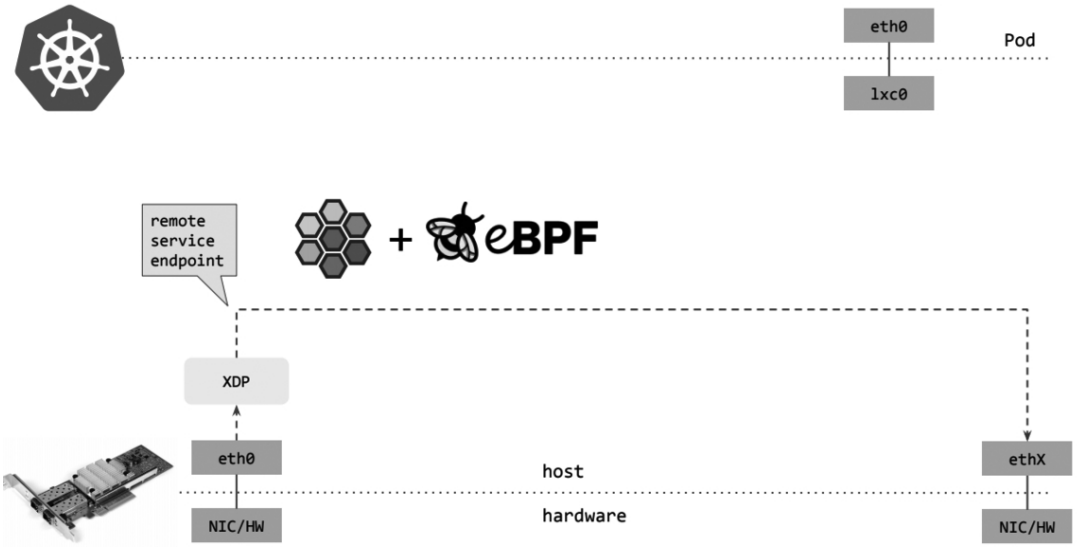
図 4.2: kube-proxy を利用した場合の経路

一方で BPF を使用した場合は Service から同一 Node 上の pod へ転送する場合の図 4.3 及び Service から別 Node の pod へ転送する場合の図 4.4 のように kube-proxy の転送経路と比べて大幅に経路が短縮されていることがわかる。




Source: KubeCon + CloudNativeCon Europe 2020: eBPF and Kubernetes[40]

図 4.3: BPF を用いた場合の自身の Serivce への経路




Source: KubeCon + CloudNativeCon Europe 2020: eBPF and Kubernetes[40]

図 4.4: BPF を用いた場合の他の Node への経路

## 4.2 性能比較

### 4.2.1 性能比較実験

Kubernetes クラスタの構築を行い、iptables と BPF での pod のローリングアップデート適用時間の差及びレスポンスタイムの差を測定した。

実験に伴い pod はリソースの都合上、固定文字列を返却する web サーバーを 200 pod、CNI は iptable と BPF の両方に対応をしている Calico を採用した。実験機は図 4.1 の環境を 3 台用意し、kubeadm[41] にてクラスタ作成を行った。また、フィルタリング設定は Listing4.6 のように設定を行い、iptables を使用している状態では ipset に 200 pod の ip が羅列されている状態を確認した上で実験を行った。

表 4.1: 実験環境

| | |
|---|---|
| OS | Ubuntu 20.04.3 LTS |
| CPU | Intel(R) Xeon® Silver 4114 *2 |
| RAM | 128GB (32GB * 4) |
| DISK | 120GB SSD (BOOT) + 1TB 7.2K RPM HDD |
| CNI | Calico v3.20.2 |

Listing 4.6: networkpolicycalico

```
1  apiVersion: projectcalico.org/v3
2  kind: NetworkPolicy
3  metadata:
4    name: deny-all-allow-default
5    namespace: default
6  spec:
7    types:
8    - Ingress
9    ingress:
10   - action: Allow
11     protocol: TCP
12     source:
13       namespaceSelector: test-label == 'default'
```

### ローリングアップデート適用時間

ローリングアップデート適用時間に関しては 図 4.2 のような結果になっており、3 Node 200pod という環境下においてはほぼ差がでない結果となった。

表 4.2: ローリングアップデート適用時間

| 環境 | 適用時間 |
|---|---|
| iptables | 3min 45sec |
| BPF | 3min 50sec |

### レスポンスタイム

レスポンスタイムの測定は 10,000 リクエストを各環境の NodePort に送ることで測定を行い、図 4.5 のような結果になった。

iptables では大半のレスポンスタイムが 0.005sec であったのに対して、BPF 環境下では 0.002sec でのレスポンスタイムが大半を占めていることがわかる。これは iptables によるシーケンシャルなフィルタリング処理が BPF によって削減された影響と予想される。

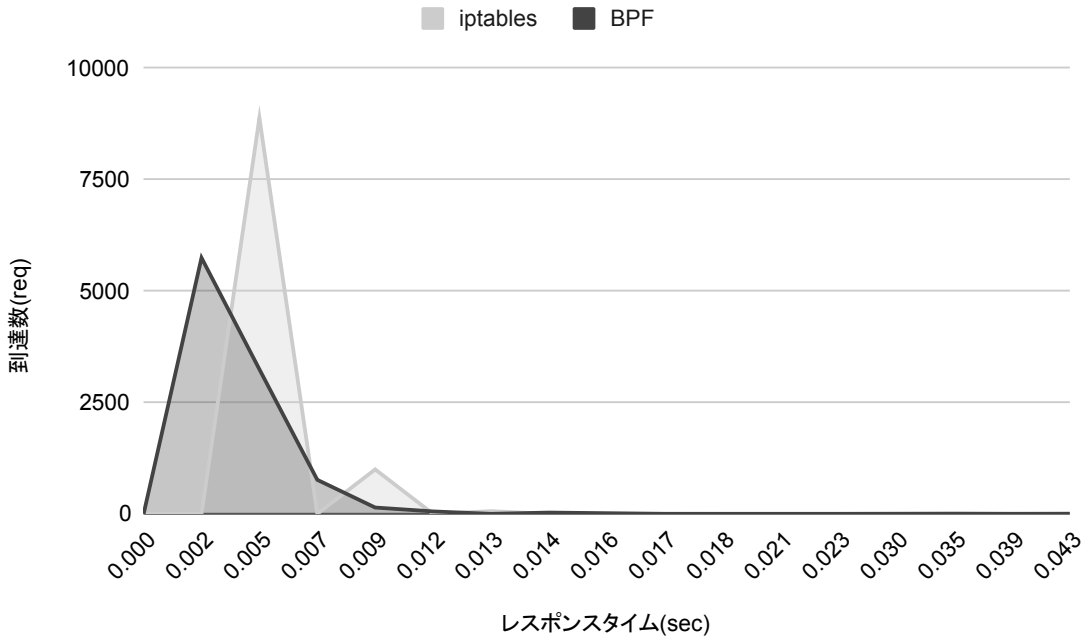
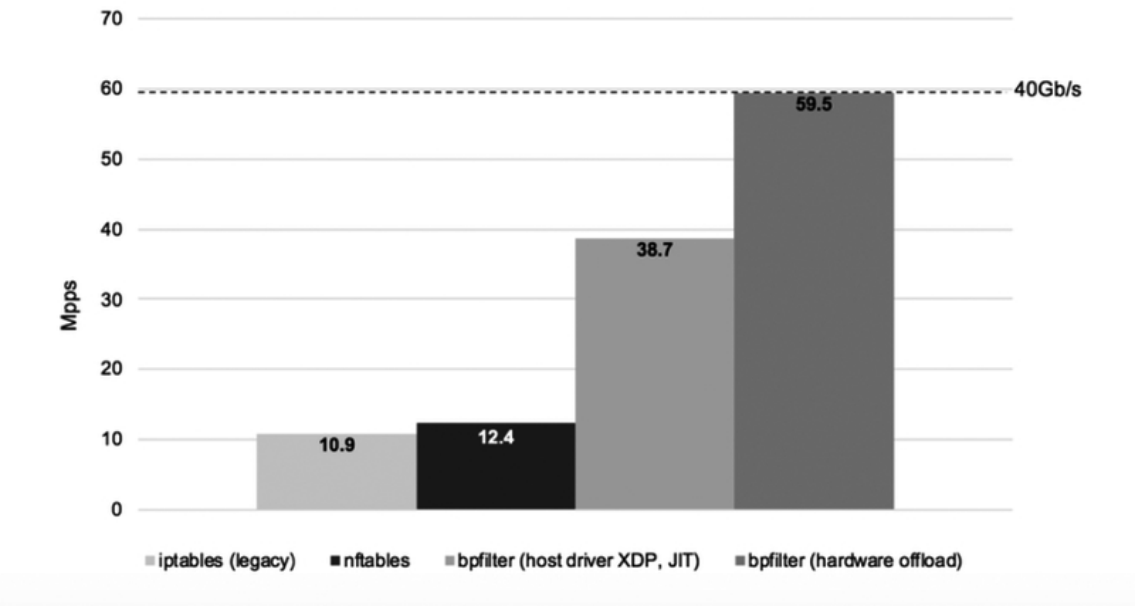



図 4.5: BPF と iptables のレスポンスタイム比較

### 4.2.2 性能比較調査

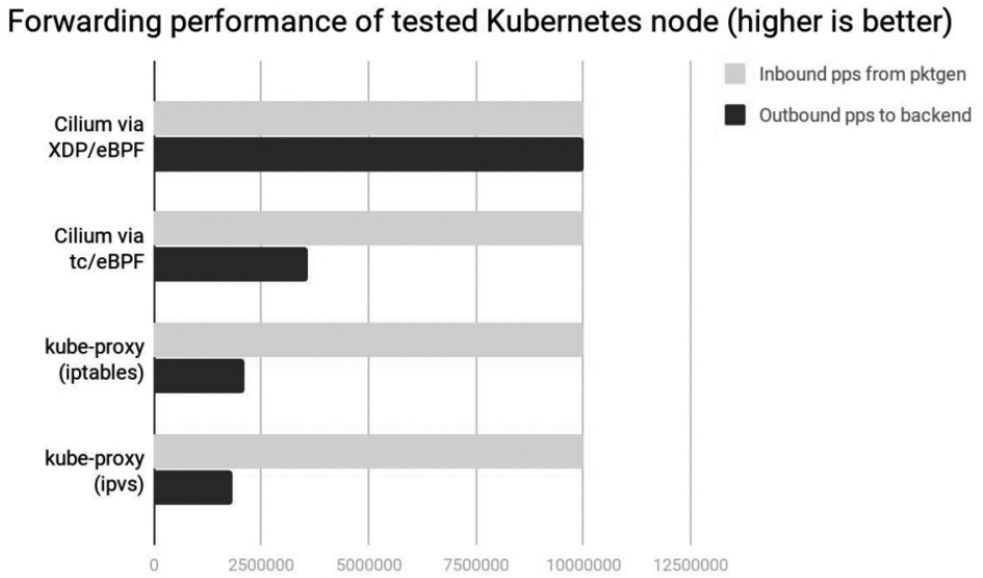
Quentin Monnet は FRnOG 30 にて iptables と BPF の packets per second(pps) 比較を発表している。 [42] この比較では単純なパケット破棄にて比較を行っており、BPF を使った場合 (bpfilter) は iptables 及び nftables を使った場合に比べて 3 倍以上多くパケットドロップを処理できたことを示している。




Source: FRnOG 30[42]

図 4.6: BPF と iptables の pps 比較

また KubeCon + CloudNativeCon Europe 2020 では 図 4.7 のように kube-proxy と Cilium にて BPF を使った場合の性能比較が発表された。 [40] この図では 1 秒間に 10M パケットを受信した場合に Cilium と XDP を使った場合は受信したすべてのパケットがバックエンドサービスに届けられたのに対して kube-proxy を使った場合は 2.1M ほどのパケットしかバックエンドサービスに到達せず、残りのパケットは破棄されたことを示している。




Source: KubeCon + CloudNativeCon Europe 2020: eBPF and Kubernetes[40]

図 4.7: kube-proxy と Cilium の性能比較

上記のような性能面のメリットから、Meta では実際に DDoS 対策と L4LB に BPF を使用しており [44]、他にも Netflix[45] や CloudFlare[46] といった企業が BPF の活用を発表している。

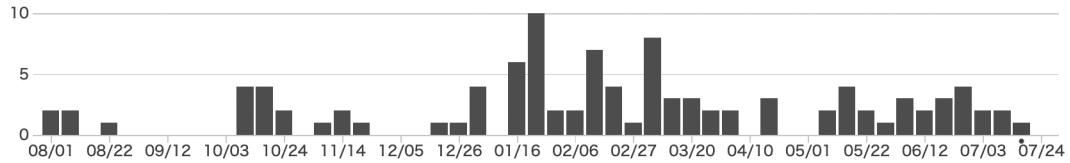
## 4.3 まとめ

iptables と BPF と用いたパケットフィルタリングでは、性能面・維持面・保守性において様々な差異が存在し、移行妥当性を結論づける上で大きな要因になるであろうと考えられる。

**性能面** BPF はパケットフィルタリングにおいては XDP 及び tc に BPF プログラムを Hook させることで iptables を遥かに凌駕する性能を発揮することが示されており、秒間で何万というパケットを処理する必要のあるシステムにおいては iptables の代替ツールとして十分に妥当であると考えられる。一方で小規模なクラスタでは性能面のメリットはそこまで大きくなく、何万ものフィルタリングルールが保持するクラスタか pps が非常に大きいサービスでもない限り、早急に移行する必要はないであろうと考えられる。

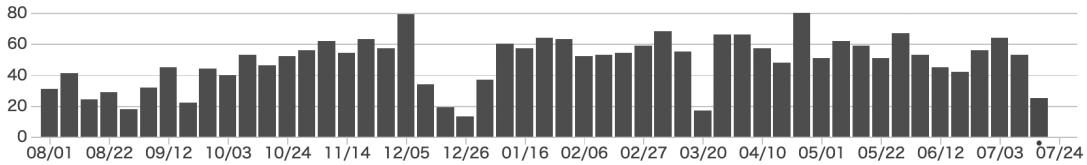
**維持面** XDP にて最大限に性能を発揮させるためには NIC もしくはドライバの制限があるなど、iptables にはないハードウェア制約が存在する点は XDP のデメリットと言える。（ただ、Generic XDP を用いても iptables より遥かに高速にパケットの処理は可能である）また BPF Verifier を意識したコード記述が必要である点、NIC にアタッチするコードの記述、BPF Maps の前提知識がないと必要な情報が得られない点など、iptables と比べて敷居が高いと感じる。Cilium を用いて Kubernetes のノード間通信に BPF を活用する場合は、NetworkPolicy にて Yaml 形式でルールを記述するため前述のような敷居の高さは感じないが、BPF の最大の利点である、ユーザー空間からカーネル空間で動くコードを記述出来るというメリットはなくなり、BPF の動作を変更したい場合は Cilium に変更リクエストを投げる必要がある。

**保守性** iptables が今後新たな開発が活発に行われる可能性が低いのに対して、BPF はその汎用性故にパケットフィルタリングのみならず様々な用途での活用が期待されており、今後も BPF を用いた新しいツールの開発が進むであろうと考えられる。実際に Cilium のアプローチは巨大なアプリケーションが Kubernetes を用いた場合に iptables がパフォーマンスのボトルネックとなる問題を解決可能であり、BPF 非対応の flannel のコミットが図 4.8 のように、ほぼ動きがないのに対して図 4.9 のように、BPF に対応した Cilium のコミットは現在も活発である。




Source: GitHub - flannel-io/flannel[32]

図 4.8: flannel commit activity




Source: GitHub - cilium/cilium[43]

図 4.9: cilium commit activity

BPF を用いたパケットフィルタリングは iptables と比較し、非常に高速にフィルタリングを行うことが出来るが、BPF コードを記述する必要があることや、ハードウェアを意識しなければならないことなど、単純にパケットフィルタリングとして活用するには多少ハードルが高く感じる。しかし Kubernetes 環境下などの iptables がパフォーマンス上問題となりやすい環境では Cilium を代表とする CNI が BPF に対応しており、フィルタリングの記述などは NetworkPolicy を通して記述することができるため、非常にシンプルにフィルタリングを記述することが出来る。

フィルタリングリストが何万となるような大規模なクラスタを運用しているサービスはパケットフィルタリング技術として十分に BPF への移行は妥当であると結論づけると共に、パケットフィルタリングにおいて BPF へ移行するメリットが薄いような小規模なクラスタにおいても、セキュリティ向上やオブザーバビリティを向上したいという要望がある場合においては BPF は非常に有益であり、BPF 対応の CNI に移行することは多角的な面では妥当であると考える。

# 第5章　おわりに

## 5.1　本研究の成果

本研究では現状の iptables の課題を提示するとともに、BPF をパケットフィルタリングに用いた場合にどのようなアプローチを取る必要があるのかを示した上で、メリット・デメリットを提示した。また、Cilium や Calico のように Kubernetes のノード間通信に BPF を活用している場合においては容易に BPF の恩恵を授かることができるが、本研究で示したように内部では XDP や tc に BPF を Hook させており、パケットをトレースする場合や Cilium にコントリビュートする場合において本研究の知識が役立つであろうと考える。

また、BPF の利点はパケットフィルタリングのみならず、ネットワーキング、オブザーバビリティ、セキュリティといった多岐に渡って活躍が期待できる柔軟性にあると考えており、実際に当初は大きなシェアを保持していた flannel は 2022 年の６月２３日から７月２３日の間に 11 Pull Requests しか Merge されていないのに対して、Calico が 84 Pull Requests、Cilium が 248 Pull Requests と BPF をサポートしている CNI としていない CNI の間で大きな差が生まれている。Cilium は eBPF & Cilium Community[47] を運営するなど、BPF の普及にも非常に積極的である。実験にもあったようにパケットフィルタリングのみを考えるのであれば、小規模なクラスタでは BPF へ移行するメリットは薄いが、現在の CNI の開発事情から鑑みると BPF を活用して出来ることは今後も拡大していく一方であり、小規模だがよりセキュアにクラスタを観測を行いたいであったり、CIlium を用いてサービスメッシュを構築したいといったニーズを保持しているエンジニアに対しては本研究は CNI や BPF を理解する手助けになりうるのではないかと考えている。

# 参考文献


[1] docker.com https://www.docker.com/

[2] kubernetes.io https://kubernetes.io/ja/

[3] cgroups(7) — Linux manual page https://man7.org/linux/man-pages/man7/cgroups.7.html

[4] GitHub - containernetworking/cni: Container Network Interface - networking for Linux containers https://github.com/containernetworking/cni

[5] iptables(8) - Linux man page https://linux.die.net/man/8/iptables

[6] iptables の仕組みを図解 — Carpe Diem https://christina04.hatenablog.com/entry/iptables-outline

[7] bpf(2) — Linux manual page https://man7.org/linux/man-pages/man2/bpf.2.html

[8] Kubernetes とは何か？ — Kubernetes https://kubernetes.io/ja/docs/concepts/overview/what-is-kubernetes/

[9] cilium.io https://cilium.io/

[10] Kubernetes のプロキシー https://kubernetes.io/ja/docs/concepts/cluster-administration/proxies/

[11] Ipset - ArchWik https://wiki.archlinux.jp/index.php/Ipset

[12] Scale Kubernetes to Support 50,000 Services [I] - Haibin Xie & Quinton Hoole https://www.youtube.com/watch?v=4-pawkiazEg

[13] netfilter.org https://www.netfilter.org/

[14] firewalld - Wikipedia https://ja.wikipedia.org/wiki/Firewalld

[15] IPCHAINS(8) http://www.skrenta.com/rt/man/ipchains.8.html

[16] The BSD Packet Filter:A New Architecture for User-level Packet Capture https://www.tcpdump.org/papers/bpf-usenix93.pdf

[17] extended BPF - Alexei Starovoitov https://lkml.org/lkml/2013/9/30/627

[18] BPF and XDP Reference Guide https://docs.cilium.io/en/stable/bpf/

[19] Express Data Path - Wikipedia https://en.wikipedia.org/wiki/Express_Data_Path

[20] GitHub - iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more https://github.com/iovisor/bcc

[21] Man page of EXECVE https://linuxjm.osdn.jp/html/LDP_man-pages/man2/execve.2.html

[22] Man page of NAMESPACES https://linuxjm.osdn.jp/html/LDP_man-pages/man7/namespaces.7.html

[23] VPSはどんな技術で動くの?ハイパーバイザー型とコンテナ型の違いを解説 — GMOクラウドアカデミー https://academy.gmocloud.com/vps/20211228/13893

[24] GitHub - lxc/lxc: LXC - Linux Containers https://github.com/lxc/lxc

[25] GitHub - opencontainers/runc: CLI tool for spawning and running containers according to the OCI specification https://github.com/opencontainers/runc

[26] GitHub - containerd/containerd: An open and reliable container runtime https://github.com/containerd/containerd

[27] Open Container Initiative - Open Container Initiative https://opencontainers.org/

[28] rktとは https://www.redhat.com/ja/topics/containers/what-is-rkt

[29] Node.jsとは https://nodejs.org/ja/about/

[30] Cloud Native Computing Foundation https://www.cncf.io/

[31] kubelet—Kubernetes https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[32] GitHub - flannel-io/flannel: flannel is a network fabric for containers, designed for Kubernetes https://github.com/flannel-io/flannel

[33] GitHub - projectcalico/calico: Cloud native networking and network security https://github.com/projectcalico/calico

[34] Container Registry — Google Cloud https://cloud.google.com/container-registry?hl=ja

[35] NGINX(エンジンエックス)｜日本公式サイト https://www.nginx.co.jp/

[36] tc(8) — Linux manual page https://man7.org/linux/man-pages/man8/tc.8.html

[37] bpfilter — LWN.net headlines https://lwn.net/Articles/867803/

[38] BPF, eBPF, XDP and Bpfilter… What are These Things and What do They Mean for the Enterprise? — Netronome https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterpri

[39] Helm https://helm.sh/ja/

[40] KubeCon + CloudNativeCon Europe 2020: eBPF and Kubernetes: Little Helper Minions for Scaling Microservices - Daniel Borkmann https://static.sched.com/hosted_files/kccnceu20/8f/Aug19_eBPF_and_Kubernetes_Little_Helper_Minions_for_Scaling_Microservices_Daniel_Borkmann.pdf

[41] kubeadm のインストール https://kubernetes.io/ja/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

[42] FRnOG 30: Faster Networking à la française - Netronome https://www.netronome.com/blog/frnog-30-faster-networking-la-francaise/

[43] GitHub - cilium/cilium: eBPF-based Networking, Security, and Observability https://github.com/cilium/cilium

[44] GitHub - facebookincubator/katran: A high performance layer 4 load balancer https://github.com/facebookincubator/katran

[45] AWS re:Invent 2019: [REPEAT 1] BPF performance analysis at Netflix (OPN303-R1) https://www.youtube.com/watch?v=16slh29iN1g

[46] Cloudflare が Magic Firewall で、プログラム可能なパケットフィルターを eBPF を使ってどのように構築しているか— The Cloudflare Blog https://blog.cloudflare.com/ja-jp/programmable-packet-filtering-with-magic-firewall-ja-jp/

[47] eBPF & Cilium Community https://www.youtube.com/c/eBPFCiliumCommunity/featured