University of Amsterdam
Faculty of Physics, Mathematics and Informatics.
MSc System and Network Engineering.
Research project 2:

eBPF Based Networking

July 16, 2017

Author:
N. de Bruijn

Supervisors:
Dr. Grosso
Ł. Makowski MSc.

`nick.debruijn@os3.nl`

Abstract

In this research, we make a performance comparison between the standard way of container networking using Linux bridges and iptables and a new innovative that uses eBPF programs as traffic filter. We are mainly interested in the network performance of eBPF container networks based on TCP throughput and latency. To benchmark eBPF's performance, we performed different measurements including, TCP throughput, latency, and the impact of the number of traffic policies applied to containers. Our results show that eBPF was not affected by the number traffic policies and reaches better overall performance compared to a container network using iptables for traffic policy enforcements.

Contents

CHAPTER 1

Introduction

Recently the use of containers increased in popularity, especially in the era of continuous deployment and development where containers are being used to create infrastructures based on microservices. Microservices are applications composed of many smaller loosely coupled services [1]. By separating large services into smaller services it gets easier to develop, maintain and deploy new services.

To support such infrastructure we need a lightweight approach, which will allow such services to run in a separated manner. Therefore, we see that microservices are usually being placed in so called containers. Containers can be seen as a lightweight equivalent of a virtual machine that only virtualises the processes of a physical machine and shares the underlaying OS [2]. To provide process isolation, each container runs in its own namespace [3] and uses Cgroups [4] for resource allocation. To provide a simple and automated way of deploying containers, several management tools have been developed to accomplish this. Examples of such management tools include LinuX Container (LXC) [5] and Docker [2].

For the interconnection of containers we need a network just like any other system would need. In order to do so different approaches can be taken. However, the traditional way is to connect the virtual interface of the container to a Linux bridge, and the Linux bridge to a physical interface. It is then possible to enforce traffic policies using iptables [6, 7]. Unfortunately, the main concern of this approach is that it relies on iptables which tend to have scalability issues in microservice architectures.

In a continuous deployment environment it is not unlikely that multiple microservices and therefore containers are being deployed or removed multiple times a day. Because containers use IP addresses as filter criteria there is a need to update the traffic policies each time a change happens. This means that in the case of iptables new entries have to be made and old entries should be removed. This could lead to possible scalability issues whenever there are thousands of containers running.

A project called Cilium aims to overcome the scalability problem that arises with iptables by rethinking the way of container networking and traffic policy enforcements. Instead of using IP addresses to enforce security policies, labels are used [8]. By enforcing traffic policies based on labels it is possible to eliminate the need of iptables. The mechanism used for this is known as extended Berkeley Packet Filter (eBPF) [9]. eBPF programs run in the Linux kernel and have their own instruction set which allows extending kernel functionality.

Currently, Cilium uses eBPF programs to define a new way of processing packets and enforcing security policies at the traffic control (TC) layer [10]. This means that forwarding decisions can be made even before the traffic reaches the Linux network stack. It is expected that this early processing of packets and elimination of iptables increases the overall network performance of the container network.

1.1 Research question

The contribution of this research is to **evaluate the usability of Cilium as a packet filtering system in a container infrastructure**. To help answering this research question we focus on the performance aspect:

- What throughput and latency we acquire in the case of using Cilium's eBPF program and Linux's iptables as packet filter?
- What effect does the number of iptables rule entries and Cilium policy entries have on the throughput and latency in both cases?
- Is there a turning point in performance when the number of traffic policies defined by Cilium policies or iptables rules increases?
- Does the complexity of Cilium's traffic policy influence the TCP throughput?

1.2 Report outline

In Chapter 2 we present the most relevant related work for this research project. Chapter 3 explains background information on container networking. We will introduce eBPF in Chapter 4 and show how eBPF can be used in container networking. In chapter 5 we will discuss traffic policies and show how this can be applied in container networks using iptables and Cilium labels. Our approach will be presented in Chapter 6 and contains information on the conducted experiments. The results of the experiments will be presented in Chapter 7. In chapter 8 we will discuss the results and make assumptions based on the results. Our conclusion is presented in Chapter 9. In Chapter 10 we discuss the open issues and suggest future work.

CHAPTER 2

Related work

There has been a lot of research into evaluating the interest and the usability of container architectures over hypervisor based virtualization. In [11] R. Morabito, J. Kjällman, and M. Komu make a comparison between two hypervisor based virtualization techniques and two container based virtualization techniques. Another comparison was made in [12] by W. Felter, A. Ferreira, R. Rajamony, and J. Rubio making a performance comparison between native, Docker, and KVM. Both researchers conclude that containers reach better performance compared to other virtualization techniques.

There has also been done a large number of research evaluating the interest and the usability of containers for different use-cases. For example, in [13] C. Ruiz et al. evaluate the usability of containers in high performance computing and in [14], M. Amaral et al. evaluate the usability of containers in microservices architectures based on container performance.

As for traffic policy enforcement, in [15] D. Hoffman et al. show the impact of the number iptables rules on the throughput and latency. Furthermore, in [16] D. Hartmeier made a performance comparison of different packet filters indicating their effect on the network performance using different numbers of traffic policies.

Although the previously conducted research shows the performance benefits of containers and the impact of packet filters on network performance, no research exists that makes a clear performance comparison between different approaches of traffic policy enforcement in container based networks. Therefore, our contribution is to make a network performance comparison between a standard way of packet filtering using iptables and a new approach using BPF programs in a container network. The interest of doing so is mainly because research such as [17] and [18] show the performance benefits using BPF programs.

CHAPTER 3

Container networking background

The goal of this chapter is to provide background information on the fundamental layers of container based networking: container network architecture and type of communication used.

3.1 Container networking architectures

At the basis of container networks we find the container network model. Container network models are well-defined plugins or network drivers that manages the network interfaces and is among other things responsible for connecting containers to a network and the assignment of IP addresses. The way containers are being connected to the network is by using a virtual ethernet device (veth) pair. A veth pair can be seen as a virtual wire that connects two virtual interfaces. These plugins or drivers can be seen as an interface or APIs between the container run time and the network plugins [19].

Currently, there are two proposed standards for configuring network interfaces for Linux containers. Firstly, Container Networking Model (CNM) [20]. Secondly, Container Networking Interface (CNI) [21]. Both models serve the same goal but accomplish this by taking a different approach. Both models are introduced in the next section.

3.1.1 Container Networking Model (CNM)

The Container Networking Model is the standard model used by Docker [22]. Their implementation of the CNM is known as Libnetwork [23]. In essence, Libnetwork provides an interface between the Docker daemon and the network drivers.

At the foundation of the CNM we find three components: 1. sandbox; 2. endpoints; 3. the network. Figure 3.1 illustrates this set.

- Sandbox; refers to the Network Namespace that isolates the container network from the host and other containers running on that host. A sandbox may contain multiple endpoints such that it can connect to multiple networks. Network in the context of the CNM is a network implementation such as Linux bridges and Virtual Lans (VLANs) [24].
- Network Namespace includes the network configurations such as the container's interfaces, routing tables, and DNS settings.
- Endpoint; used to connect the sandbox to a network.

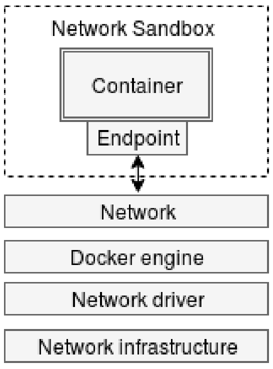


Figure 3.1: CNM model

3.1.2 Container Network Interface (CNI)

The Container Network Interface (CNI) is a container networking specification proposed by CoreOS that became a Cloud Native Computing Foundation project [25]. The CNI was developed as a minimal specification to be an interface between the container runtime and network plugins.

Just like in the case of CNM, it is possible for containers to join multiple networks driven by different plugins. What differentiates CNI from CNM is that CNI describes networks in JSON configuration files that instantiate as new namespaces when CNI plugins are invoked. The container runtime is responsible for the allocation of a network namespace for the container and should assign it a container ID, then it uses CNI to pass along a number of configuration parameters to the network driver (plugin). The plugin is responsible for assigning the container to a network and providing it with an IP address. A report of the assigned IP address is sent back to the container runtime using JSON [26].

At the foundation of CNI we find the following three main components: 1. CNI Specification; 2. Plugins; 2. a Library. Figure 3.2 illustrates these components.

- CNI specification; used as API between the container runtime and the network plugins for container network setup.
- Plugins; used to provide network setups (network drivers).
- Library; used to provide a Go[1] implementation of the CNI specification that runtimes can use to more easily consume CNI.

Libraries are being used to write plugins to configure network interface in Linux containers. Because the interface uses JSON schemes, making it straight-forward to create CNI plugins.

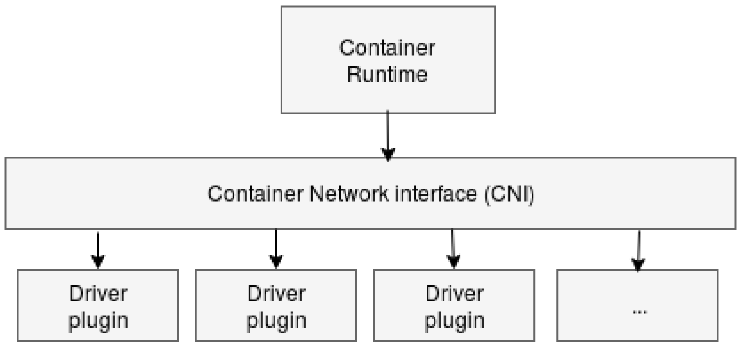


Figure 3.2: CNI model

[1]https://github.com/golang/go

3.2 Types of communication

In the previous section we discussed the two proposed standards for container networking. The current section covers types of container communication we consider important for this research. Regardless from the type of communication either CNI or CNM can be used.

Linux bridge

The most basic and classic approach is to connect the container to a Linux bridge. The Linux bridge is a virtual Layer 2 device that is being emulated by the Linux kernel. The Linux bridge allows multiple veths to be connected simultaneously creating a single broadcast domain among containers. Figure 3.3 illustrates this approach.

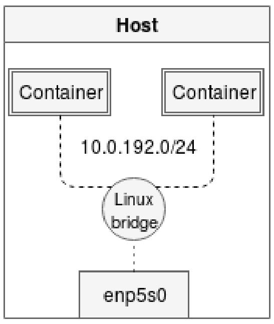


Figure 3.3: Linux bridge architecture

Static routing

The method described above can be extended to allow multi-host communication. This is done by coupling the Linux bridge to a physical interface. If both sides use this setup, a link can be created using a physical connection between the two physical interfaces. Static routes can be applied so that packets originating from containers can be forwarded to the correct interface. In such approach it is recommended to allocate different subnets on both hosts to prevent overlapping IP addresses. Figure: 3.4 illustrates this method.

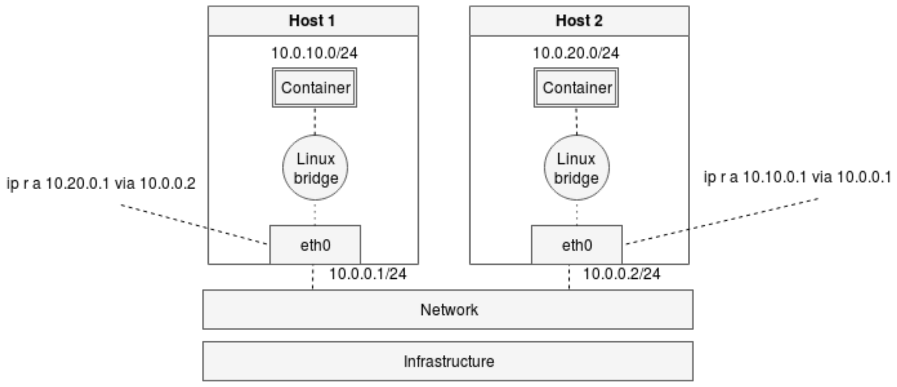


Figure 3.4: Simple Multi-host communication.

Overlay

Another approach is to use an overlay network. An overlay network is a virtual network that creates a virtual topology on top of the physical topology of the underlying network. Hosts are usually connected through virtual interfaces which correspond to paths in the underlying physical network. A standard that allow tunnels to be build in such virtual topology is known as VXLAN [27]. The tunnels that VXLAN creates are identified with a VXLAN network identifier (VNI). VXLAN based overlay network allows forwarding Layer 2 segments over a Layer 3 network by encapsulating packets in a VXLAN header. This allows containers residing on different hosts to communicate with each other as if they are local. Figure: 3.5 illustrates an example of a VXLAN overlay.

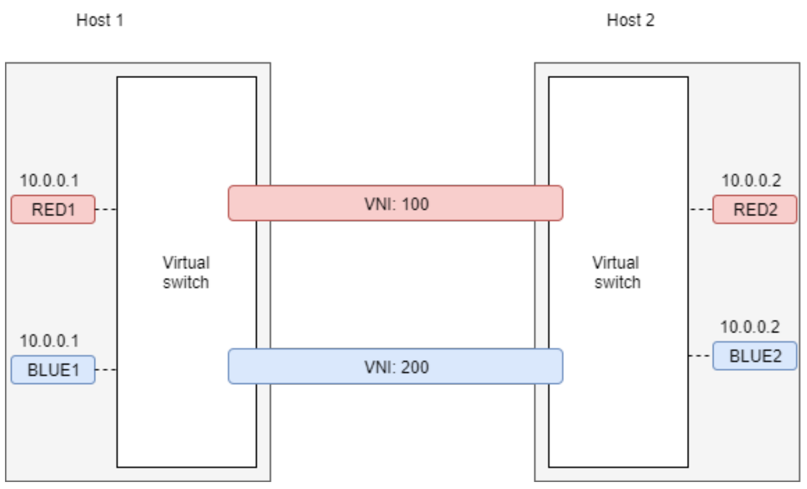


Figure 3.5: VXLAN communication.

CHAPTER 4

extended Berkeley Packet Filter (eBPF)

In the previous chapter we discussed the basics of container networks and different approaches to implement container networks. In the current chapter we will focus on a recent technique known as eBPF.

The Berkeley Packet Filter (BPF) was introduced in 1992 as a kernel based virtual machine-like program provide fast interpretation and execution to filter packets using a simple instruction set [17]. In 1999 BPF has been extended to BPF+ [28]. This extension introduced new features to BPF including a just-in-time (JIT) assembler to increase performance, and a byte code optimizer and verifier. The result is a general packet filter framework that can be used to inject BPF programs in the Linux kernel to extend its functionality during runtime. The current version, known as extended BPF is available since the release of kernel 3.18. Currently (e)BPF is being used in many fields to create efficient applications and services. Examples of such services include Linux analysis tools [29], Network engineering [30] and intrusion detection [31].

4.1 eBPF characteristics

eBPF consists of eleven 64-bit registers, with 32-bit subregisters, a program counter and a 512-byte stack [32]. These registers are named r0 to r10, where r10 is the only read-only register which contains the frame pointer address to access the eBPF stack space.

4.1.1 Instruction set & program limitations

eBFP uses its own instruction set that currently consists of 87 usable instructions. However, this number increases at every kernel release, extending the functionality of eBPF. The current limit of an eBPF program is 4096 instructions for each program. If there is a need for more instruction, for example for more complex programs, then tail calls can be used. Tail calls allow eBPF programs to jump to other eBPF programs. The current number of allowed tail calls is currently limited to a maximum of 32. The maximum values were defined to guarantee kernel stability.

4.1.2 eBPF maps

In order to keep state among multiple BPF program invocations, eBPF uses key/value stores named Maps. Maps reside in kernel space but can also be accessed through file descriptors from user space, maps can even be arbitrarily shared among other eBPF programs or user space applications. The current limit of maps an eBPF program can access directly is 64.

4.2 Creating an eBPF program

Currently there are two ways of creating eBPF programs. The first way is to write the program directly in byte code and the second way is a more common approach, which is to write the program in source code using a higher-level programming language like C, Go, or P4. The advantage of writing your eBPF program in a standard programming language is that there is no need for exact byte code knowledge. There are tools like LLVM[1] that are capable to compile source code into byte code. LLVM compiles the byte code into an ELF[2] file which can be injected into the Linux kernel.

To inject the bytecode into the kernel, the bpf() syscall is being invoked. However, before the program is loaded into the kernel, the program is being checked by an in-kernel verifier. This verifier guarantees that the program is safe to be executed in the kernel such it does not crash whenever the program is injected. This verifier makes it possible to inject eBPF programs safely in the kernel during run time.

Whenever the verifier successfully checked the byte code and does not find any problem that could lead to kernel crashes then the bytecode is being injected at the desired hook point. Now the eBPF program is loaded in the kernel, it can be used to filter packets on the hook point. Figure 4.1 illustrates a graphical representation of the above process. Note that in figure 4.1 the eBPF program is hooked at the traffic control (TC) layer so that filtering can be done even before a packet enters the kernel network stack.

Possible decisions and functionality of eBPF programs hooked to the TC layer include rewriting packet content, extend/trim packet sizes, redirect packets to other netdevices and enforcing traffic policies.

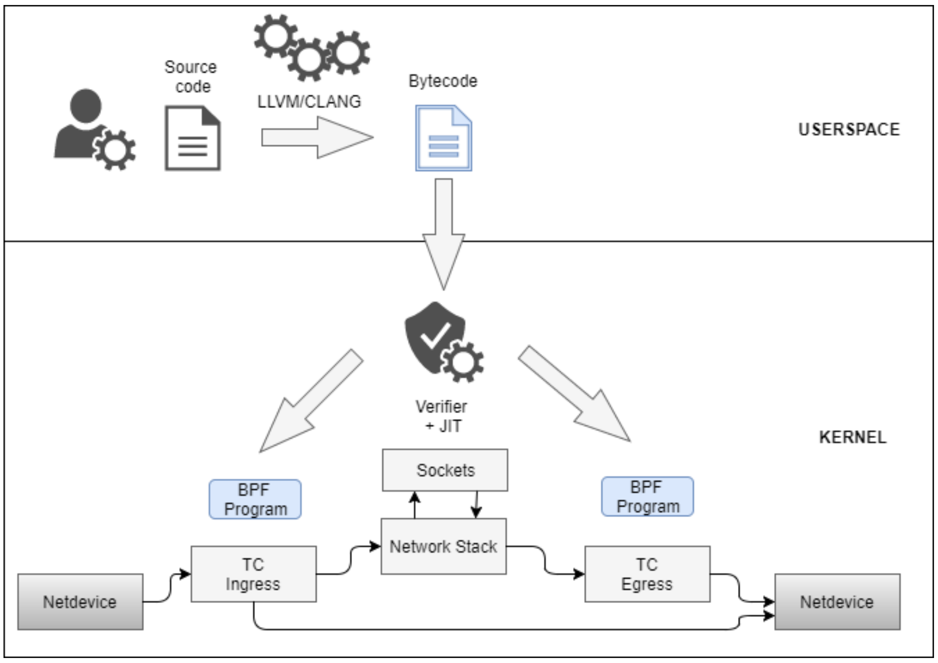


Figure 4.1: Overview of BPF kernel injection[3].

[1]https://llvm.org/
[2]http://wiki.osdev.org/ELF
[3]**https://cilium.readthedocs.io/en/latest/architecture/**

4.3 eBPF in container networks

eBPF programs can also be used in container networks. In these cases, eBPF programs are injected in the kernel to filter and redirect packets. A project that focuses on creating a container network using eBPF programs is Cilium [33]. Cilium provide a layer on top of an existing container management system to provides a daemon and network driver which allow containers to use eBPF programs to forward traffic.

Cilium does so by creating an eBFP program at the start of each container and attaches the eBPF program to container's veth, the host veth, and to the TC layer of the data path. The eBPF program can now be used to e.g. parse code, apply policies, forward packets to other eBPF programs or to the TC layer. Furthermore, the Cilium daemon is responsible for compiling, updating, and injecting the eBPF programs. This can be done on the fly, so that the programs can be updated while the container is running.

Because eBPF programs hook into the kernel, it is possible to filter on system call level which is a much lower level than any standard packet filter. In theory, this should mean that forward decisions can be made at an early state. Figure 4.2 and figure 4.3 illustrates the differences in how filtering is done using eBPF and iptables.

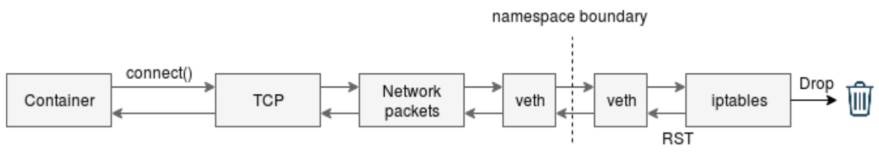


Figure 4.2: Policy filtering using iptables

In the case of iptables, if a container wants to connect to another container it send a connect() system call to the Linux network stack. The Linux network stack processes the system call and assembles a network packet. This network packet gets forwarded to the veth pair. The host veth is connected to a netdevice where the packet gets filtered. Whenever the container is not allowed to establish this connection then the network packet is dropped. To notify the requesting container, a RST message is send back following the same path.

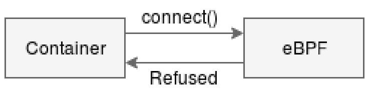


Figure 4.3: Policy filtering using eBPF.

In the case of eBPF, the connect() system call can be filtered by the eBPF program that is hooked in between the container and the Linux network stack. This means that the system call can serve as input for the filter and policies can be enforced at an early state.

eBPF based architecture

For a container network to work with eBPF, not only should containers have eBPF programs, but every netdevice that takes part in the network should have an eBPF program. By gluing all eBPF programs together we create a microservice like infrastructure that is possible to change at any given moment. Figure 4.4 illustrates a simple local eBPF based container network.

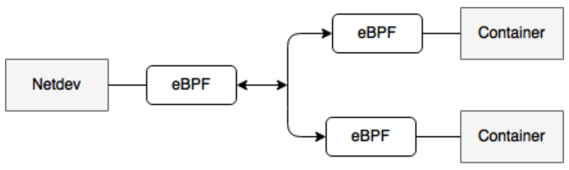


Figure 4.4: Simple eBPF network.

CHAPTER 5

Traffic policies

For security reasons it is recommended to enforce some kind of traffic policies for your container network so that you can control which service's (containers) are allowed to communicate with each other. This is an important consideration as most containers are deployed with no restriction to any open ports. This means that every container in the same network should be able to communicate with each other. One of the problems that arise here is that whenever one of your containers is breached, this container can be used to access other containers in your network. By setting restrictions on the containers using traffic policies you can significantly reduce the risks and impact in such occasions.

This section will be limited to traffic policy enforcement using iptables for the classic way of networking and Cilium's traffic policies for an eBPF container network.

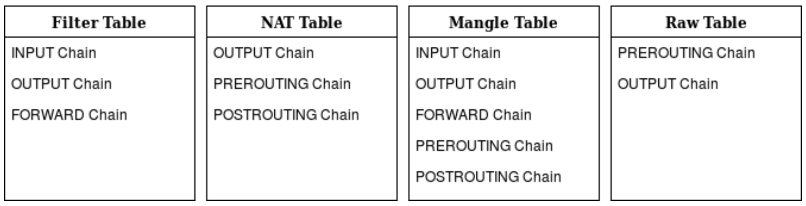
5.1 Policy enforcement using iptables

To enforce traffic policies using iptables we can take two approaches. First, we can define rules directly on a container itself, this means that every container maintains its own list of restrictions. And secondly, place iptables rules on the Linux bridge connecting the containers. This allows having a single place to control and update restrictions. However, the second approach is recommended as the orchestration of the first approach where every single container needs to maintain its own list of iptables rules may lead to scalability issues in large environments.

5.1.1 Iptables characteristics

Iptables uses netfilter as filter mechanism that allows packets to be accepted, dropped, queued, or returned based on its Layer 3 and Layer 4 characteristics. Furthermore, iptables allow you to insert network address translation (NAT) rules allowing containers using private IP addresses to be translated to public routable IP addresses.

The way iptables is organised is by using so called tables where each table has its own chain with rules. By default, iptables come with the following pre-defined tables, the filter, NAT, Mangle, and Raw table, each serving its own purpose. Figure 5.1 illustrates the predefined tables and chains used by iptables. In the case of container networking we mainly focus on two of the tables namely: 1. NAT table; 2. Filter table.



Filter Table	NAT Table	Mangle Table	Raw Table
INPUT Chain	OUTPUT Chain	INPUT Chain	PREROUTING Chain
OUTPUT Chain	PREROUTING Chain	OUTPUT Chain	OUTPUT Chain
FORWARD Chain	POSTROUTING Chain	FORWARD Chain	
		PREROUTING Chain	
		POSTROUTING Chain	

Figure 5.1: Iptables pre-defined tables and chains.

NAT table is being used for network address translation (NAT), just as its name suggests. Whenever desired, this chain can be used to translate containers' private IP addresses to public routable IP addresses to allow containers to communicate with the Internet.

To create container specific filter rules, we mainly focus on Filter table. The Filter table allows rules specifying restrictions for packets based on its Layer 3 and Layer 4 characteristics.

5.1.2 Iptables rules matching

Each packet entering a netdevice can be filtered by netfilter. Whenever a packet is being filtered it matches the content of the packet header to the complete set of rule entries in each chain until it finds a matching rule for that packet. This matching is done using a top down approach to make sure every rule entry is checked. Whenever netfilter reaches the bottom of the chain and does not find any matching rule then it applies the default rule, which is by default an accept rule. Figure 5.2 illustrates this process.

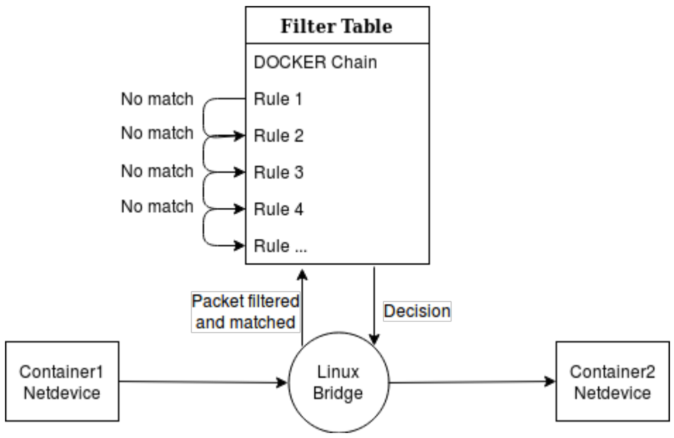


Figure 5.2: iptables matching procedure.

This way of matching seems to be a decent solution whenever we have a couple of restrictions defined. However, in a complex microservices infrastructure running hundreds of containers using this iptables approach can face scalability issues and might become a performance bottleneck in the network. This has to do with the fact that if we want to define traffic policies for all those containers, we have to enter thousands of rules in our iptables filter table to accomplish this. If for each incoming packet this list has to be matched, then the time this process takes might influence the speed at which we can process packets.

Another problem that arises in a microservices infrastructure is that the infrastructure usually changes multiple times a day. For iptables this means that for every change, the chain entries need to be updated which can be hard to orchestrate.

5.2 Traffic policy enforcement using Cilium (eBPF)

In the previous section we gave a quick overview of how iptables can be used to enforce traffic policies and gave an indication why iptables' design might not be suitable for container network hosting microservices. In this section we will take a closer look of how the Cilium project tries to overcome this iptables problem by implementing a different mechanism for policy enforcement that does not relies on iptables or Linux bridges.

5.2.1 Cilium labels

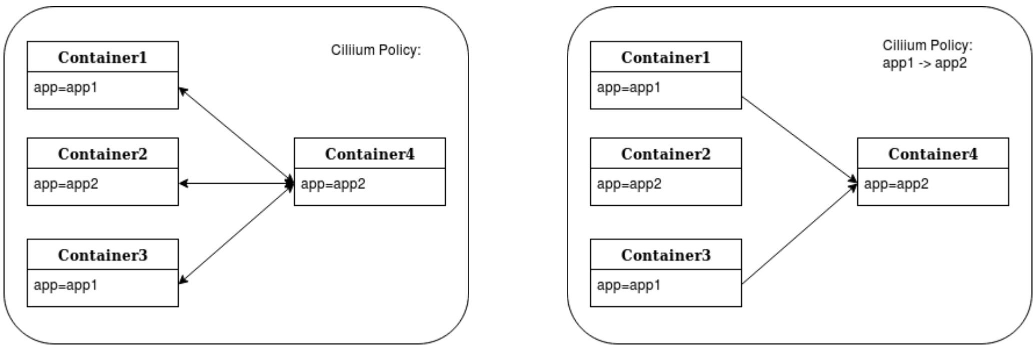
The way Cilium enforces security policies is based on labels [34]. Each container (endpoint in Cilium) is assigned a label that describes the purpose or coherence of a single or group of containers. Whenever a policy needs to be applied to a container, it can use the container's label so that the policy will be enforced to every container containing this label.

Label characteristics

Labels consist of a key and value where the key portion is mandatory and must be unique. The value portion is optional. However, if the value portion is used, then both the key and value should match. Keys and values may be formatted as a single string by specifying key=value.

Using labels

Every container should be labeled at run time, only then traffic policies can by applied for the container's label. Whenever no policies are enforced for a specific label then no restrictions are enforced at all. This means that the container accepts every incoming packet. However, when one traffic policy is enforced saying that container with label app1 is allowed to talk with containers with label app2, then all other containers containing a label other than app1 are blacklisted and that traffic is dropped. Furthermore, the policies are uni-directional and stream based, this means that whenever we have a policy app1 → app2. Then app1 can send traffic to app2 and receive reply messages to this traffic, but app2 is not allowed to talk to app1. To allow this, there should also be a policy specifying app2 → app1. Figure 5.3 shows the difference in non policy enforcement and a single policy enforcement.



(a) Cilium with no policy enforcement. (b) Cilium with policy enforcement.

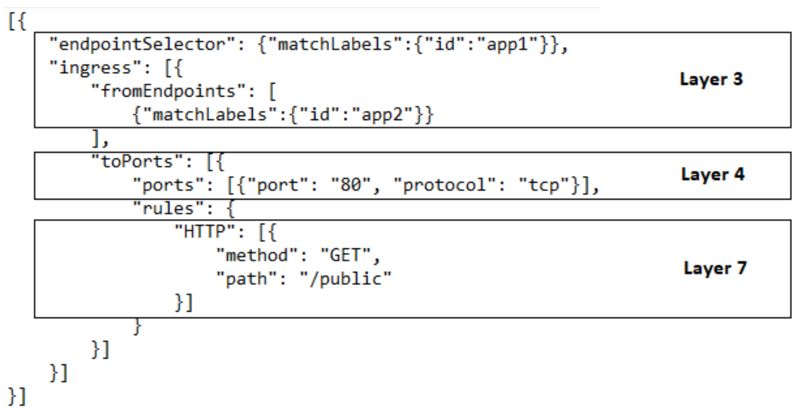
Figure 5.3: Example of Cilium's policy enforcement and effect on communication.

If we would simple translate this approach to iptables rules this would result in thousands of rules to express every single combination possible and we will end up with the same scalability issue discussed in section 5.2.2. Instead of translating this to iptable rules, Cilium associated a cluster wide unique ID for every unique label combination. This requires a key-value store that will capture all label combinations that occur to associate every combination an unique label ID. This label ID is then integrated into the packet header when a container sends out a packet. At the receiving side you can use this label ID to do a single hash table look-up to figure out if this label ID (label combination) is allowed to talk with this container.

The result of this is, that regardless the amount of labels or the complexity of the policy it should always result in a single hash table look-up.

5.2.2 Defining policies

In addition to iptables's Layer 3 and Layer 4 policies, Cilium also provides capabilities to enforce policies on the application level. This allows engineers to define more fine grained restrictions to containers. The policies are defined as JSON objects and can be imported to Cilium using the command line interface (CLI). Figure 7.8 shows an example of a Cilium policy enforcing restrictions on all three layers.



```
[{
    "endpointSelector": {"matchLabels":{"id":"app1"}},
    "ingress": [{
        "fromEndpoints": [
            {"matchLabels":{"id":"app2"}}
        ],
        "toPorts": [{
            "ports": [{"port": "80", "protocol": "tcp"}],
            "rules": {
                "HTTP": [{
                    "method": "GET",
                    "path": "/public"
                }]
            }
        }]
    }]
}]
```

Figure 5.4: Cilium JSON policy.

CHAPTER 6

Approach

In this research project we focus on evaluating the usability of Cilium as a packet filter in a container infrastructure by looking at its network performance. However, before we can make a clear statement of Cilium's performance, we will compare it against the network performance of the approach using a Linux bridge, static routing, and iptables. A container management system that allows us to setup such infrastructure is Docker [2]. The approach we take to measure the performance is a black box approach, this means that no optimization is done to network performance.

6.1 Experimental environment

The experimental environment we setup for our experiments consists of two native servers residing in the NSE OpenLab. Both servers are configured with Ubuntu 17.04 with kernel 4.10 to meet the software requirements of Cilium.

The hardware used are two Supermicro X8DTT-H servers using an Intel(R) Xeon(R) CPU E5620 @2.40GHz with 23G DDR3 @1066 MHz of RAM.

Both servers are configured with the Docker container engine version 17.05.0-ce, build 89658be and with Cilium version 0.9.90, build 08c1e0c4. Both Docker and Cilium run natively on the server. For Cilium this means that it has to be compiled from scratch. In order to switch between the traditional environment using only Docker and the Cilium environment, we can simply turn on/off the Cilium plugin. This makes it unnecessary to setup separated environments for Cilium and Docker.

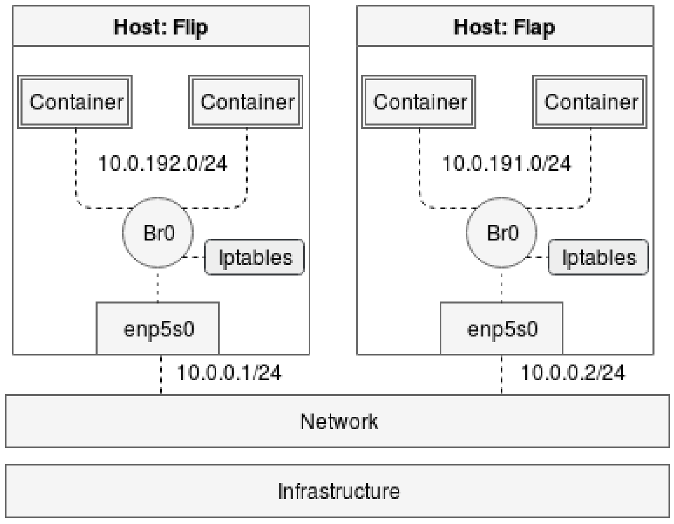


Figure 6.1: Experimental environment using Docker.

In figure 6.1 shows the experimental environment using Docker. In this environment we run two containers per host connected to the Br0 Linux bridge. On the host Flip our containers are assigned with a private IP address in the 10.0.192.0/24 IP range. On the host Flap, the containers are assigned with a private IP address in the range 10.0.191.0/24. To connect the two hosts, we created a directly connected link between the hosts using a 10G interface. In order to make the networks reachable over this link, we configured static routes on both ends. To enforce any traffic policies, we specify iptables rules to filter packets on the Br0 Linux bridge, such that we have a centralized place to orchestrate our traffic policies.

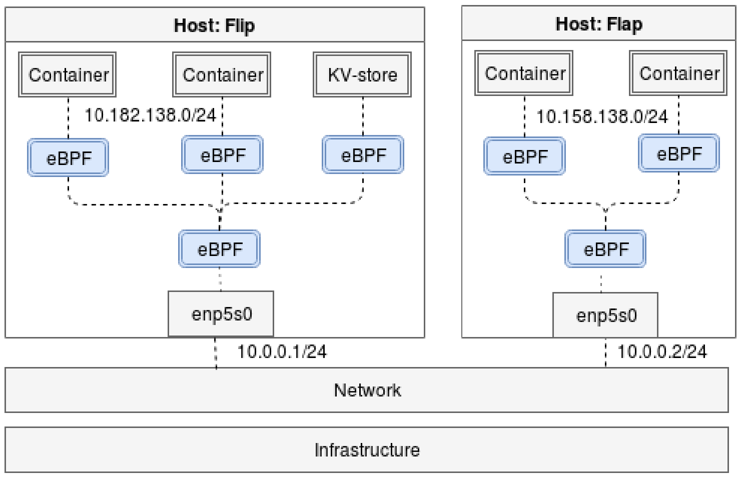


Figure 6.2: Experimental environment using Cilium.

In figure 6.2 illustrates the environment in the case when the Cilium plugin is enabled and containers are created with the Cilium-net network driver. Instead of having Iptables and Linux bridges, we can now see the eBPF programs that take care of the networking and policy enforcements. Also to prevent IP conflicts, we have assigned a different range of IP addresses for containers using the Cilium-net network driver. For the host Flip we assign the IP range 10.182.138.0/24 and for the host Flap we assign the IP range 10.158.138.0/24. Note that the ranges are derived from the public IP addresses assigned to our servers, x.x.x.182 for Flip and x.x.x.158 for Flap. Furthermore, to allow our two hosts to communicate with each other we use the same link created for Docker and also configured static routes for these networks.

In order for Cilium to work with policies in a multi-host environment we need to setup a key value-store. The key value-store used is Consul [35] and is running inside a container using the Cilium network driver. Both hosts can connect to this key value-store using the IP address of Flips enp5s0 interface together with the Layer 4 port number 8500.

6.2 Tooling

To conduct our experiments we will make use of three Linux tools: 1. Iperf3; 2. Netperf; 3. Perf.

6.2.1 Iperf3

Iperf3 is a kernel based network benchmark tool for active measurements of the maximum achievable bandwidth on IP networks. The tool works in a traditional client-server model where the server opens a socket and the client generates packets. We will use this tool create a TCP stream from one container to another to measure the maximum reachable TCP throughput without any network performance tweaking.

6.2.2 Netperf

Iperf3 does not allow us to measure the latency in a efficient way. Therefore, we use Netperf for this purpose. Netperf works in the same manner as Iperf3, in a client-server model but comes with some additional functionality. The additional functionality we are interested in is the TCP_RR

(Request, Response) option which allows us to send TCP requests and receive responses. The sizes of these requests and responses can be set by specifying it using parameters. The output of the program will give us the transfer rate per seconds which we can use to derive the latency from.

6.2.3 Perf

Linux Perf is a performance analyzing tool in Linux that allows us to trace packets through the kernel.

6.3 Experiment Scenario

We are interested in researching the network performance of eBPF and iptables in two settings, single-host and in a multi-host setup. For each scenario we will test the throughput and latency with no additional Cilium policies/iptables rules entries and the influence whenever we start adding Cilium policies/iptables rules. We measure this for both eBPF and iptables where we test this for iptables in two settings. Firstly, with no matching rule so that the filter has to go trough all entries. Secondly, for iptables with a top of chain rule in order to measure the throughput and latency in the most optimal case.

Single-host means that we run two containers on the same host and test the TCP performance from the client container to the server container. Because these containers run local, there is no external networking involved.

Multi-host means that we run one container on every host where host Flap will run the client container and host Flip will run the server container. This multi-host setup requires external networking and therefore uses the 10G interface that has a direct connection to the other host.

Furthermore, we are interested in the impact of the complexity of Cilium policies on the TCP throughput. To test the impact of the complexity of a policy we create a single traffic policy for a container and start to increase its complexity by adding more Layer 4 restrictions to it. We will test this for both single-host and multi-host communication.

Lastly, we will do kernel tracing for the network that is affected by the number of traffic policies to get a better understanding of the factors that affect the performance.

6.3.1 Experiments

To answer our research questions we conduct four experiments. Every experiment will run for 1 minute, this should give the TCP stream enough time to initialize its stream. To measure the variance of the TCP streams we will conduct the experiment 10 consecutive times. In the next sections we provide the experiment specific details.

Experiment - 1 TCP throughput

Our first experiment is to measure the TCP throughput between the localhost containers, and in the multi-host setup. The below listing shows the conditions used for the TCP stream.

- We will use Iperf3 to send an unidirectional bulk data transfer over a single TCP connection with a maximum transmission unit (MTU) of 1500 bytes.
- In order to measure the influence of the traffic policy rules entries, we will conduct our experiment for each of the following entries set: 0, 1, 5, 10, 25, 50, 100, and 200.

Experiment 2 - TCP latency

For the second experiment to measure the latency on localhost and in the multi-host setup, we use similar conditions:

- We will use Netperf to send a Request and Response to the server container using a 100 byte request and a 200 byte response using a single TCP connection. The maximum transmission unit is still 1500 bytes.
- We will use the same set of traffic policy entries to measure the influence of the policies and iptables rules on the TCP latency.

Experiment 3 - Cilium policy complexity

The third experiment will run with the following conditions:

- We will use Ipref3 with the same conditions as in experiment 1.
- In order to measure the influence of the complexity of traffic policies we add layer 4 rules to a single policy.
- We will use the following restriction set: No restrictions, single policy allow communication, additional layer 4 restrictions, 10 additional layer 4 restrictions, 50 additional layer 4 restrictions, 100 additional layer 4 restrictions, and 200 additional layer 4 restrictions.

Experiment 4 - Kernel tracing

Our last experiment is to use perf to trace packets through the Linux kernel with as goal to gather more information about the measured performance:

- We will run this experiment for the approach that shows us a performance decrease whenever the number of traffic policies increases.
- We will use the following traffic policy set: 0, 100, 200.

CHAPTER 7

Results

In this chapter we present the results of our four experiments. In all four cases we present first the results obtained on the single-host scenario, followed by the results for the multi-host environment.

7.1 Experiment 1 - TCP Throughput

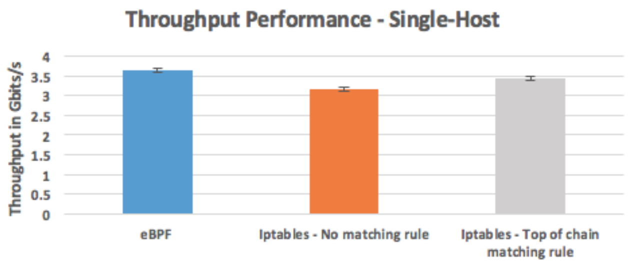


Figure 7.1: Baseline TCP throughput on single-host for eBPF (left), iptables with no matching rule (center) and iptables with a top of chain matching rule (right).

Figure 7.1 clearly shows that the eBPF network performs better compared to iptables. Where eBPF reaches a throughput of 3.64 Gb/s, iptables with no matching rule did not reach higher than 3.17 Gb/s. The throughput of iptables with a top of chain matching rule performs with 3.42 Gb/s slightly better than with no matching rule. The reason Docker's Iptable top of chain matching rule reaches higher throughput has to do with the fact that no matching rule means that there are no additional iptables rules except for the 12 default iptables rules Docker uses to allow container communication.

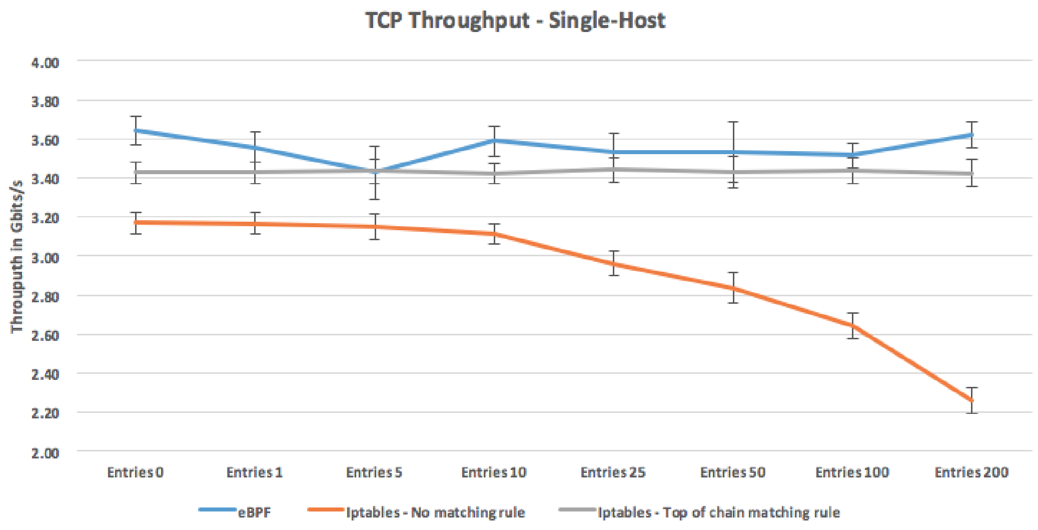


Figure 7.2: TCP throughput on single-host as function of increasing number of policies for eBPF (blue), iptables with no matching rule (orange) and iptables with a top of chain matching rule (gray).

Figure 7.2 shows that the number of policies seems not to affect the performance of the eBPF network in any way. However, we do see the throughput decreasing whenever we have one and five eBPF policy entries stored in the key value-store. But as the throughput starts to increase again at 10 entries and even gets higher at 200 entries, we believe that this performance decrease at 5 entries has to do with the variation of the send rate of the TCP stream and not with the number of eBP policy entries. For iptables shows a complete different trend. For iptables with no matching rule the numbers of iptables entries seems to greatly affect the throughput. Where it starts with 3.17 Gb/s it keeps decreasing all the way to 2.26 Gb/s for 200 iptables rules entries. This is a performance loss of 29%. The iptables with a top of chain matching rule shows a very steady trend where the throughput stays around 3.42 Gb/s. Taking the two iptables results, we can state that that the average performance of iptables in a single-host situation is somewhere in between the two values for a given number of entries.

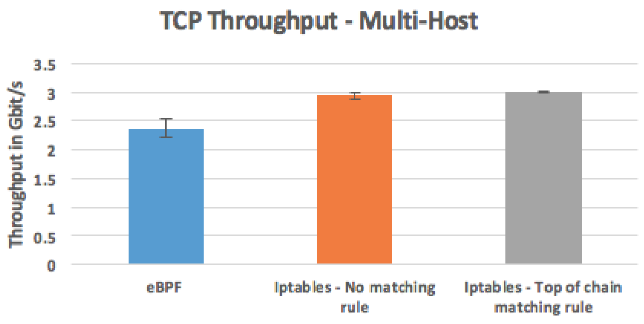


Figure 7.3: Baseline TCP throughput on multi-host for eBPF (left), iptables with no matching rule (center) and iptables with a top of chain matching rule (right).

Figure 7.3 shows that iptables reaches a much higher throughput compared to the eBPF network. Where eBPF reaches only 2.37 Gb/s, iptables reaches 2.93 Gb/s with no matching rule and even 2.99 Gb/s with a top of chain matching rule.

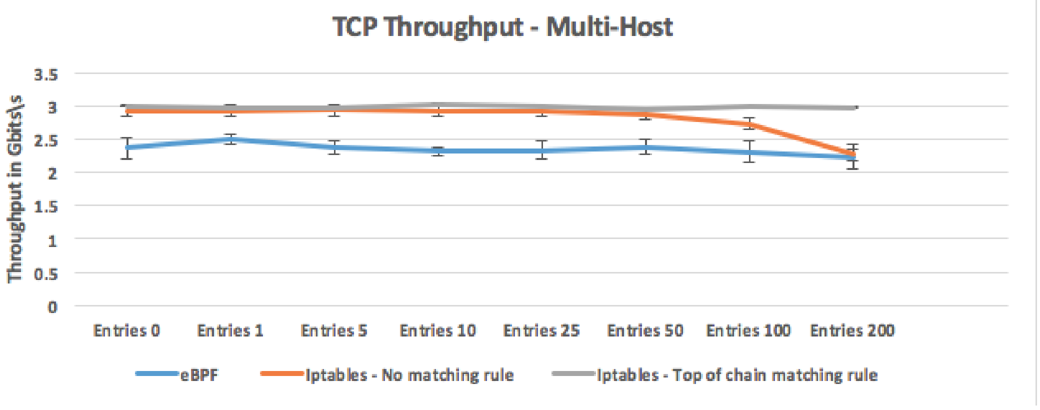


Figure 7.4: TCP throughput on multi-host as function of increasing number of policies for eBPF (blue), iptables with no matching rule (orange) and iptables with a top of chain matching rule (gray).

From figure 7.4 we can observe that eBPF performs worse than iptables in a multi-host environment. Where eBPF reaches a maximum throughput of 2.51 Gb/s, reaches iptables 2,94 Gb/s in the case of no matching rule and 2,99 GB/s with a top of chain matching rule. However, for iptables with no matching rule, we can observe the influence of iptables rules entries starting at 50 entries. At 200 entries we can even observe that iptables with no matching rule is reaching the performance of eBPF. After this point the eBPF network will start to show better performance.

7.2 Experiment 2 - TCP Latency

For this experiment we used a TCP request/response stream to measure the latency of eBPF and iptables. For the detailed description of the experiment we refer the reader to the descriptions given in Chapter 6.

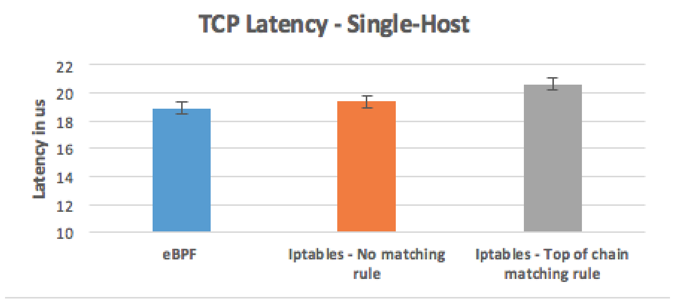


Figure 7.5: Baseline TCP latency on single-host for eBPF (left), iptables with no matching rule (center) and iptables with a top of chain matching rule (right).

Figure 7.5 shows that ePFB has the lowest latency. This was expected as eBPF also showed a higher throughput in experiment 1. eBPF has a latency of 18.91 μs compared to iptables with no matching rule 19.40 μs and iptables with a top of chan matching rule 20.63 μs.

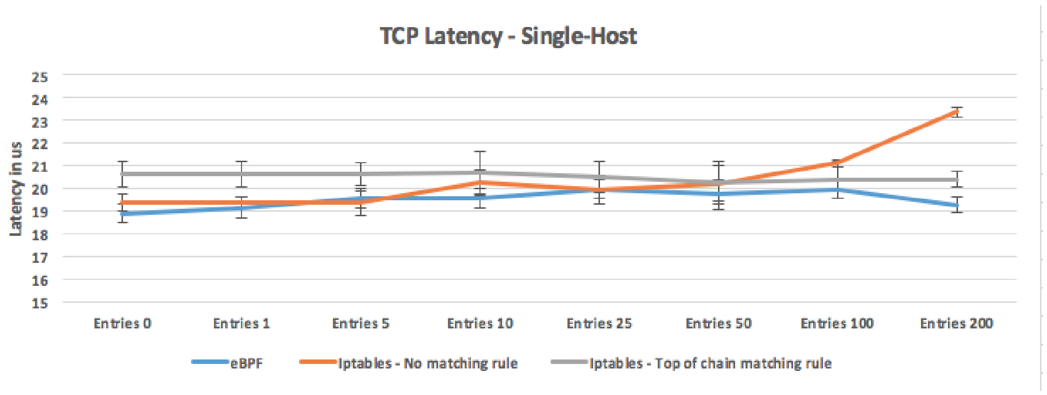


Figure 7.6: TCP latency on single-host as function of increasing number of policies for eBPF (blue), iptables with no matching rule (orange) and iptables with a top of chain matching rule (gray).

In figure 7.6 we can observe the same situation regarding the performance as we seen the TCP throughput. The eBPF network and the classic container network with a top of chain rule show a pretty stable trend. However, for no iptables matching rule, we see an increase for the latency after 25 traffic policies entries.

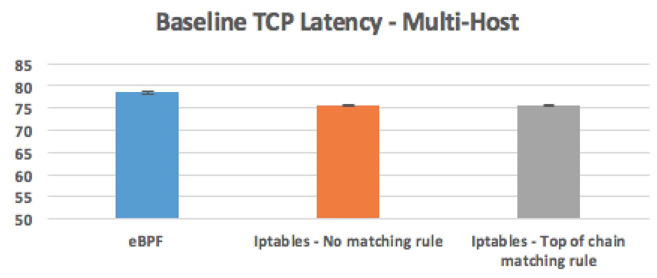


Figure 7.7: Baseline TCP latency on multi-host for eBPF (left), iptables with no matching rule (center) and iptables with a top of chain matching rule (right).

Based on the throughput measurements, we expect that the eBPF network should reach a higher latency than in both iptables cases. If we look at the figure then this is exactly what can be observed. The latency in the eBPF network reaches 78 μs compared to iptables 75 μs for both approaches.

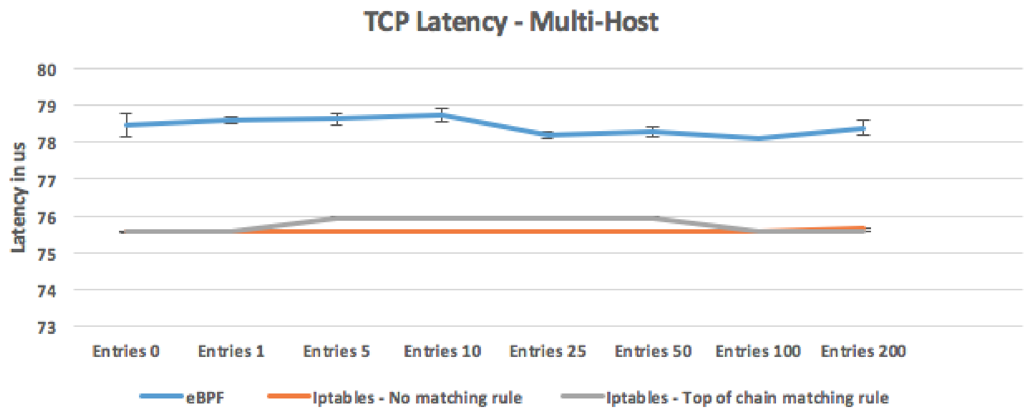


Figure 7.8: TCP latency on multi-host as function of increasing number of policies for eBPF (blue), iptables with no matching rule (orange) and iptables with a top of chain matching rule (gray).

We can see the same trend as which we seen in section 5.2.2 where both the Cilium policies and iptables rules seems not to affect the TCP throughput. This can also be observed for the TCP latency. Cilium seems to perform worse than iptables even when 200 policy/iptables rules entries are in place. The only difference in latency, until 200 policy/iptables rules entries seems to be caused by the variation of the TCP stream.

7.3 Experiment 3 - Cilium policy complexity

We tested the complexity of a Cilium policy by creating a single Cilium policy for a container and increases its complexity by adding Layer 4 restrictions using a predefined set. For the detailed description of the experiment we refer the reader to the descriptions given in Chapter 6.

While performing this execution we were limited by the number of Layer 4 restriction we could add. Currently, Cilium's policies do not allow you add more than 40 restrictions in a single Cilium policy. The set was reduced from 200 to 40. The new set used is 0, 10, 20, 40

7.3.1 Policy complexity results

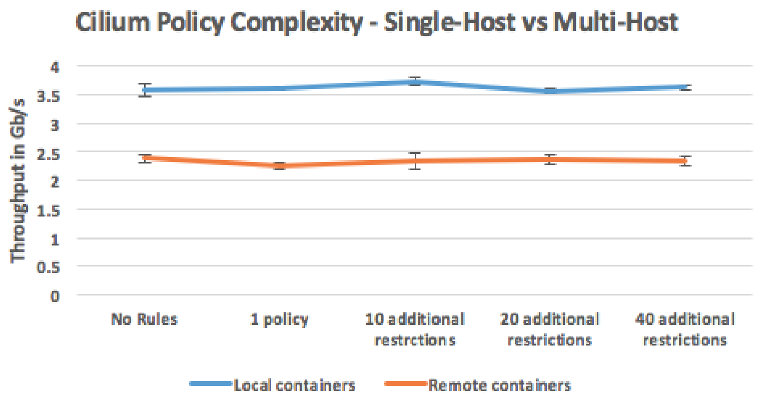


Figure 7.9: Effect of Cilium policy complexity on the TCP throughput as function of increasing number of policies for single-host (blue) and multi-host (orange).

Figure 7.9 shows the results of the impact of Cilium's policy complexity on the TCP throughput. Both the local and remote situation show a similar trend, and in both situations there seems to be no influence on the TCP throughput caused by the complexity of the policy.

7.4 Experiment 4 - Kernel tracing

We used kernel tracing to verify if iptables was responsible for the performance loss using iptables as traffic policy mechanism. For the detailed description of the experiment we refer the reader to the descriptions given in Chapter 6.

7.4.1 Local vs Remote container communication

The outcome from the perf report verifies our expectations that iptables is the main factor for the performance loss when the number of iptables rules entries increases.

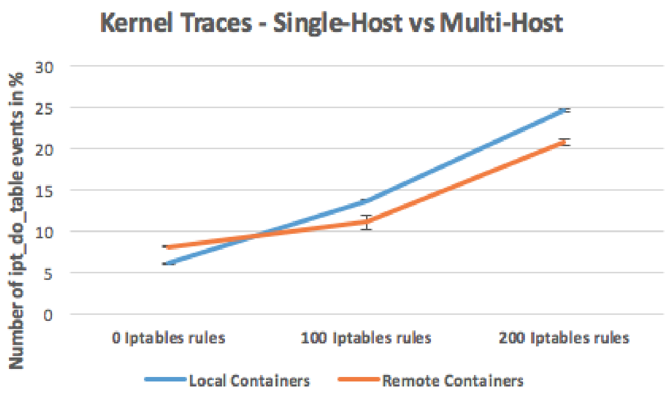


Figure 7.10: Number of ipt_do_table events as function of increasing number of policies for single-host (blue) and multi-host (orange).

Figure 7.10 shows that as the number of iptables rules starts to increase the number of ipt_do_table also increases. As there were no other remarkable changes observed in the Perf report, this is a clear indication that the only cause of performance loss is due the number of iptables rules. The numbers of both cases are close related. However, in a remote situation there seems to be less performance loss.

CHAPTER 8

Discussion

In this project we compared the TCP throughput and latency of a eBPF container network against a classic container network based on Linux bridges and iptables to determine the usability of Cilium as a packet filter in a container network.

The benefits of this elimination of iptables can been observed in the result section of the conducted experiments. In the results of the TCP throughput experiments for local container communication we can clearly observe that the eBPF network outperforms the classic container networks. We can also observe that the number of traffic policies does not affect the throughput in the eBPF network. In comparison to the classic container network there is an improvement as here we can observe a big performance loss based on the number of traffic policies enabled. The results of the kernel traces indicate that the only performance loss is due to iptables. At this point it is still unclear why eBPF reaches better performance than the the classic container network with a top of chain iptables rule. However, we believe that this has to do with paths packets take in the kernel, which is for eBPF better optimized or shorter. Future research could be performed to determine the complete kernel paths taken by both approaches.

Observing the remote communication results, we notice that eBPF actually reaches a lower throughput than the classic container network. We assume that this is caused by extra overhead generated by eBPF when it has to send data over a physical link. However, this is just an assumption and should be looked into in future work. In the situation of the classic container network we can again observe a performance loss due to iptables. This performance loss is less noticeable up to 50 traffic policy entries than in local container communication. However, after 50 traffic policies entries, we start to observe a big performance loss. The kernel traces again indicate that this is only because of iptables entries. As eBPF is not effected by the number of traffic policies and the classic container network does, we believe that in a large and complex real-life setup we could still reach better performance using eBPF.

The biggest downsides we faced using eBPF as container network, is that Cilium doesn't allow us to define more than 40 rules in a single policy. Even if a second policy is created with one or more extra layer 4 policies for the same container, Cilium output an error and doesn't allow it. However, according to T. Graf, lead developer of Cilium, this is an open issue that they try to solve in the upcoming release. He points out that this is not due the limitations of eBPF or eBPF maps.

8.1 Performance tuning

Even tough we wanted to test eBPF and iptables as a black-box approach where we test both approaches with no additional network optimization, we still wanted to see if both approach could be optimized as the result of the tests showed rather poor overall performance. When traffic doesn't have to leave the host, we expect our containers to send 64k packets. With 64k packets we should be able to reach much higher speeds than the maximum of 3.64 Gb/s. Furthermore, the performance over the 10G interface, where we reached a maximum of 2.99 Gb/s seemed rather questionable as well. Therefore, we decided to look into this to see what is the cause of this low performance.

8.1.1 Determine the baseline

We started with creating a baseline by performing a Iperf TCP benchmark between our physical 10G interfaces. Here we managed to reach a throughput of 9.80 Gb/s. This indicates that communication between two hosts can definitely be sent at a higher rate than the 2.99 Gb/s we managed to reach. To figure out if the problem had to do with Docker or Cilium, we created a netnamespace with interfaces on both servers that communicate with each other using the 10G interface. The scripts[1] that we used for that can be found in appendix A. We used Iperf to send a TCP stream from the veth of one netnamespace the the veth of the second netnamespace, for this test we reached a throughput of 6.6 Gb/s. This is another indication that it should be possible achieve higher throughput rates than we reached, even when using namespaces and veth pairs. With this information our baseline is set to 6.6 Gb/s.

8.1.2 Kernel tracing

Kernel tracing was used to trace the packet sizes through the kernel to figure out if the kernel did something to our packets while using Docker or Cilium. The maximum size observed for both local and remote communication was 1464. This is a clear indication that our packets are being segmented at some point in the kernel. However, as we did not find any packet larger than 1446 in the kernel trace, it was hard to tell where the packet got segmented. However, we did find a **skb_segment** kernel symbol which is a function for packet segmentation on the given skb[2] (socket buffer). This is a clear indication that the kernel is doing packet segmentation making sure the maximum packet size is 1464.

8.1.3 TCP Segmentation offloading

This made us look into the offload functions for our (virtual) interfaces. Here we found that every virtual interface has **tx-tcp-segmentation** turned off by default. This means that the segmentation is done by the kernel using the CPU. By enabling this offload function, the TCP segmentation is offloaded to the NIC allowing you use larger window sizes and ignore the MTU set to the interface. This feature should be turned on by default, it remains a question to us why this feature was turned off for all interfaces. As for the results of our performance measurements we are now able to reach 20.80 Gb/s on the eBPF network and 20.10 Gb/s using iptables for local container communication. For the remote communication we were able to reach about 1.5 GB/s more for both networks. We believe that it is still possible to optimize the performance over the 10G interface as we still did not reach our baseline of 6.6 Gb/s, this is can be looked into during future work.

To see if this influences our earlier results, we performed a small test using the setup from experiment 1 with 1 iptables rule and another test with 200 iptables entries. For the test with 1 iptables rule entry we achieved 19.97 Gb/s, for the test with 200 iptables rules we achieved a troughput of 17.10 Gb/s. This indicates that the higher achieved throughput does not influence the results of our experiments.

[1]Special thanks to Daniel Borkmann for providing these scripts

[2]http://vger.kernel.org/ davem/skb.html

CHAPTER 9

Conclusion & Future work

During our literature study in chapter 2 till chapter 4, we show that there are different approaches and container network designs to choose from. Which approach to take is dependent on the needs and complexity of the infrastructure. Furthermore, we give an introduction to eBPF based container networking as of how it is currently being used in Cilium. We end the background by showing the approach of traffic policy enforcement for a classic container network and for the eBPF network. We have seen that Cilium takes traffic policy enforcements one step further by introducing layer 7 policies which allow you to create more fine-grained traffic policies. This means that Cilium completely eliminates the need of iptables rules to enforce restrictions between containers.

eBPF seems to be a promising addition to container networking. In this research we show that Cilium's implementation of an eBPF container network can overcome the scalability concerns that arise with using iptables as traffic policy mechanism in container networks. Furthermore, we give a clear indication of the performance and traffic policies impact on the throughput of the eBPF network. The results of the experiments show us that Cilium's eBPF network is not affected by the number or complexity of traffic policies.

In the case of local container communication we reach better performance in every situation with the eBPF network compared to the classic container network. In the case of remote container communication, the eBPF network reaches a lower throughput than classic container network. However, in this situation there is a turning point at 200 traffic policy entries. After this point the eBPF network starts to show better performance. Therefore, we believe that eBPF networks scales better in microservices architectures than the classic container network as the number of traffic policies does not affect the eBPF network performance.

9.1 Future work

Through out the project we have performed a performance comparison between Cilium's eBPF network and Docker using the most classic form of container networking using Linux bridges and iptables. The results clearly indicates the differences between both networks. However, it is not always completely clear what causes this differences in throughput and latency. Therefore, it would be interesting to research such open issues to get a better understanding of what influence the performance and how to increase the performance, possible with the aid of kernel tracing.

Another interesting topic could be to extend this research by performing network performance tests using different container network methods like Macvlan or other packet filters.

CHAPTER 10

Acknowledgements

I would like to thank the supervisors P. Grosso and Ł. Makowski for their constant feedback, guidance and input throughout the research project. In addition, I would like to thank T. Graf for the introduction to Cilium networking and lastly, D. Borkmann and A. Martins for the guidance with debugging Cilium.

Bibliography

[1] J. Stubbs, W. Moreira, and R. Dooley. Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways*, pages 34–39, June 2015.

[2] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[3] Eric W. Biederman. Multiple instances of the global linux namespaces. 2006.

[4] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):39, 2015.

[5] LXC authors. LinuX Containers. `https://linuxcontainers.org/`.

[6] Mark Church. Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks. `https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks`, 2017. Visited on: 06-06-2017.

[7] Docker. Understand container communication. `https://docs.docker.com/engine/userguide/networking/default_network/container-communication/#container-communication-between-hosts`.

[8] Cilium Authors. Architecture Guide Policy enforcement. `https://cilium.readthedocs.io/en/latest/architecture`. Visited on: 06-06-2017.

[9] The berkeley packet filter. `https://www.kernel.org/doc/Documentation/networking/filter.txt`.

[10] Cilium: Networking and security for containers with bpf and xdp. `https://opensource.googleblog.com/2016/11/cilium-networking-and-security.html`, 2016.

[11] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015.

[12] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[13] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. *Performance Evaluation of Containers for HPC*, pages 813–824. Springer International Publishing, Cham, 2015.

[14] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, Sept 2015.

[15] Daniel Hoffman, Durga Prabhakar, and Paul Strooper. Testing iptables. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pages 80–91. IBM Press, 2003.

[16] Daniel Hartmeier. Design and performance of the openbsd stateful packet filter (pf). In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 171–180, Berkeley, CA, USA, 2002. USENIX Association.

[17] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. `http://www.tcpdump.org/papers/bpf-usenix93.pdf`, December 1992. Visited on: 07-06-2017.

[18] S. Jouet, R. Cziva, and D. P. Pezaros. Arbitrary packet matching in openflow. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6, July 2015.

[19] Harmeet Sahni. The Tale of Two Container Networking Standards: CNM v. CNI. `http://www.nuagenetworks.net/blog/container-networking-standards/`.

[20] Docker inc. Container Network Model. `https://github.com/docker/libnetwork/blob/master/docs/design.md`.

[21] Cloud Native Computing Foundation. the Container Network Interface. `https://github.com/containernetworking/cni`.

[22] Mark Church. Docker Container networking Model. `https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks#The_Container_Networking_Model`.

[23] Docker inc. Libnetwork. `https://github.com/docker/libnetwork`.

[24] B. Dykes D. McPherson. VLAN Aggregation for Efficient IP Address Allocation. `http://www.faqs.org/rfcs/rfc3069.html`.

[25] The Linux Foundation. CNCF Hosts Container Networking Interface (CNI). `https://www.cncf.io/blog/2017/05/23/cncf-hosts-container-networking-interface-cni/`.

[26] Lee Calcote. The Container Networking Landscape: CNI from CoreOS and CNM from Docker. `https://thenewstack.io/container-networking-landscape-cni-coreos-cnm-docker/`.

[27] T. Sridhar M. Bursell L. Kreeger P. Agarwal K. Duda D. Dutt M. Mahalingam C. Wright, M. Bursel. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. `https://tools.ietf.org/html/rfc7348`.

[28] Andrew Begel, Steven McCanne, and Susan L Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 123–134. ACM, 1999.

[29] IOVisor community. BPF-based Linux IO analysis, networking, monitoring. `https://github.com/iovisor/bcc`.

[30] C. Partridge, A. C. Snoeren, W. T. Strayer, B. Schwartz, M. Condell, and I. Castineyra. Fire: flexible intra-as routing environment. *IEEE Journal on Selected Areas in Communications*, 19(3):410–425, Mar 2001.

[31] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[32] Cilium Authors. Architecture Guide. `https://cilium.readthedocs.io/en/latest/architecture/`.

[33] Cilium. Linux Native, HTTP Aware Networking and Security for Containers. `https://github.com/cilium/cilium`.

[34] Cilium Authors. Architecture guide - Labels. `http://docs.cilium.io/en/latest/architecture/#labels`.

[35] HashiCorp. Consul. `https://www.consul.io/`.

APPENDIX A

Netnamespace scripts

```
#!/bin/sh
set -x
LOCAL_IP=f00e::1
REMOTE_IP=f00e::2
IFACE=enp5s0
ROUTE_HO=f00e::/64
CONT_VETH=f00d::3/64
REM_VETH=f00d::1
HOST_VETH=f00d::4/64
MAC_REMOTE=00:60:dd:47:a1:32
killall irqbalance
killall irqbalance
killall NetworkManager
killall NetworkManager
sysctl net.ipv4.ip_forward=1
sysctl net.ipv6.conf.default.forwarding=1
sysctl net.ipv6.conf.all.forwarding=1
ethtool -K $IFACE gso on gro on tso off
ip -6 a add $LOCAL_IP/96  dev $IFACE 2> /dev/null
ip -6 r add $REMOTE_IP/128 dev $IFACE 2> /dev/null
ip netns del blue 2> /dev/null
ip netns add blue
ip link del dev veth0 2> /dev/null
ip link add veth0 type veth peer name veth1
ip link set veth1 netns blue
ip link set mtu 1500 dev veth0
ip netns exec blue ip link set mtu 1500 dev veth1
ip netns exec blue ip link set up dev lo
ip netns exec blue ip link set up dev veth1
ip netns exec blue ip -6 a add $CONT_VETH dev veth1
ip netns exec blue ip -6 r add $ROUTE_HO dev veth1
ip link set up dev veth0
ip -6 a add $HOST_VETH dev veth0
ip -6 r add $REM_VETH/128 dev $IFACE
ip -6 n replace $REMOTE_IP dev $IFACE lladdr $MAC_REMOTE nud permanent
ip -6 n replace $REM_VETH  dev $IFACE lladdr $MAC_REMOTE nud permanent
VMAC=`cat /sys/class/net/veth0/address`
ip netns exec blue ip -6 n replace $REMOTE_IP dev veth1 lladdr $VMAC nud permanent
ip netns exec blue ip -6 n replace $REM_VETH  dev veth1 lladdr $VMAC nud permanent
```

```
#!/bin/sh
set -x

LOCAL_IP=f00e::2
REMOTE_IP=f00e::1
IFACE=enp5s0
ROUTE_HO=f00e::/64
CONT_VETH=f00d::1/64
REM_VETH=f00d::3
HOST_VETH=f00d::2/64
MAC_REMOTE=00:60:dd:47:a1:2e
 killall irqbalance
 killall irqbalance
 killall NetworkManager
 killall NetworkManager
 sysctl net.ipv4.ip_forward=1
 sysctl net.ipv6.conf.default.forwarding=1
 sysctl net.ipv6.conf.all.forwarding=1
 ethtool -K $IFACE gso on gro on tso off
 ip -6 a add $LOCAL_IP/96   dev $IFACE 2> /dev/null
 ip -6 r add $REMOTE_IP/128 dev $IFACE 2> /dev/null
 ip netns del blue 2> /dev/null
 ip netns add blue
 ip link del dev veth0 2> /dev/null
 ip link add veth0 type veth peer name veth1
 ip link set veth1 netns blue
 ip link set mtu 1500 dev veth0
 ip netns exec blue ip link set mtu 1500 dev veth1
 ip netns exec blue ip link set up dev lo
 ip netns exec blue ip link set up dev veth1
 ip netns exec blue ip -6 a add $CONT_VETH dev veth1
 ip netns exec blue ip -6 r add $ROUTE_HO dev veth1
 ip link set up dev veth0
 ip -6 a add $HOST_VETH dev veth0
 ip -6 r add $REM_VETH/128 dev $IFACE
 ip -6 n replace $REMOTE_IP dev $IFACE lladdr $MAC_REMOTE nud permanent
 ip -6 n replace $REM_VETH  dev $IFACE lladdr $MAC_REMOTE nud permanent
VMAC=`cat /sys/class/net/veth0/address`
 ip netns exec blue ip -6 n replace $REMOTE_IP dev veth1 lladdr $VMAC nud permanent
 ip netns exec blue ip -6 n replace $REM_VETH  dev veth1 lladdr $VMAC nud permanent
```