Eötvös Loránd Tudományegyetem

Faculty of Informatics

Filler

Filler

Filler Title

Supervisor:

Fejes Ferenc

Filler

Author:

Seres Bendegúz

Computer Science BSc

Budapest, 2020

Contents

Chapter 1

Introduction

For cloud computing to become how we know it today, it had to go through a series of steps, each improving upon the previous solution. When mainframes first started to appear, their astronomical price made it practically impossible for companies to give each of their employees a device of their own. However, having only a single person operate the mainfram would lose precious compute time. To solve the problem of allocating the mainframe's resources more efficiently, time-sharing was invented. This allowed employees to use the same hardware in parallel, reducing idle time.

The next big leap in the process of maximizing resource utilization was virtaulization. One of the goals of virtualization was to distribute these finite resources in the most efficient way possible. Virtualisation works by simulating virtual hardware, called a *Virtual Machine* (VM), which then can be used to run a guest operating system on. The parameters of this virtual machine, like CPU, memory, disk space, can be freely configured (limited only by the resources available to the host), and even the network interface controller (NIC) or various networking devices can be simulated or shared with the VM. The problem with virtualization is the need to simulate the whole operating system in each virtual machine.

The idea of the VM was taken further with the introduction of operating-system level virtualization. This approach provided a solution for the problem of needing to unnecessarily simulate the whole operating system. In this model, there are multiple user-space "instances", isolated from eachother, sharing the same kernel. These instances are what we call containers. On Unix-like systems, this abstraction of the filesystem and the resources is done by a more advanced implementation of the *chroot* mechanism, called *namespaces* and a resource-management function, provided by the kernel, called *cgroups*.

The advent of containers have paved the way for the *microservice* model. In this model, each part of the software acts as a self-contained service, that can be easily swapped with another one if needed. The benefits of this approach are numerous, including robustness (if a container malfunctions for some reason, an other can be placed in its place), scalability (more copies of the service can be started or exisiting copies destroyed easily at will), modularity (these services are losely coupled, they can be easily replaced with newer version of the same service, or even a completely different service), etc...

Managing these containers is tiresome, and quickly becomes an almost impossible task as the application grows and becames more complex. A solution was needed that would automize the management of these services, making it easier for developers to concentrate on the application itself. The solution came in the form of the *container orchestration* tools.

The most widely used container orchestration tool is arguably Kubernetes, which was origianlly designed by Google. Kubernetes has the ability to automate most tasks related to the management of containers, like creating and destorying containers based on need.

For these containers to be able to communicate with eachother inside the cluster, a complete virtual network has to be created. This virtual network will be the main focus of this thesis.

1.1 Motivation

The primary goal of this thesis is to provide an easy-to-use solution for limiting the bandwidth between pods in our Kubernetes cluster in a controlled way. To achieve this, it uses a technology called BPF (Berkeley Packet Filter), or as nowadays called, eBPF. eBPF gives you the ability to run mini programs on a wide variety of kernal and application events. It makes the kernel programmable for people without background in kernel development.

For networking capabilities, Kubernetes relies on third-party plugins called CNI (Container Network Interface). These plugins make it possible for containers to connect to other containers, the host or outside the network by automatically configuring the network when containers are created or destroyed. In addition, most of them provide various other functionalities like added security measures They do this by creating an *overlay network* on top of the alrady existing one. An overlay network is just a virtual network, providing the same functionalities as a phisycal network. It functions by encapsulating the network packets in an additional layer.

In the case of smaller clusters, pretty much any CNI plugin will do the job just fine. Howerever, it's a good idea to pick the CNI plugin carefully right at the beginning, because when the cluster grows to contain more then just a handful of services, the plugin you chose can have a great impact on scaling it further, and changing it without breaking the cluster is usually not a trivial task.

There are plenty of CNI plugins to choose from, so choosing the right one requires quite a bit of research. To mention just a few: Cilium, Flannel and Weave Net are among the most popular choices. Flannel is the developed by CoreOS, and arguably the most popular one. It's easy to install, and works by creating a layer 3 overlay network, in which each pod has a subnet for allocating IPs for its containers internally. Weave uses a *mesh overlay network*, which means that each node in the overlay network is linked to multiple other ones. Each host has a routing component installed, and these communicate with eachother, exchangeing information about the topology, so they all stay up to date. Cilium uses eBPF to provide these functionalities and some security related ones. This means that in a lot of cases, it's the fastest solution, because it can inspect and modify the packages inside the kernel, which in turn have huge implications in scaling the cluster.

The problem with current solutions is the lack of functionality regarding network resource management. Network is a finite resource, so it would be great, if we could control the quality of the service provided to customers based on the amount they pay for it. Currently, this is only possible by setting up rules manually, which is practically impossible when you have more than a couple of customers. Furthermore, while providing this functionality inside the CNI plugin itself is possible, this would stop users of other CNI plugins, which doesn't have this function, to use it. My solution provides this functionality regardless of the CNI plugin used, while still being easy to configure and use, using the *cgroup* and *eBPF* subsystems already provided by the kernel.

1.2 Thesis Structure

The thesis consists of the following chapters:
Chapter 2 contains the user documentation. It describes the required dependencies for the project, gives some instructions for setting up and configuring necessary components of the system and explains how to use the software. Chapter 3 elaborates on the approach I used when implementing the software.
Chapter 4 contains some remarks about the development environment and the technologies I used in the project.
Chapter 5 is the developer documentation. This chapter iw written for developers, and its main goal is to help them understand the inner workings of the software, making troubleshooting or even extending the project with new functionalities easier.
Chapter 6 describes the various tecniques used for testing the software at each stage of the development process.
Chapter 7 gives the reader a summary about the outcome of the project.

Chapter 2

User Documentation

2.1 Use Cases

The primary use case of the software is to limit the bandwidth between services inside the cluster.

2.2 Dependencies

In this section, I go through the dependencies that are required by the project. I'm assuming that a Kubernetes cluster is already set up, so I only describe the changes necessary for my project to work, although the Kubernetes section inside the development environment chapter describes how to install and configure a Kubernetes cluster if needed.

2.2.1 Kernel

On both the master node and the worker nodes, a BPF compatible kernel has to be installed. The distribution I've used was *Arch*, which ships with a kernel that has all the necessary kernel features enabled by default.

For distributions which don't have the necessary features enabled by default, the following configuration can be used to compile the kernel:

```
CONFIG_CGROUPS=y
CONFIG_BLK_CGROUP=y
CONFIG_DEBUG_BLK_CGROUP=y
```

```
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_RDMA=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_HUGETLB=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_CGROUP_PERF=y
CONFIG_CGROUP_BPF=y
CONFIG_CGROUP_DEBUG=y
CONFIG_SOCK_CGROUP_DATA=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT_ALWAYS_ON=y
CONFIG_BLK_CGROUP_IOLATENCY=y
CONFIG_IPV6_SEG6_BPF=y
CONFIG_NETFILTER_XT_MATCH_BPF=y
CONFIG_NETFILTER_XT_MATCH_CGROUP=y
CONFIG_BPFILTER=y
CONFIG_BPFILTER_UMH=y
CONFIG_NET_CLS_CGROUP=y
CONFIG_NET_CLS_BPF=y
CONFIG_NET_ACT_BPF=y
CONFIG_CGROUP_NET_PRIO=y
CONFIG_CGROUP_NET_CLASSID=y
CONFIG_BPF_JIT=y
CONFIG_BPF_STREAM_PARSER=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
CONFIG_BPF_KPROBE_OVERRIDE=y
CONFIG_TEST_BPF=m
```

2.2.2 Cgroup v2

Cgroups (or *control groups*) is feature, provided by the kernel, for limiting the amount of resources a process can use. The resources that can be controlled by cgroups include CPU, RAM, block I/O, network I/O, etc… You can use cgroups by manually modifying files inside the `/sys/fs/cgroup` pseudo-filesystem. This folder contains the cgroup hierarchy which is just a series of folders nested inside eachother, with the root cgroup being at the top. Cgroups in the lower end of the hierarchy share the resources of their parents, and can be further limited. The init process, which has the PID 1, sits at the top of the hierarchy. Processes can be added to cgroups by appending their PIDs to the `tasks` file inside the chosen cgroup. Processes also inherit the cgroup membership from their parent process. Because a deeper understanding of the inner workings of cgroups is not required for using the software, I won't go into greater details, but curious readers can easily find more information about cgroups on the internet.

My solution requires the second version of cgroups to work. Compared to the first version, the main differences are the unified hierarchy and the thread granularity. The unified hierarchy means that while in the first version, there were separate hierarchies for each resource types, v2 combines them into a single hierarchy. In v1, there was thread granularity, meaning that individual threads were assigned to cgroups. v2 changed this by only assigning processes to the cgroups.

Although cgroup version 2 was in the kernel as early as 2014 [^fn1], most distributions still hasn't adapted it, and use the first version by default. You have to explicitly tell the kernel that you want the newer version. To do this, a *kernel parameter* has to be passed at boot time:

`systemd.unified_cgroup_hierarchy=1`[^fn2]

There are multiple ways to pass a parameter to the kernel, but for our purpose it makes the most sense to do it through the boot loader. For most users, this will be GRUB, so I'll only show how to do it with GRUB, but information regarding other boot loaders can be found on the Arch wiki[^fn3].

When GRUB starts, and asks you to select the OS you want to boot into, you can press `e` to access the configuration. The kernel parameters have to be placed at the and of the `linux` line:

```
linux   /boot/vmlinuz-linux root=UUID=bf3f606b-e532-4b8a-a552-f90bafd14776
rw loglevel=3 quiet systemd.unified_cgroup_hierarchy=1
```

Pressing `F10` will boot with the modified configuration.

Configuring the kernel in this way will only work until the system is rebooted, at which point the default parameters will be used again. To make the change persistent, the GRUB configuration file has to be updated, which can be found at /boot/grub/grub.cfg.

Although it's possible to manually edit this file, this is usually not recommended, because a small typo could make the system unable to start. The proper way to update GRUB configurations is to first modify the /etc/default/grub file, which is specifically used for user overrides, then use GRUB's own tool to generate the configuration file.

The same linux line has to be placed inside the /etc/default/grub file that was used previously, then the grub-mkconfig -o /boot/grub/grub.cfg command can be used to generate the configuration and override the previous one.

2.2.3 Bpftool

My current solution relies on bpftool, which is a tool used for managing BPF programs. Without it, the bpf() system call would have to be called manually. The sources of bpftool can be found in the kernel repository.[^fn8] It can be installed with make install. Bpftool is a complete tool in itself, so it's usage is not limited to this project. It can be used to list eBPF programs inside the kernel with bpftool prog list. It can load the compiled eBPF object file with bpftool prog load <prog.o> <target>. It's even possible to dump the bytecode for an existing program using bpftool prog dump xlated id <prog_id>. Here is an example how a loaded program might look like after the kernel rewrites it:

```
   0: (bf) r6 = r1
   1: (69) r7 = *(u16 *)(r6 +176)
   2: (b4) w8 = 0
   3: (44) w8 |= 2
   4: (b7) r0 = 1
   5: (55) if r8 != 0x2 goto pc+1
   6: (b7) r0 = 0
   7: (95) exit
```

There are many other things that one can do with bpftool (like managing BPF maps), but it's mostly out of scope for this project.

2.2.4 Docker

Docker is arguably the de facto standard when it comes to container runtimes in the enterprise environment. Although in newer versions of Kubernetes, Docker support is dropped[^fn16] in favor of lighter solutions. Docker is a tool for managing containers, images, builds and more. It's using the libcontainer runtime, which was later "repackaged" as runC. The Docker daemon can be controlled by using its REST API.

After cgroup v2 is mounted, the Docker daemon will most likely fail to start. This is due to the fact that most container runtimes, including docker, still don't support the new cgroup hierarchy.[^fn4] Fortunately, the project's master branch[^fn5] already contains the necessary changes for cgroup v2 support. Compiling it and using the resulting `dockerd-dev` binary solves the problem.

To compile the newest version, clone the respository at https://github.com/moby/moby, then use the Makefile that came with it to compile and install it:

```
# make binary
```

2.2.5 LLVM, Clang

For compiling the the BPF program, my solution uses the *LLVM*[^fn7] compiler collection, although GCC should have BPF support too[^fn6]. For a frontend to LLVM, it uses *clang*.

2.2.6 Kubernetes Python Client

The Kubernetes Python Client[^fn9] is the official client library for Kubernetes. The `requirements.txt` file inside the project's folder contains the exact version I used, so it can be installed with:

```
$ pip install -r requirements.txt
```

Alternatively, the newest version can be installed with:

```
$ pip install kubernetes
```

2.3 Usage

To use the program, a Kubernetes cluster has to be running already. The `ratelimit_master.py` file contains the script that should be started on the master node, while `ratelimit_slave.py` contains the code that runs on the worker nodes. While the `ratelimit_master.py` script can be executed by any user that's able to communicate with the Kubernetes API, `ratelimit_slave.py` can only be executed by root, because attaching BPF programs requires higher privileges.

The program requires no further actions from the user, the program will do the attaching and detaching of the BPF programs automatically when it detects the starting and stopping of pods in the cluster.

The rate limit that the program will assign to the pod, can be specified with a **label** in the pod's YAML file. The label that the program will be looking for is `rate`, and it's value must contain the desired bandwidth in the following format: `"<limit in Megabytes>M"`. For reference, here is an example YAML file, that describes a pod named *test*, with the bandwidth limit of 1 MB/s:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
  labels:
    component: web
    rate: "1M"
spec:
  containers:
    - name: test
      image: dashsaurabh/progressive-coder
      image: quay.io/openshiftlabs/simpleservice:0.5.0
      ports:
      - containerPort: 9875
```

Chapter 3

Implementation

For the solution to be usable in a real-life scenario, it had to be scalable, meaning that managing the bandwidth of 10 pods should take roughly the same amount of effort as managing thousands. So when implementing the program, a great emphasis was placed on ease-of-usage and automation. The user only needs to specify the bandwidth once, in it's yaml file that will be used to create it. Afterwards, the program will make sure that each new pod created with that pod description will have it's bandwidth limited to the specified value.

The bandwidth can be specified with a special label in the pod's yaml file. When the file is used to create a new pod, it is sent to the master node, which notifies an availabe worker node. The worker node creates the pod, at which point a new cgroup, belonging to the newly created pod, appears. At the same time the cgroup is created, a *pod creation* event is fired. This event will cause the ratelimiter, running on the master node, to generate the eBPF program with the bandwidth limitation specified in the yaml file. This eBPF program is sent to the worker node which started the pod. A high-level overview can be seen on the following image.

The main idea that served as the primary goal for the projects was the attachment of the BPF program to the pod's cgroup. After the eBPF program is sent to the appropriate worker node, the ratelimiter creates this attachment.

At this point, the program is loaded into the kernel, and the ratelimiting is already active.

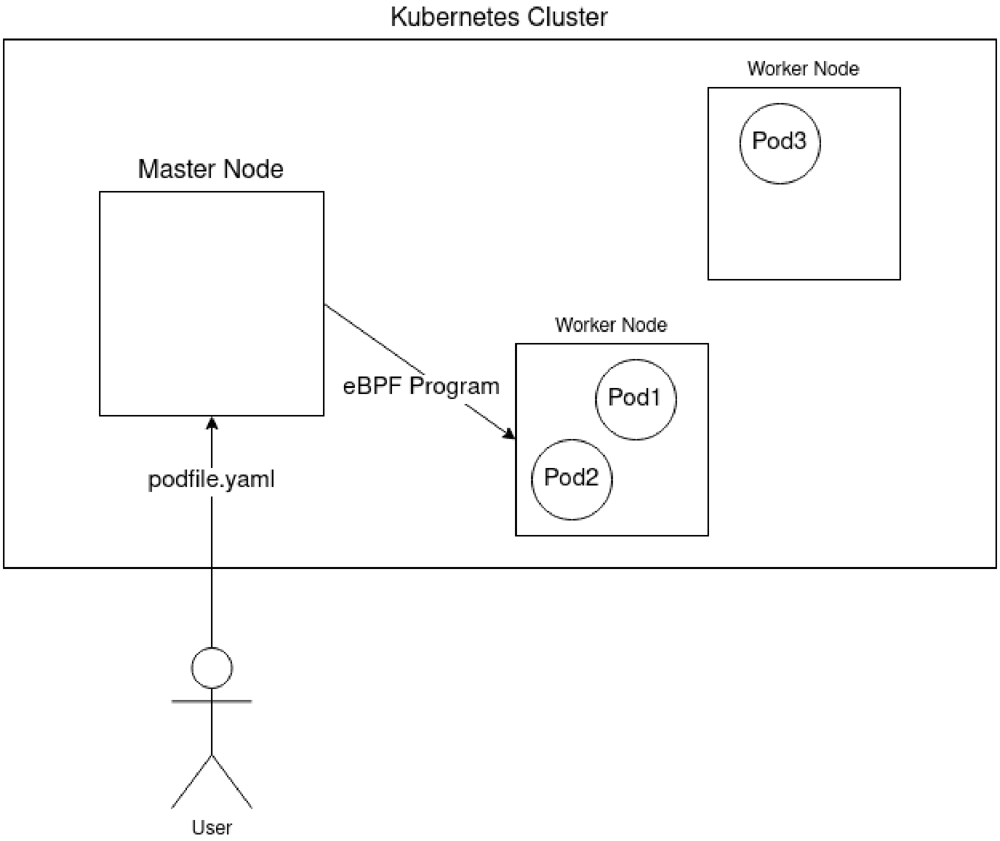


Figure 3.1: Creating a new pod inside the cluster.

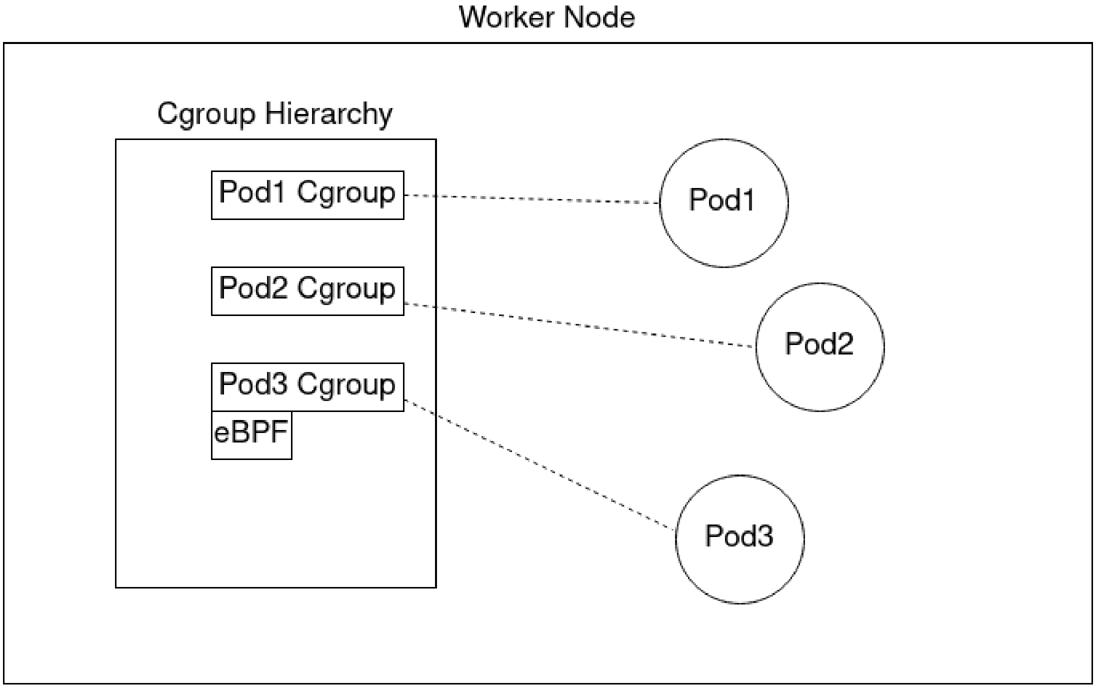


Figure 3.2: Cgroup hierarchy in relation to pods

3.1 Token Bucket Algorithm

The ratelimiting is implemented with the *token bucket* algorithm, which is an algorithm used specifically for limiting bandwidth and burstiness on packet switched networks. It works by the analogy of a *bucket*, that gets constantly filled with *tokens*. Tokens are removed from the bucket at a constant rate (bandwidth). If the bucket is full, the packets are dropped.

```c
static __s64 tokens = 1250000;
static __u64 time = 0;
static const __u32 bps = 125000; //speed in bytes/sec
static const __u32 extra_tokens = bps >> 9; //few extra tokens for smoother operation

int pkt_tbf(struct __sk_buff *skb)
{
  __u64 now = bpf_ktime_get_ns();
  __u64 new_tokens = (bps * (now - time)) / 1000000000;
  tokens += (new_tokens + extra_tokens);
  tokens -= skb->len;
  time = now;
  if(tokens > bps)
    tokens = bps;
  if(tokens < 0)
  {
    tokens = 0;
    return DROP_PKT;
  }

  return ALLOW_PKT;
}
```

In the case of TCP traffic, the dropping of packages results in those packages having to be resent by the sender. TCP's inbuilt mechanism for dealing with packet loss will cause it to adjust it's rate or transmission, which will eventually be identical to the bandwidth limit created by the token bucket algorithm.

In contrast to TCP, dropping these packets in UDP traffic results in information being lost, as in UDP, no validation of the packet being received is done, so it won't get resent.

Chapter 4

Development Environment

For the initial experimenting phase and later testing if the function implemented was working properly, *Qemu* and *Vagrant* were used. These tools helped by emulating the target machine, without the need to use real hardware.

4.1 Qemu

As mentioned before, Qemu (Quick emulator) is a virtual machine emulator, similar to Virtualbox or VMWare. It supports booting with a custom kernel directly from the host. This eliminates the problem of copying the kernel inside the VM after every modification, or compiling the kernel inside the slower VM. When I started the project, I used Qemu for testing newer kernel functions. As most of the features that the project requires are still not widely adopted, I had to compile the kernel with the required features explicitly turned on. Qemu made it convinient to test kernel features without affecting my own host operating system.

4.2 Vagrant

At a later phase, when the project's direction was sufficiently clear, I switched to Vagrant. Vagrant automates most of the steps required to bring up a suitable environment (like automatically syncing the project's directory to the VM). This made it much easier and faster to experiment with new ideas and features.

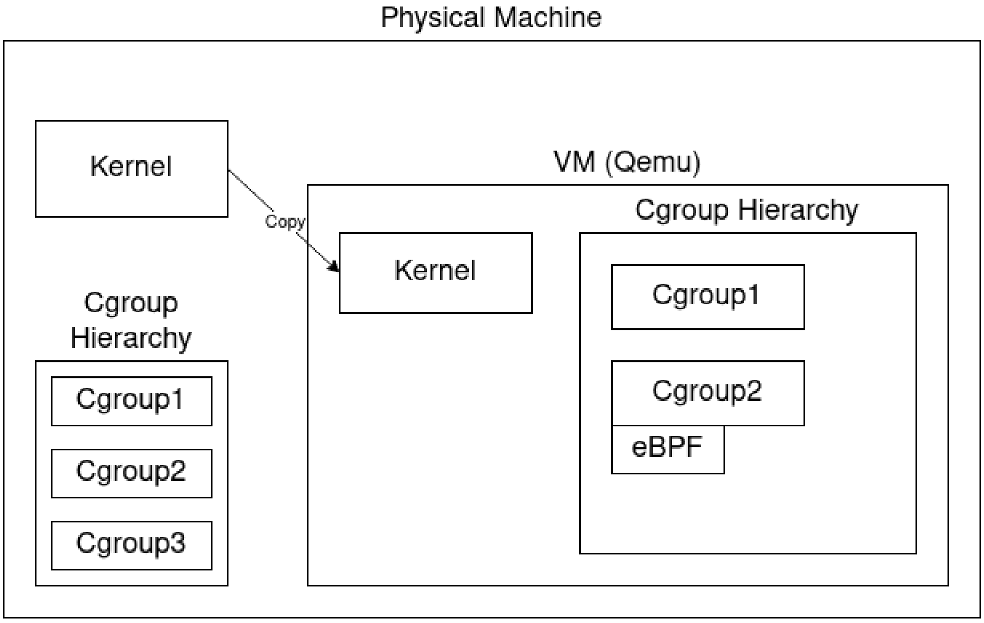


Figure 4.1: Emulating the target device using Qemu

4.3 Docker

I chose Docker as the container runtime, mainly because its wide adoptation in the industry, and the number of available resources for it. As of November 2020, the official release of the docker daemon still not supports the second version of *cgroups*, which my solution relies on, so I had to use the development version.

4.4 Kubernetes

As it turns out, installing and configuring a Kubernetes cluster is not a trivials tasks. That's the reason why for example Minikube or Microk8s exists. These tools make it easy to have a fully functional cluster with minimal or even zero configuration. Due to the nature of my project, I couldn't use these off-the-shelf solutions, so I had to create a working cluster myself.

To create the cluster, first the required binaries have to be installed. There are two ways to create a cluster: you can do it manually, or you can use `kubeadm`. Because time was pressing, and because I wouldn't have gained any additional benefits from creating the cluster by hand, I chose the second option. On Arch linux[ˆfn13], the following packages had to be installed:

- kubectl-bin
- kubelet-bin
- kubeadm-bin
- cni-plugins
- conntrack-tools

The kubelet-bin and kubeadm-bin packages have to be installed on each node in the cluster. These two programs will be used to create and maintain the cluster. The kubectl-bin package contains the kubectl tool that can be used to communicate with the Kubernetes API. cni-plugins and conntrack-tools are needed for networking inside the cluster.

When all the required packages are installed, there are two more requiremenets that needs to be fulfilled before using kubeadm to do most of the work. First, swap has to be turned off. This is becauser the Kubernetes scheduler, that's tresponsible for the availability of the pods, should know about the swap configurations for it to be ale to guarantee a pod's availabilty. This would needlessly complicate things, so the community decided that swap support will be deferred for a later date, when the advantages of this feature would be big enough.[ˆfn11] Swap can be turned off with:

```
# swapoff /dev/<swap_parition>
```

After swap is turned off, the kubelet service has to be started. With systemd, this can be done with:

```
# systemctl start kubelet.service
```

To have the service start automatically after reboots, enable the service:

```
# systemctl enable kubelet.service
```

Now kubeadm should be able to initialize the cluster:

```
# kubeadm init --apiserver-advertise-address=0.0.0.0
--ignore-preflight-errors=SystemVerification
```

Here, the --apiserver-advertise-address option tells the kubeadm where should the cluster's API listen for incoming messages. The

`--ignore-preflight-errors=SystemVerification` line tells the kubeadm to ignore some errors, which was # TODO

After kubeadm finished with initializing the cluster, it will print some commands for configuring kubectl to use the cluster's API.

At this point, the cluster should already be running, but as the last step, it's recommended to install a CNI plugin. CNI (Container Networking Interface) is the specification that containers use to communicate with eachother. Kubernetes uses this specification to make communicatoin between the pods inside the cluster possible.[^fn12] By default, Kubernetes comes with its own CNI plugin, called kubenet. The only problem is that this plugin has very limited capabilities, so it's always a good idea to install a more advanced CNI plugin. Because CNI is only a specification, there are multiple ways to implement it. This is why many CNI plugins exist, each implementing the specification in a different way.

The CNI plugin has to implement the following rules:

- All containers can communicate with all other containers without NAT
- All nodes can communicate with all containers (and vice-versa) without NAT
- The IP that a container sees itself as is the same IP that others see it as

The plugin I chose was Weave Net, which can be installed after kubectl has been configured with the following command:

```
$ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl
version | base64 | tr -d '\n')"
```

If everything went according to the plan, the cluster is running and fully functional now.

Chapter 5

Developer Documentation

The projects contains the following modules and files:

5.1 ratelimit_master.py

This module contains the logic that runs on the cluster's master node. When its started, it first loads the cluster's configuration, then it begins watching for events in the cluster. If it detects the starting or stopping of a pod, the corresponding function is triggered. If the event it detected was the deletion of pod, it sends a message to the right node, that the pinned BPF program should be removed. In the case of a pod creation event, first the BPF program with the value for rate limiting, which is given in the pod's configuration file is created, then it's sent to the node, where the pod was created, together with information that tells the node how to attach the program.

5.2 ratelimit_slave.py

The `ratelimit_slave.py` module runs on each worker node. When it's started, it creates a server socket, then blocks, waiting for incoming connections. Based on the message it gets from the establised connection with the master node, either the `__attach()` or `__detach()` function is executed. When the master node want's to attach a new BPF program, it sends the pod's unique ID, the size of the BPF program and the BPF program itself to the slave.

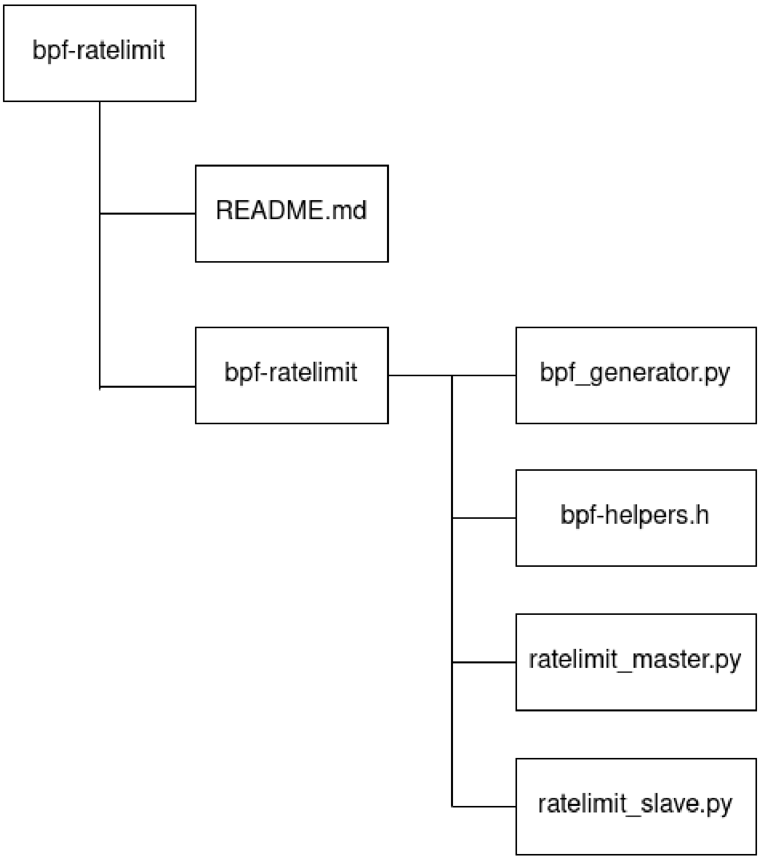


Figure 5.1: Folder Structure

The pod's ID tells the slave where should the program it received be attached. Kubernetes gives every pod it manages a unique ID. This ID can be used to find the pod's *cgroup*. When Kubernetes is running, a cgroup is created in the `/sys/fs/cgroup` pseudo-filesystem named `kubepods.slice` (assuming cgroup v2 is used). In this cgroup, there are additional cgroups for each *pod QoS class*, namely:

- Guaranteed
- Burstable
- BestEffort

Finally, inside these cgroups are the cgroups belonging to the actual pods. The full path to a pod's cgroup could look something like this:

`/sys/fs/cgroup/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-po`

This could be elaborated further still, as the individual pods' cgroups contain the cgroups of the containers that are inside the pod, but that's outside the scope of this project.

The size of the BPF program is sent next, because without it, the slave wouldn't know when should it stop expecting more packages containing the BPF program. After the program is sent to the slave, it places it in the `/sys/fs/bpf` directory, then tries to attach it to the cgroup of the pod. To avoid name collisions if multiple BPF programs are sent to the same machine, the programs are placed inside their own folder, for which the following naming convention is used:

`/sys/fs/bpf/<unique-id-of-pod>`

At almost every stage of the communication between the master and the slave, the slave sends back a response, telling the master if it can proceed with the next step, or some error occurred.

Compared to attachment, the detachment of the program is much more straightforward, requiring only a few steps. As the cgroup of the pod is taken care of by Kubernetes, the only remainig artifact is the BPF program inside `/sys/fs/bpf`. First, similar to the attachment, the unique ID of the pod is sent by the master. With this information, the slave removes the remaining BPF program.

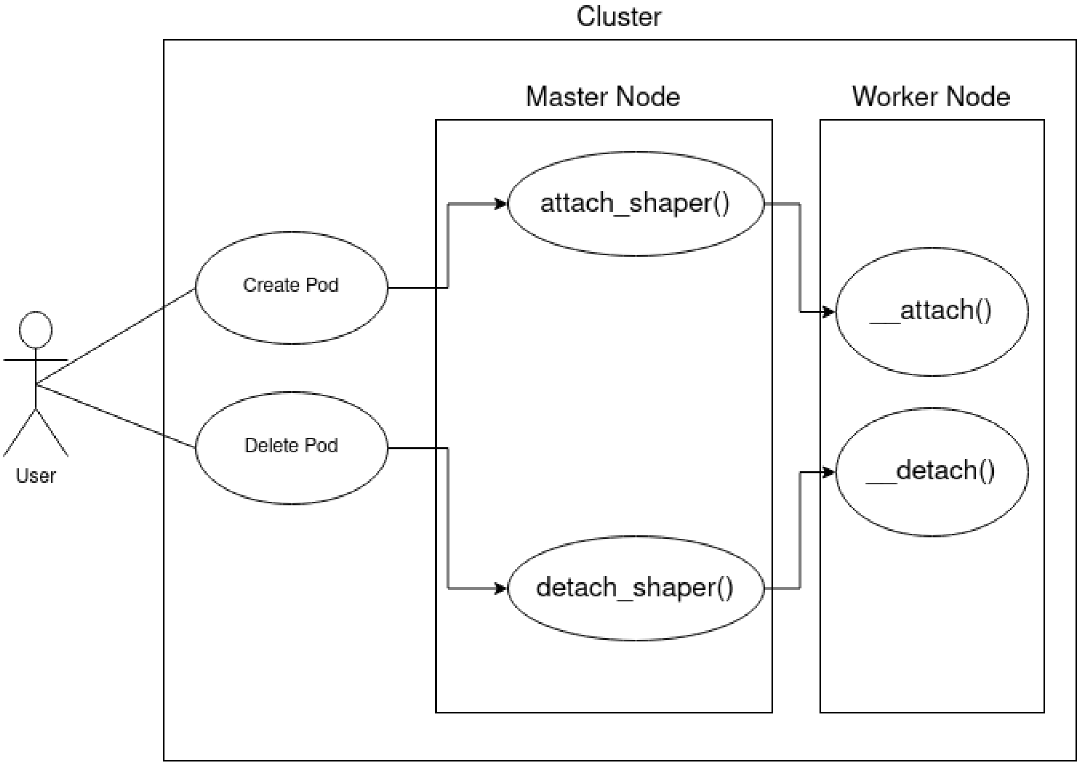


Figure 5.2: Use-Cases and flow of control

5.3 `bpf_generator.py`

This module is responsible for generating the BPF program that will be attached to the pod's cgroup. It contains the source code of the BPF program that will be generated, with the network rate part left out, so it can be dynamically filled in by the script.

5.4 `bpf_helpers.h`

The `bpf_helpers.h` header file contains some helper functions that are needed for compiling the BPF program. The `bpf_generator.py` module links to it while generating the program.

5.5 Use-Cases

The following diagram depicts the use-cases of the program and the flow of control for each case.

The project's repository can be found at https://github.com/EarlPitts/bpf-ratelimit.

Chapter 6

Testing

The make sure that the project was proceeding in the right direction, the program had to be tested at each stage of the development process. As the project was moving forward, testing the program to see if it produces the desired functionality was becoming more and more elaborate.

6.1 Marker

The first goal of the project was to be able to compile and load the BPF program into the kernel in a suffieciently consistent manner, as it was the foundation of everything that followed. This was done using the host operating system, using a BPF program that marked the outgoing packets. These packets could be observed with a tool like `tcpdump`, or rules could be added with `tc` to only allow the marked packets to leave the machine.

Next, the same was done using a virtual machine. With `tc`, rules were declared on the virtual machine that only allowed the marked packets to get through.

6.2 Shaper

At the next stage of the project, the marker got replaced with the *shaper* program, that would remain in the final solution. This meant that not the packet itself, but the number of packets leaving the machine were affected, so the prvious testing method couldn't be used anymore. To test if the shaper was working as expected, various network monitoring tools (like `bmon`[^fn14]) and a tool called speedtest-cli[^fn15] were used. These tools were able to precisely measure the incoming and outgoing rate, which could be used to tell if the shaper was working correctly.

6.3 Containers

Under the hood, if you remove the multiple layers of abstractions, you will find that Kubernetes uses the same old container concept that you are probably familiar with. So the next logical step was to test the shaper on a container. To achieve this, the shaper was attached to the cgroup of the container. Then the same tools were used to test the bandwidth, only this time it had to be done from the context of the container. Fortunately, the docker frontend provides multiple commands that make this task almost trivial. The `docker exec` command can be used to execute arbitrary command inside the container. With the `-i` and `-t` flags it can even make this "interactive" (meaning that STDIN will remain open even it the container is not attached) and a pseudo-tty can be allocated respectively. The command expects a container as argument, and a command that we want to execute inside the container. This way, we can easily get a shell inside the container with the following command:

```
docker exec -it <container_id> sh
```

> *If the container has a more user-friendly shell like bash, that could be used instead.*

Now that we are inside the container, the aforementioned tools can be easily installed with the help of the available package manager. With the tools present, the same measurements can be taken as before, confirming that the shaper functions correctly with the container.

I used *netcat* for generating network traffic, because it's small size and ubiquity. I simply started netcat in listening mode on the recipient side, dumping it's output into `/dev/null`, and started netcat on the sender side, redirecting `/dev/random` into it.

6.3.1 Container from Host

In this scenario, the recipient end of the communication was inside a container. The image I used for setting up the container didn't contain the `ip` tool, so I checked the IP address of the container from the host with `ip neighbor`. This lists all the hosts on the local network. Docker uses the `172.17.0.0/24` network range, so it was easy to find the only IP that was in that range.

Container

- `docker run --rm -it -p10000:10000 ubuntu bash`
- `nc -l 0.0.0.0 10000 > /dev/null`

Host

- `nc 172.17.0.2 10000 < /dev/random`

6.3.2 Container from Container

This time traffic was sent from a container to an other one.

Container 1

- `docker run --rm -it -p10000:10000 ubuntu bash`
- `nc -l 0.0.0.0 10000 > /dev/null`

Container 2

- `docker run --rm -it ubuntu bash`
- `nc 172.17.0.2 10000 < /dev/random`

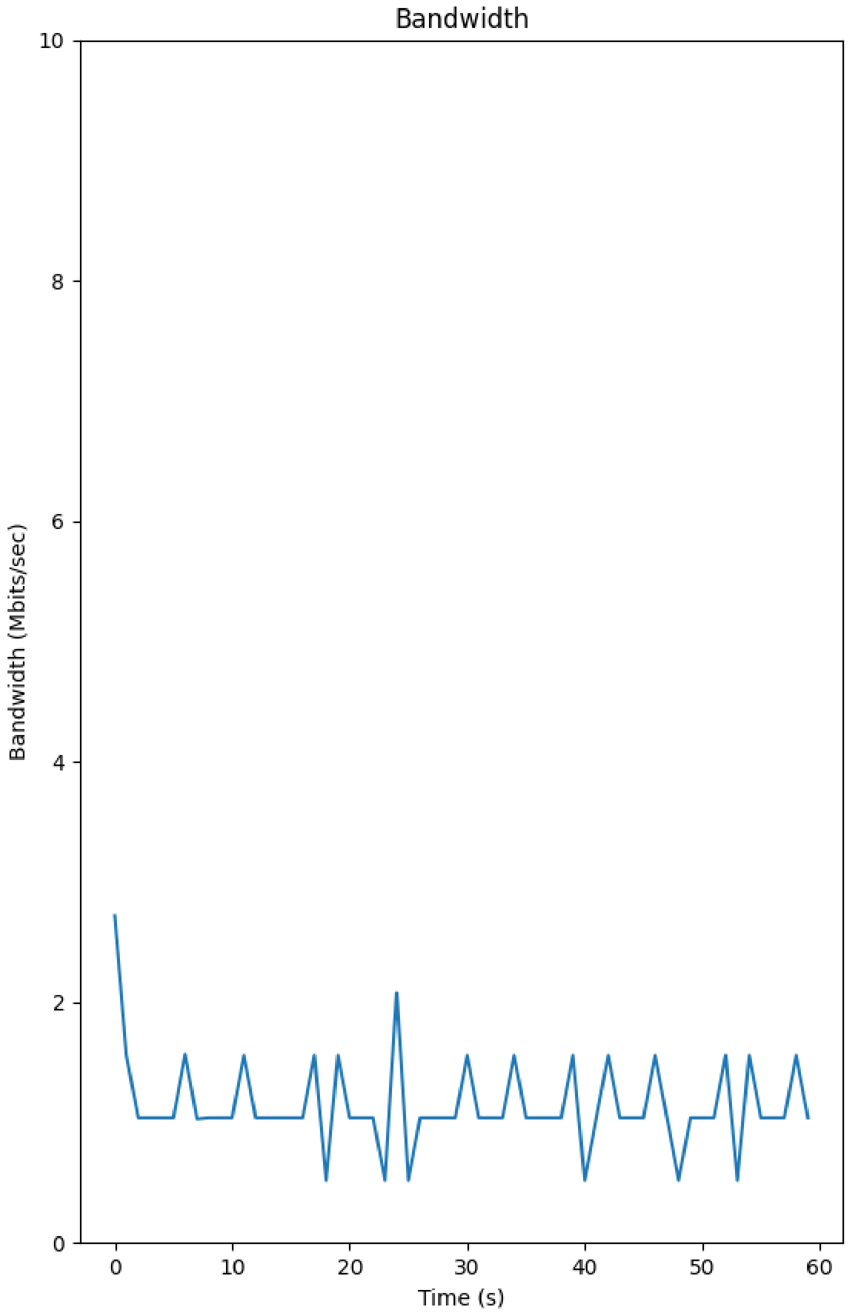
6.4 Kubernetes

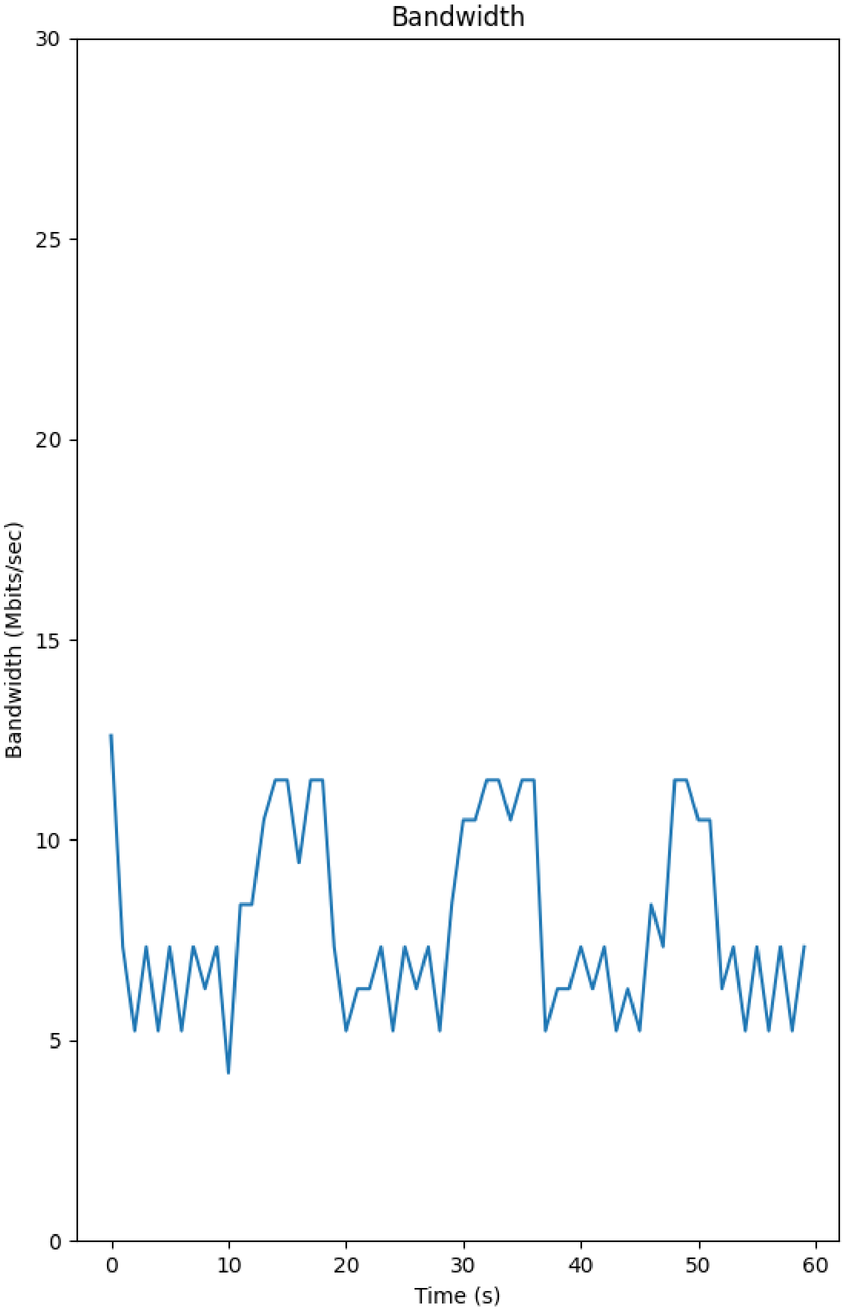
For testing in the Kubernetes cluster, I created two custom docker images, one with netcat listening, similar to the method I used when testing the containers, and one with *iperf*.

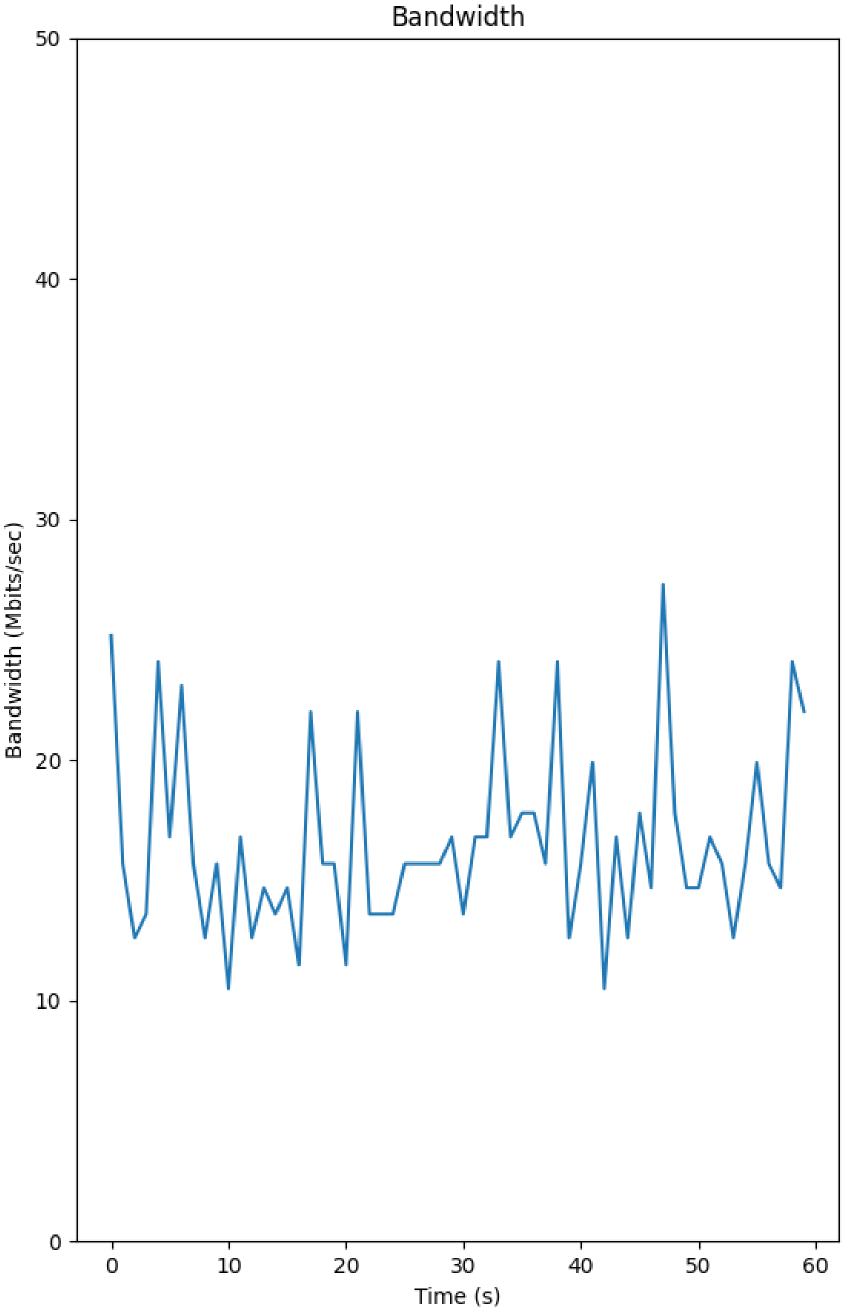
```
FROM alpine:3.4

RUN apk update
RUN apk add netcat-openbsd

EXPOSE 10000/tcp

CMD ["nc", "-lk", "0.0.0.0", "10000", ">", "/dev/null"]
```

```
FROM alpine:3.4

RUN apk update
RUN apk add iperf

EXPOSE 5001/tcp

CMD ["iperf", "-s"]
```

These images were used for creating the pods, then an other pod was used to send traffic to them. Several pod yaml file was created, each with different bandwidth limit defined in them. The results were plotting using Python's *Matplotlib* library.


Bandwidth
Bandwidth (Mbits/sec)
0
2
4
6
8
10
0
10
20
30
40
50
60
Time (s)


Bandwidth
Bandwidth (Mbits/sec)
0
5
10
15
20
25
30
Time (s)
0
10
20
30
40
50
60


Bandwidth
Bandwidth (Mbits/sec)
0
10
20
30
40
50
Time (s)
0
10
20
30
40
50
60

Chapter 7

Summary

7.1 Results

Towards the end of the project, Cilium, a company providing eBPF based solutions for networking, announced a new release, which, as one of its new features, implemented the same functionality as the one my solution provides.[ˆfn10] Although it was rather demotivating first, seeing that someone has done what I was working on, it nonetheless proved that the project was solving a real-world problem.

The project was a great opportunity to learn about networking concepts more in-depth. TODO

7.2 Further Plans

Although the project is in a usable state, there are plenty of room for improvement. As closing thoughts, I would like to briefly mention some ways the project could be made better.

7.2.1 XDP