Eötvös Loránd University

Faculty of Informatics

Institute of Computer Science

Department of Information Systems

Network resource management in Kubernetes

Supervisor:
Ferenc Fejes
PhD Candidate

Author:
Kristóf Aranyos
Computer Science BSc

Budapest, 2021

Contents

Chapter 1

Introduction

1.1 Motivation

Most services available on the world wide web today are running on some kind of autonomously scalable container orchestration system. One of the most widely used engines is Kubernetes, opted for by several industry leaders.

Kubernetes itself does not include networking, offloading this task to CNIs (Container Network Infrastructure). There are several of these freely available (such as Calico [1], Cilium [2], Flannel [3], Weawe Net [4], etc) but the feature set of these is wildly varying and their configuration is inconsistent. One of their deficiencies is that they do not support introducing bottlenecks in the network in order to test the quality of the services in, for example, a low-bandwidth or high-congestion networking environment.

1.2 Results

The goal of the thesis is to develop an application capable of simulating these conditions on Pod or Service level. The application is connected to the Kubernetes API, listening to Pod create, update and delete events. The Pods' containers get attached an eBPF program according to their metadata. These eBPF programs then run in kernel space and allow or drop packets to limit bandwidth and simulate network congestion.

To prove the effectiveness and correctness of the application, it will be benchmarked on various different systems and with multiple tools.

1.3 Structure

The thesis will consist of three main parts; User Guide, Developer Documentation, and Benchmarks.

The User Guide will describe the general ideas behind the application as well as the system requirements, the installation guide, how to configure the limits for the system, and how to run the application.

The Developer Documentation will contain a detailed description of the Kubernetes ecosystem, the Linux and eBPF ecosystem, and the details of the application's inner working. This latter will include a description of the implemented filters as well.

Lastly, the Benchmarks chapter will include detailed measurements of the application's performance and it's network managers' effectiveness. Both the bandwidth and loss manager is measured and the measurements are compared to the target values.

Chapter 2

User Documentation

The User Documentation contains a basic overview of the application, the system requirements, an installation guide and instructions on how to use the application.

2.1 System requirements

2.1.1 Hardware requirements

The application itself is lightweight and does not require strong hardware. It however requires a Kubernetes cluster to work on, therefore Kubernetes' system requirements apply to it. It is worth to note, that serving tens of thousands if not more users (which Kubernetes clusters are intended for) needs multiple strong server machines. For testing purposes, a single pc or laptop will suffice.

There are two kind of Kubernetes nodes: master and worker nodes. In every cluster, there is one master node, which controls the workers. There is no limit on worker count. It is possible to run a master and a worker node on one computer to avoid needing multiple machines.

Hardware requirements of Kubernetes master nodes:

- CPU: At least two cores @ 1 GHz. X86-64 recommended, ARM64 might work, but it is untested.
- RAM: At least 2 GB of free memory.
- Disk space: At least 2 GB recommended for the tooling (k8s-core, cert-updater, fluentd, kube-addon-manager, rescheduler, network, etcd, proxy, kubelet).

Hardware requirements of Kubernetes worker nodes:

- CPU: At least one core @ 1 GHz. X86-64 recommended, ARM64 might work, but it is untested.
- RAM: At least 700 Mb of free memory.
- Disk space: At least 1 GB recommended for the tooling (fluentd, dns, proxy, network, kubelet).

2.1.2 Software requirements

While Kubernetes runs on Windows as well as on Linux, generally most servers will run the latter, and a core part of the application - the eBPF programs - also run in the Linux kernel. Therefore, Linux is required and while the code might compile on Windows or macOS, it certainly will not run on them.
Another thing worth mentioning is that eBPF requires a fairly new kernel to run as it is a bleeding edge technology. The development took place on a computer running Manjaro 20.1.1 with Linux 5.10.15 kernel.

Software requirements of the application:

- Linux with Kernel [5] 5.10 or newer
- Kubernetes [6] v1.20 or newer, K3S [7] is recommended for on-premises clusters
- Clang [8] 10.0 or newer
- BPF tools [9]
- LibBPF [10]
- Go [11] 1.15 or newer

2.2 Overview

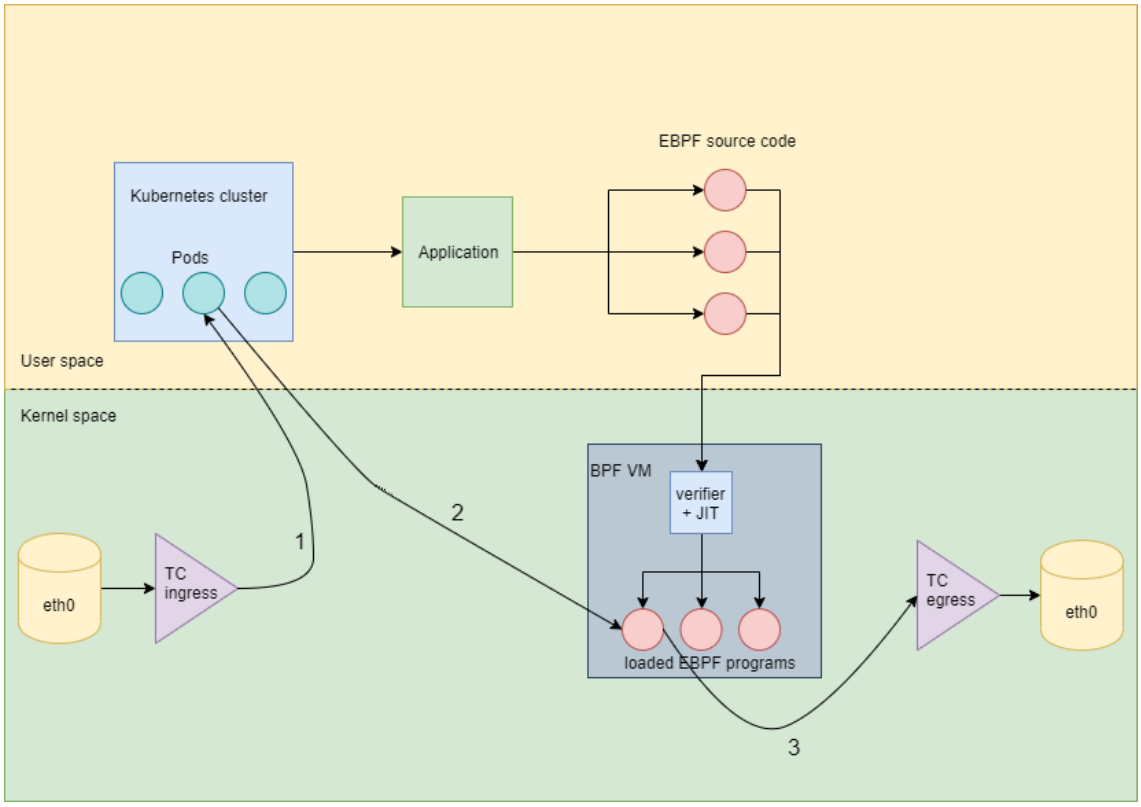

Figure 2.1: Application architecture

Simply put, the application manages the traffic flowing through a Kubernetes cluster according to a set of rules. These rules can be as simple as applying a bandwidth limit, or dropping given packets. They can also be much more complex through user defined programs, if required.

The center of operation is an executable written in Golang that uses the Kubernetes API to constantly monitor the state of the cluster. On startup, it fetches all Kubernetes services and pods, reading their metadata to decide if a limit should be set on them. All limits are applied on the pods themselves, even service ones. Services are linked to a set of pods, their limits are inherited by the pods, though if a pod has colliding limit, the pod-level one is applied.

Since a pod is not one process, but an abstraction above closely related ones, it is not possible to simply control their traffic. Instead, even though they are treated as one unit, the underlying processes are iterated and all of them are set to the same limits.

The process of applying limits is the following:

1. Finding the pod on which the limit gets applied
2. Iterate over the processes of the pod
   (a) Get the cgroup of the process
   (b) Compile a custom-tailored eBPF program using pre-written templates
   (c) Load the eBPF program into the Linux kernel
   (d) Attach the eBPF program to the cgroups' socket buffer hooks
3. Save the limit info of the pod

The reason each pod is required to compile its own eBPF program in **step 2B** is due to an inherent property of the eBPF system: it does not support parametrizing the programs. This is bypassed by the application using C macros that are set compile-time; further info about this in the Developer Documentation part.

2.3 Installation guide

For development, Manjaro 20.1.1 with Linux 5.10.15 kernel was used. The Installation Guide will also be based on this.

2.3.1 Initial setup

Linux kernel version

The first step is to ensure a new enough linux kernel is running on the machine. The features used in the eBPF programs require a kernel version newer than 5.10. To check, open a terminal and run

```
$ uname -r
```

This outputs the version of the currently running kernel. If the kernel version is too old, a simple way to update it is to use the Manjaro Settings Manager. After opening it, click on `Kernel`. It will show a list of kernels available to be installed. Choose a version of 5.10 or over. Click `Install`, then restart the machine and check the version again.

Required software

If the kernel is up to date, the next step is to install the required tools that will be used to compile the Go application, the eBPF programs - which, by the way are written in a subset of C - and load them into the kernel. The easiest way is to open a terminal and run

```
$ sudo pacman -S base-devel go clang bpf libbpf
```

This will install every required piece of software, namely:

1. `base-devel`: This is a basic bundle of software that is used by virtually everything else.
2. `go`: The Golang compiler and tooling. Required for compiling the application itself.
3. `clang`: A C compiler used to generate the object files from C source code.
4. `bpf`, `libbpf`: These utilities are used to load the object files into the kernel.

2.3.2 Kubernetes Cluster

Kubernetes - being a complex software bundle - requires the help of guides to install. Since it is a core requirement, a short one is included in the thesis.
The project uses K3S due to the ease of setup, as it bundles a CNI (Flannel) and Ingress controller (Traefik [12]). These require little to no configuration - something that is a severe problem when using raw kubeadm, kubelet and kubectl utilities.

Using the bundled executable

The easiest way to set up the cluster is to simply run the bundled executable. This is located in the *k3s* folder. Normally, including executables is to be avoided; in this case, the main motivations were a simple setup procedure and the fact that at the time of writing the application, the latest K3S releases were not capable of using Cgroup v2. The fix was already done, a release had yet to be published.
To start the Kubernetes cluster, open a terminal and run

```
$ sudo k3s/run.sh
```

This will start the master node with an additional worker node. The ramp-up period depends on the speed of the machine, generally, half a minute should be enough for it.

Installing natively

A more elaborate and recommended way to install the Kubernetes cluster is to follow the official way. This will set up a service on the machine that will by default start up on boot. This spares the burden of having to manually start the executable each time. To install the K3S service, open a terminal and run

```
$ curl -sfL https://get.k3s.io | sh -
```

The install script will automatically set up everything required to run the service, this usually takes around a minute or so. After installation, the cluster will automatically start to run, creating a master and a worker node. Ensure the service is running using

```
$ systemctl status k3s.service
```

Configuring the cluster

Configuring the cluster itself is depenent on intent. For development or production clusters, this part can be skipped entirely as it follows the configuration of the example bundled service.

The example service is kept as minimal as possible to keep complexity down. The server binary itself listens on `port 80`, and the `config.yml` file contains directives to run this executable. A pod is created to serve as a platform the executable can run on, and a service is masking the pod towards the ingress controller. This will be explained further in the Development Documentation. Traefik is used as an ingress controller (a reverse proxy), to interface the service to the real world. All these elements reside inside the thesis-ns namespace to differentiate them from everything else and keep things tidy.

To apply the configuration, run the

```
$ sudo k3s/k3s kubectl apply -f k3s/config.yml
```

or

```
$ sudo k3s kubectl apply -f k3s/config.yml
```

command in a terminal depending whether you used the executable or installed the k3s service.

By default, K3S will put the kubeconfig file - which will be needed later - into the `/etc/rancher/k3s/config.yaml` folder. It is recommended to move this into the home directory for ease of use, before it is forgotten. This can be done by running

```
$ cp /etc/rancher/k3s/config.yaml ~/.kube/config
```

The last thing to configure is the cgroup v2. It is usually not enabled by default. To do so, the following steps can be taken.

1. `$ sudo nano /etc/default/grub`
2. Append `cgroup_no_v1=net_prio,net_cls systemd.unified_cgroup_hierarchy=1` to the end of both `GRUB_CMDLINE_LINUX_DEFAULT` and `GRUB_CMDLINE_LINUX` lines.
3. Save and exit
4. `$ sudo update-grub`
5. Reboot the machine

After rebooting, make sure it is actually set up correctly. It can be checked using the following command.

```
$ mount | grep cgroup
```

The output should resemble `cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,relatime)`.

2.3.3 Compiling and running the executable

Compiling the application

The application itself is written in Golang and can be compiled and run either in terminal, by systemd [13] daemon or an IDE of your choice (Goland [14] is recommended).

The newest dependency management software of Golang is called Go modules. As the name states, projects are organized into a coherent entity called a module, which other modules can depend on. A dependency tree is built from these, with the current module at its root. This project also uses go modules.
Go modules have some rules that help to standardize compilation and program entry points. To compile the application, simply run

```
$ cd ebpf_loader && go build -o podmgr .
```

This will create an executable in the current folder that can be then directly run by stating

```
$ sudo ./podmgr
```

The application always requires root privileges because it calls commands that interact with the system directly, such as the loading of eBPF programs into the kernel and managing cgroups.

Adding a daemon

It is possible to create a daemon for the application which would start automatically upon boot. Nowadays, the most popular init system is systemd, but sysvinit, OpenRC and upstart daemon are also supported.

To make a systemd daemon, create a .service file in the `/etc/systemd/system` folder, and set the ExecStart parameter to the executable file.

2.4 Using the application

The application uses the Kubernetes API (Application Programming Interface) to continuously monitor the cluster. This API requires authentication, provided by the kubeconfig file. By convention, this file resides in the `/.kube/config` directory. Section 2.3.2 explains where this file is located when using a K3S installation.

Alternatively, the location of the kubeconfig file can be passed to the application through the kubeconfig command line flag. This is useful when the location of the file can not be changed.

2.4.1 Effect level

There are several ways the application can affect the cluster, however a common thing between them is that they all act on pod level regardless how they are configured.

There are two ways to apply the rules. The first one is directly applying them to the pods, and the second way is to apply them to services. Services are abstractions over multiple pods, and they can be used to have them all get the same settings without having to repeat the rule.

When both service- and pod level rules are set, the pod level ones overrule the service ones. In all other cases, trivially only the set rules are applied.

The rules themselves can be applied through appending metadata in the yaml (YAML Ain't Markup Language, a type of config) files that describe Kubernetes components.

It is desirable to avoid using multiple rules on one pod; though it is possible to make them work together, collisions or conflicts may still happen. As a rule of thumb, the less logic is in the way of the transmitted data, the faster the connection is.

Adding a pod level rule

Pod level rules are directly set on the pods. Simply add a metadata member called `annotations`, then list the rules to be applied. Deployments are also supported, in this case the annotations go into the template metadata.

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    app: myapp
  annotations:
    bandwidth: "10 mbps egress"
spec:
  ...
```

Code 2.1: Examle yaml file describing a pod with a 10 Mbps bandwidth limit

Adding a service level rule

Service level rules are - like the name says - set on services that mask the pods. This allows the rules to be applied to several pods without repetition of the rule. Add a metadata member called `annotations`, then list the rules to be applied.

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
  annotations:
    bandwidth: "10 mbps egress"
spec:
  ...
```

Code 2.2: Examle yaml file describing a service with a 10 Mbps bandwidth limit

2.4.2 Using the Bandwidth manager

The bandwidth manager is a built-in rule that acts as a low-pass filter by setting a maximum bandwidth that a pod can use. This can be useful for a variety of purposes.

- Testing how a specific service reacts to low-bandwidth environments.
- Applying QoS (Quality of Service) policies
- Keeping the network load below a set value (cloud providers often bill network usage)

Internally, the bandwidth manager uses TBF (Token Bucket Filter) to drop packages that do not fit in the bandwidth. These dropped packets by themselves are already enough to maintain the limit, but when using TCP - and generally, web services use TCP - it will also throttle back the speed of the connection, sending packets slower and further regulating the limit.

Adding a bandwidth limit

To add a bandwidth limit to a pod or service, open the yaml config file of it, and add a `bandwidth` key in the annotation section.

```
...
metadata:
  annotations:
    bandwidth: "10 mbps egress"
...
```

Code 2.3: Example bandwidth limit of 10 Mbps

The value of the key follows the pattern of

- speed - `integer`, the maximum bandwidth.
- unit - `string`, could be any of `bps`, `kbps` and `mbps`.
- interface - `string`, either `ingress` or `egress`.

2.4.3 Using the Loss Manager

The Loss manager is also a built-in rule, dropping packets randomly according to some settings. The default behavior uses uniform distribution and the dropped packet ratio can be given as a percentage. For example, a loss of 5% would drop every twentieth packet on average - of course, this varies due to the nature of the random numbers. The loss manager can be an useful tool in several cases.

- Testing how a specific service reacts to high-congestion environments.
- Simulating high packet loss, for example caused by poor Wi-Fi reception.
- Simulating misconfigured networks or software.

The loss manager uses a PRNG (pseudorandom number generator) supplied by an BPF utility which supplies 32 bit long data. This is then converted to percentage by taking modulo 100 of it. The output can then be directly compared.

Adding a loss ratio

To add a loss ratio to a pod or service, open the yaml config file of it, and add a `loss` key in the annotation section.

```
...
metadata:
  annotations:
    loss: "uniform egress 2%"
...
```

Code 2.4: Example loss of 2 percent

The value of the key follows the pattern of

- distribution - `string`, either `uniform` or `exponential`.
- interface - `string`, either `ingress` or `egress`.
- threshold - `string`, for uniform distribution, a `percentage`, for exponential, a `value in [0, 1000]`.

2.4.4 Using custom programs

Custom programs can be written to further extend the functionality of the application. This enables users to police the in- and outbound traffic as they see fit without having tho write a surrounding mechanism that controls the loading and unloading of these programs. These custom programs can control traffic based on virtually anything that conforms to the eBPF rules.

Writing the eBPF program

The eBPF programs are written in a subset of C. Specific rules apply to the code, this is further explained in Section 3.2.2. More information about writing these programs can be found in Cilium's eBPF reference guide [15].

Note, the interface (ingress, egress) is automatically applied as the `INTERFACE` macro in these programs.

Applying the rules

To add a custom program, open the yaml config file of the pod and add the `ebpf-limit` key in the annotation section.

```
1 ...
2 metadata:
3   annotations:
4     ebpf-limit: "dropper egress -DDROP=1"
5 ...
```

Code 2.5: Example of a custom program applied

The value of the key follows the pattern of

- name - `string`, the name of the program.
- interface - `string`, either `ingress` or `egress`.
- parameters - these are passed to Clang during compilation, useful for setting macros.

The macros follow the pattern of `-DNAME=value` where -D is constant, NAME is the name of the macro and the value is what the name gets replaced with.

2.4.5 Shutting down the application

To quit, simply press Ctrl + C in the terminal or if a systemd daemon is used, issue the following command:

```
$ sudo systemctl stop application
```

The application will automatically remove the limits, detach all eBPF programs and delete the build artifacts.

Chapter 3

Developer Documentation

3.1 Specification

3.1.1 The problem

The core problem is rather simple: there are no tools available to simulate network bottlenecks, faults, or anything of the sort in servive mesh environments. Though some tools do implement a set of predefined rules that can be applied[1], these are usually very limited.

More elaborate limit rules can be applied and more tools are available if we drop the constraint of being Kubernetes-specific. On machine level, a whole lot of options are available, including CLI tools, daemon services, etc, but these do have disadvantages over Kubernetes-related, eBPF solutions.

- Slower: eBPF enables very fast packet handling because it does not require packets to be moved into user space. Most tools today do not use eBPF due to its newness.

- Machine- or application level limits only: Kubernetes constructs such as Pods or Services can not be referenced

[1]For example, Cilium has a bandwidth manager that used EDT to limit the bandwidth of egress packets. https://docs.cilium.io/en/v1.9/gettingstarted/bandwidth-manager/

- Hard to modify: these tools are often old and when they do not suit the usage, the necessary modification is costly - if at all possible.

These disadvantages warrant a solution that is both fast and versatile - it needs to have sane defaults while also providing extensibility.

3.1.2 Architecture

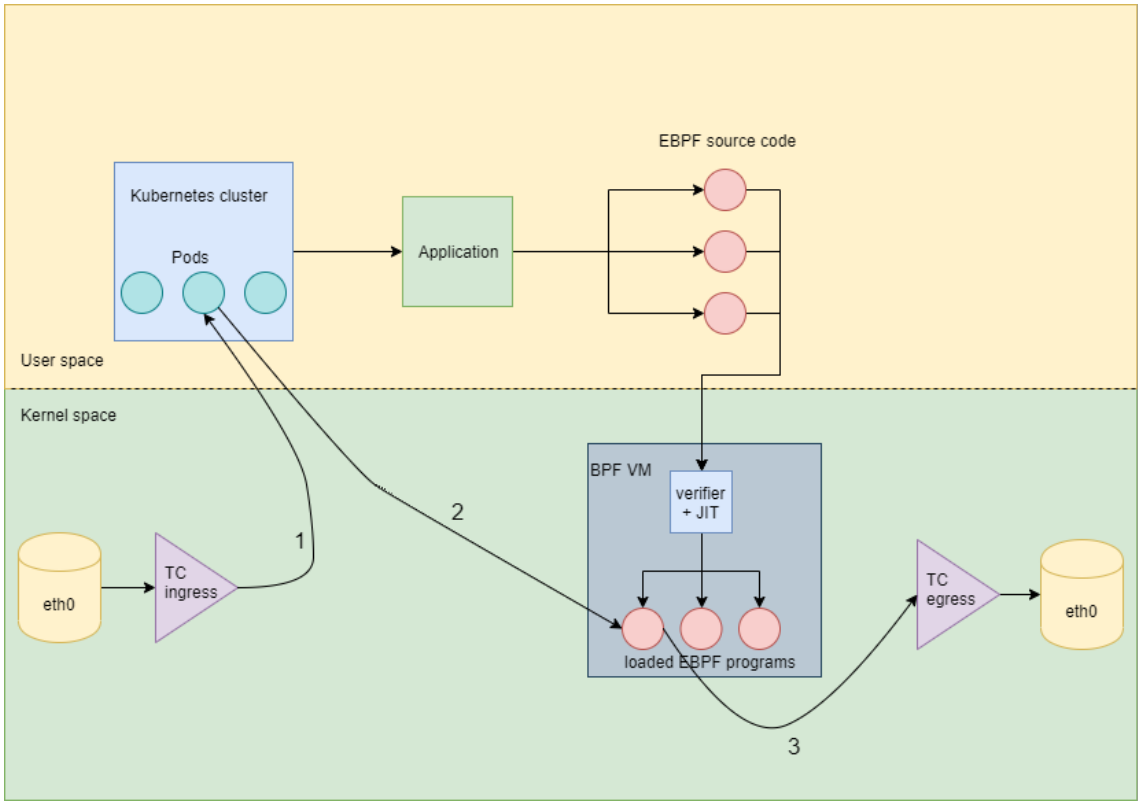

Figure 3.1: Application architecture

The proposed solution consists of three main parts.

1. The Kubernetes cluster: in most cases, this is a preexisting system, unless specifically set up for testing the application

2. The application: continuously monitors the state of the Kubernetes cluster and applies limits based on its configuration

3. The eBPF Virtual Machine: loads the eBPF programs, verifies their code and runs them whenever a packet arrives on the target ingress / egress interface.

These main parts and technologies, with the addition of some glue, play together to form a versatile yet extensible ecosystem and solve the proposed problem.

3.2 Underlying technologies

3.2.1 Kubernetes

Kubernetes, or K8S for short, is an open source container orchestration system that is simply too vast to properly explain in a BSc thesis. It supports automatic deployment and scaling of containerized workloads and services. It also works as a base for an ecosystem that is built around it. The project was open-sourced by Google [16] in 2014 and gained prominence in the last few years. Today, it is a mature platform to depend on - no matter what kind of service one is running. Chances are, if scale is a concern, Kubernetes is the answer.

Why use Kubernetes?

Today, users of online services are used to a high standard - fast response times, no downtime, no data loss, easy usage. This standard requires modern solutions; a single machine in one's garage running a LAMP [17] (Linux, Apache, MySQL, PHP) stack will not cut it anymore. In fact, oftentimes a garage full of server equipment will not cut it.

Scaling this big is hard, though, and This is what Kubernetes is trying to mitigate. Using containers, software becomes more manageable. Containers provide an unified system services can run on, without having to care about what kind of system they run on. These can then be started on-demand, for example when there is a load spike, or another container dies. This ensures that the available workforce is always present and can endure the load, no matter what size it is.
The main benefits of Kubernetes are:

- Load balancing: Kubernetes can ensure that no host gets overloaded and the services remain stable
- Scalability: Containers can be started and stopped on-demand. The deployments always fit to the load.
- Automation: Kubernetes automatically modifies its state at a given rate to meet the requirements set by the sysadmins.
- Self-healing: Dead or non-responding pods are automatically restarted.

Kubernetes architecture

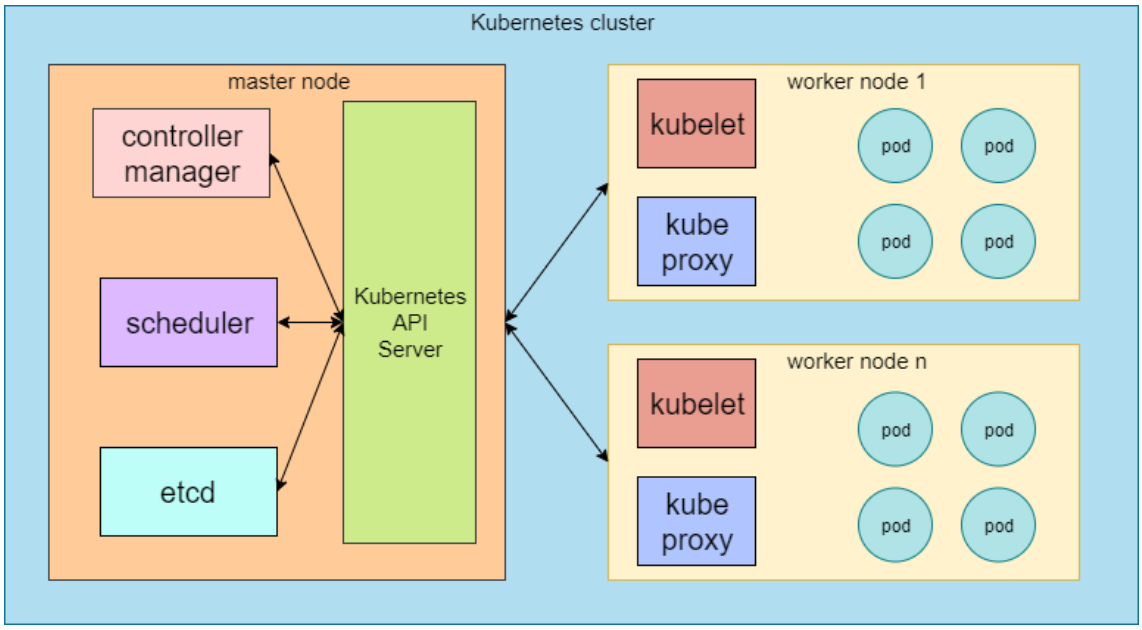

Figure 3.2: Basic architecture of Kubernetes internals

A Kubernetes cluster is made up from two primary components: the control plane (also called the master node), and one or more worker nodes. The former contains the logic required to orchestrate the latter.

Master node

The master node has a single instance and combines multiple already existing - and proven by time - technologies that make Kubernetes so flexible.

The core of everything is the kube-apiserver. It provides an API to configure and monitor everything while also communicating with other parts of the cluster, such as etcd [18], the kube-scheduler, and controller manager. Etcd is used as a reliable key-value storage that holds every piece of data related to the cluster. The kube-scheduler is responsible for assigning worker nodes to Pods that need to be started. Finally, the controller-manager changes the state of the cluster to always fit the load best.

The master node can run on any machine in a cluster, but typically it is either on the same machine as the worker nodes (in on-premises cases), or it is hosted by a cloud provider.

Worker node

The worker nodes - their number can vary - are the workhorse of the cluster. They are used to run Pods that run the target applications - be they online services or ML learning algorithms.

The worker nodes are controlled by a program called kubelet. Kubelet manages the Pods and the CRI, the container runtime, which is generally, but not limited to Docker [19] or containerd [20].

Another tool called kube-proxy implements the overlay network used by Kubernetes. This is required because the communication between pods should be private; unauthorized access from the outside is unwanted. Similarly, the Pods should not access resources outside the cluster that are not explicitly configured.

Concepts

Pods are the smallest components used in Kubernetes. As the name suggests, they represent a group of containers that are closely related and should be handled together. Pods are fast to start and shut down by design, as they are often rotated for various reasons, examples being scaling up and down, starting new pods to replace dead ones, etc.

Services are used as an abstraction over a group of Pods. Since usually there are more than one pods, something has to act as a load balancer to distribute the requests between the running pods. They are also used to expose an application to the (inner) network.

Ingresses act as a reverse-proxy and provide an unified channel external requests can enter through. Depending on the specific implementation (for example, NGINX [21] and Traefik [12] are solid ones, this thesis project uses the latter), they provide various features, the most important ones being dns/port based routing to services, load balancing and SSL termination.

These aforementioned, rather simple concepts enable the sysadmins to build complex, yet highly performant clusters that scale well under even the most extreme

loads.

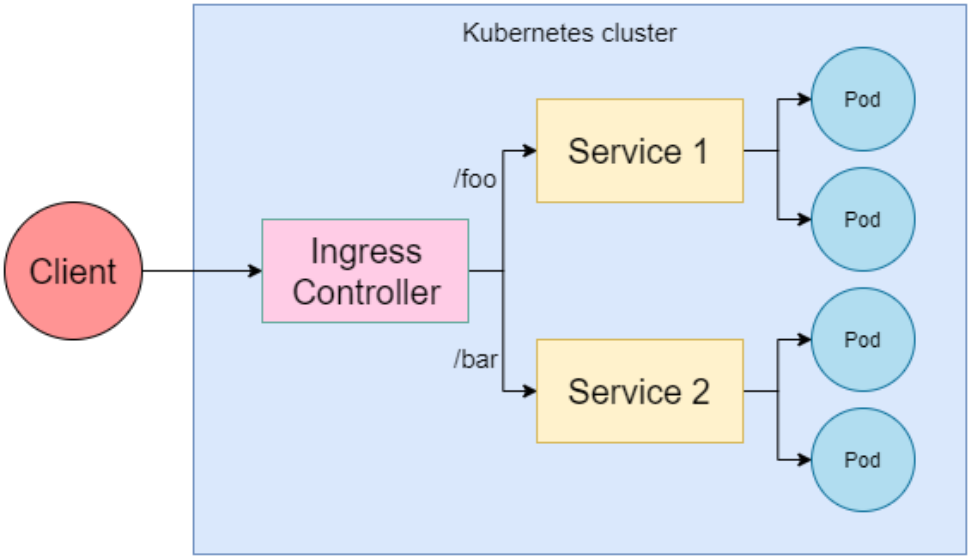

Figure 3.3: A simple Kubernetes cluster configured with two microservices

In the figure above, an ingress controller is used to let in traffic on the HTTP and HTTPS ports, routing them to two services that abstract the pods - 2 each - running web microservices. This simple example shows how the Kubernetes objects interact and enable operators to build bloat-less systems.

3.2.2 eBPF

eBPF [22] emerged from BPF [23], which is a complex system that leverages a virtual machine with a limited instruction set to run user-made programs in the kernel space. This avoids the need to copy each network packet to the user space, granting significant speed gains. These BPF programs interact directly with raw network sockets.

Before loading the programs into the kernel, BPF analyzes the source files to make sure they contain only permitted instructions. The instructions in these files are then run sequentally. The execution of the programs is event-driven; they can be attached to various events, such as the arrival or transmission of a network packet.

BPF VM

BPF uses a virtual machine that runs code made up from a set of instructions conforming to the format described in Code 3.1.

```
struct bpf_insn {
  __u16 code; // Opcode
  __u8  jt;   // Jump if true
  __u8  jf;   // Jump if false
  __u32 k;    // Instruction-dependent field
};
```

Code 3.1: BPF instruction format

The programs are made up from these instructions and are represented as an array of 4-tuple, run in a sequence.

```
struct bpf_program {
  unsigned short len; // Number of instructions
  struct sock_filter __user *filter;
};
```

Code 3.2: BPF program format

The BPF virtual machine has a rather simple architecture. A 32 bit wide register - called A - is used as an accumulator and another one - called X - is used as a helper

register. The VM also has a 32 bit wide "scratch memory store" - called M - that can store up to 16 values.

BPF has a simple RISC (Reduced Instruction Set Computing) instruction set consisting of load, save, jump, arithmetic and logic operations. A program of these instructions is usually denoted as an array of 4-tuples.

Full list of instructions:

Instruction	Addressing mode	Description
ld	1, 2, 3, 4, 12	Load word into A
ldi	4	Load word into A
ldh	1, 2	Load half-word into A
ldb	1, 2	Load byte into A
ldx	3, 4, 5, 12	Load word into X
ldxi	4	Load word into X
ldxb	5	Load byte into X
st	3	Store A into M[]
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <x>
jneq	9, 10	Jump on A != <x>
jne	9, 10	Jump on A != <x>
jlt	9, 10	Jump on A < <x>
jle	9, 10	Jump on A <= <x>
jgt	7, 8, 9, 10	Jump on A > <x>
jge	7, 8, 9, 10	Jump on A >= <x>
jset	7, 8, 9, 10	Jump on A & <x>

Table 3.1: BPF instructions part 1

Continued:

Instruction	Addressing mode	Description
add	0, 4	A + <x>
sub	0, 4	A - <x>
mul	0, 4	A * <x>
div	0, 4	A / <x>
mod	0, 4	A % <x>
neg		!A
and	0, 4	A & <x>
or	0, 4	A | <x>
xor	0, 4	A ^<x>
lsh	0, 4	A « <x>
rsh	0, 4	A » <x>
tax		Copy A into X
txa		Copy X into A
ret	4, 11	Return

Table 3.2: BPF instructions part 2

There are multiple addressing modes each having its own quirks. The table below explains their behavior.

Addressing mode	Syntax	Description
0	x/%x	Register X
1	[k]	BHW at byte offset k in the packet
2	[x + k]	BHW at the offset X + k in the packet
3	M[k]	Word at offset k in M[]
4	#k	Literal value stored in k
5	4*([k]&0xf)	Lower nibble * 4 at byte offset k in the packet
6	L	Jump label L
7	#k,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
8	x/%x,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
9	#k,Lt	Jump to Lt if predicate is true
10	x/%x,Lt	Jump to Lt if predicate is true
11	a/%a	Accumulator A
12	extension	BPF extension

Table 3.3: Addressing modes of instructions

eBPF

BPF was originally proposed in 1992, having seen low to moderate success. eBPF, which builds on the original BPF implementation, first appeared in Linux kernel 3.18 [24] released in 2014 and became widely adopted in the last few years. However, it is still somewhat of an underground technology, as it requires more-than-average knowledge about the Linux kernel and its use cases are quite narrow.

The relative bigger success of eBPF is mainly due to the fact that it not just extended, but reworked the technology from ground up. The VM was expanded to support 64 bit architectures and a variety of things absent form the predecessor.

eBPF VM

The re-implemented eBPF VM architecture has eleven general-purpose 64 bit wide registers (r0 - r10) that can also be used in 32 bit mode. R10 is read-only and contains the FP It also has a PC and a 512 byte long stack.

This new architecture also brought a new instruction format along described in Code 3.3.

```
struct ebpf_insn {
  __u8  code;   // Opcode
  __u8  dst_reg:4;  // Destination register
  __u8  src_reg:4;  // Source register
  __s16 off;    // Signed offset
  __s32 imm;    // Signed immediate constant
};
```

Code 3.3: eBPF instruction format

The new instruction format made it necessary to update the instructions themselves as well. The base stayed the same, but it was extended to support 64 bit wide data and added several new features, for example, function calls and proper branching instructions. The list is too long to include, however it is freely available [25].

This new architecture is much more powerful than the limited functionality the original BPF system provided. This introduced the requirement of having a verifier that scans the code and rejects to load anything outside the allowed parameters. eBPF has three main rules that make it safe to use.

- Only forward jumps are allowed. This ultimately means that loops are forbidden, guaranteeing the termination of the program.[2]
- All memory accesses are validated. The kernel must not allow the programs to tamper around in the memory.
- The program must have no more than 4096 instructions.[3]

These basic rules together ensure that all BPF programs run fast and terminate without exceptions.

eBPF for packet filtering

Even though BPF's original purpose was - as the name suggests - packet filtering, eBPF broadened the tool's functionality. Nowadays, a hook can be virtually added anywhere on the machine, be it in the kernel or user space code. These can be done using kprobes and uprobes respectively.

Staying on the topic, packet filtering can be done through attaching eBPF programs to ingress or egress socket buffers. Ingress socket buffers are where transmitted data is passed to a process, and egress socket buffers are where the process outputs data that gets transmitted back onto the network. These socket buffers are located in kernel space and are managed by the Linux TC.

An example of a simple bandwidth limiter eBPF program would be attached to the egress of a process. It would either let packets pass, or throw them away based on their length and the time elapsed since the last packet. More sophisticated solutions would delay the packets and not throw them away. This latter is called EDT filtering.

[2]Since Kernel 5.3, bounded loops are allowed. See https://lwn.net/Articles/794934/.

[3]Since Kernel 5.2, the limit was pushed out to one million. See https://www.spinics.net/lists/netdev/msg561449.html.

3.2.3 Go

Golang [11] is an open-source programming language that focuses on code simplicity, efficiency and reliability. It has a very opinionated view on how code should look and behave, which is referred to as idiomatic way. Go also has a wide set of concurrency mechanisms which make it perfect for writing networking and multi-core applications.

The language was designed at Google [16] by Rob Pike [26], and Ken Thompson [27] themselves, among others, first released in 2012 [28].

Under the hood, Go is a compiled and statically typed language that resembles C [29], but unlike this latter, it has a garbage collector, it is memory-safe, and has an actually usable standard library.

The project uses Go as the main language for the application because there were no special requirements, any other general-purpose languages would have worked, for example C++ or Python. Go's preference of simplicity and my familiarity with the language were the main reasons to pick it.

One important thing about Go is that it does not support most classic OOP paradigms such as inheritance or polymorphism. Instead, Go has an unique take on what OOP should be like. This by the way, often throws off developers getting acquainted with the language as it feels alien at first. For this same reason, the thesis does not include any complex OOP features - they would be pointless anyway as the internal structure is kept simple.

3.2.4 Other Linux technologies

Control Group V2

Another relatively recent technology utilized by the thesis project is cgroup v2 [30]. Cgroup is a hierarchical system where groups can contain other groups and every process belongs to exactly one group. These groups are addressable and are used to distribute system resources, for example CPU capacity or usable RAM.

The old V1 version is now deprecated and is superseded by V2. It unifies the then scattered layout of the hierarchy in one tree. Though it is widely available, some projects still do not fully support it, one example being K3S (a K8S distribution, also used by this project) until very recently.

The thesis project uses the cgroup V2 system to address processes during the eBPF attachment procedure.

Tools

Bpftool [31] is a Linux utility that is used to load and unload BPF code as well as to inspect BPF logs and maps. This latter can be done by running the following command.

```
$ sudo bpftool p t
```

Clang [8], part of the LLVM project is a C/C++ compiler that is able to compile eBPF bytecode. At the time of writing this thesis, this compiler is the most reliable way to generate eBPF programs.

3.3 Implementation

3.3.1 The application

This section will explain the actual implementation of the application and its subprograms. The application is written in Go because of its simplicity and robustness. As the main application is just orchestrating other technologies to work together, no complex structure was required.

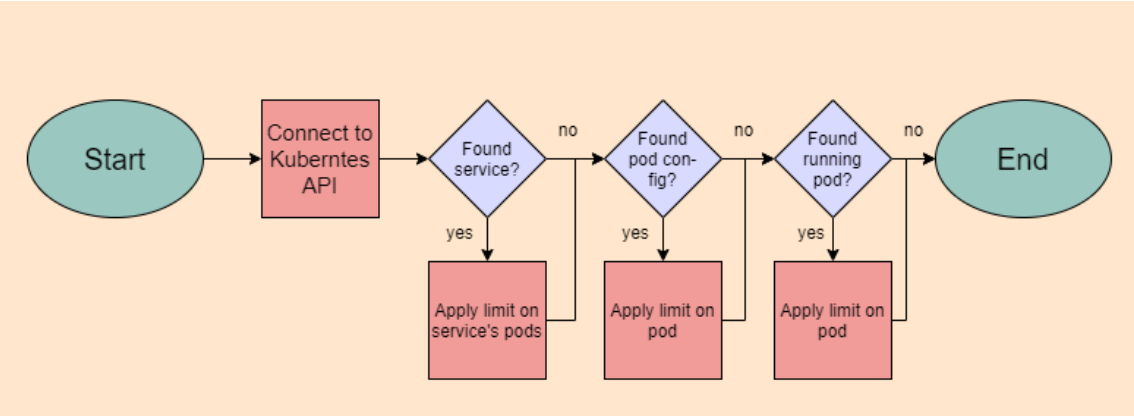

Figure 3.4: Application logic flow

The structure of application follows a simple workflow. On startup, the application connects to the Kubernetes API, sweeping the cluster for services and pods that require eBPF programs to get hooked on. It also starts to watch events happening in the cluster, in case pods get replaced.

When finding a service that is configured to use a limit, the corresponding pods are also searched and the limit is applied on them. When finding a direct pod limit, it is directly applied.

The process of applying eBPF programs that implement limits consists of compiling the eBPF programs with the given parameters, loading them into the kernel, and attaching these programs to the cgroup of the given pod's processes.

Startup

In Go programs, the entry point is a `main()` function in the package main. This is where the program flow starts.

When starting the application, the first thing that happens is locating the kubeconfig file. This file can be either in the home folder of the user, or anywhere else - this location can be passed to the application through a command-line parameter. After locating the kubeconfig file, the contents are interpreted and the application attempts to connect to the Kubernetes API.

A struct - the Go equivalent of class - called PodManager is responsible for the business logic.

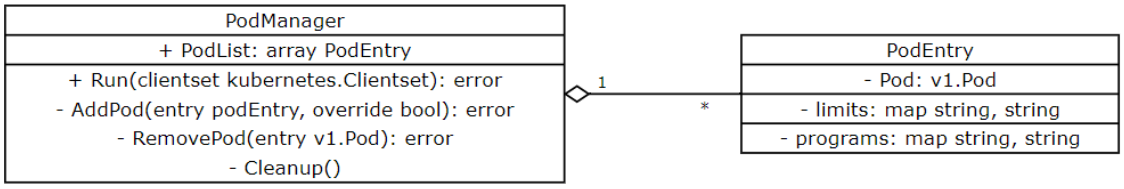

Figure 3.5: Class diagram of the pod manager code section

The PodManager holds all the data related to the program execution - information about pods and their corresponding limits and eBPF programs - and also contains the core logic. The execution start when the `Run()` method is called. This method executes the following 3 steps in a strict order:

1. Looking for configured services: most often, limits will be applied on this level.
2. Looking for configured pods and deployments: this must come after looking for services; pod-level limits overwrite service-level ones, providing a hierarchical solution.
3. Setting up an even watcher for pods through the Kubernetes API: this step comes last because at this point, we only want to find changes in the cluster.

When looking for services and pods, the software is checking their annotation parameter to see if there are any limits present. In the last step, the event watcher is set up and it runs until the application exits. This enables dynamic loading and unloading of eBPF programs whenever the cluster is changed.

eBPF program loading

The first step of eBPF program loading is to actually compile the object code. This is done using the Clang compiler. The pre-implemented programs' sources are stored in the `bpf` folder. Every program is compiled independently to provide custom settings through the macros.

During compilation, the kernel headers are linked to the object code, and specific macros are set depending on which program is being compiled. These macros are used to bypass a limitation of the eBPF VM, namely that it does not allow the programs to be parametrized. Although eBPF does support maps that are accessible from user space, these do not fit the use case.

After compilation, the next step is to actually load the object-code into the kernel. This is done through a system call, which is done by the bpftool utility automatically invoked though command line. Upon loading the object-code, the eBPF verifier examines the code and makes sure it is safe to run. If the verifier accepts the code, it gets compiled to machine code. The internal compiler also optimizes the code, making it run at speeds similar to natively compiled kernel code.

Once the eBPF program is loaded into the kernel, it must be attached to a process. This process is referred to by its cgroup, which is found in the `/sys/fs/cgroup/` folder. After the attachment, the system is live and the limits should instantly begin to take effect. This can be made sure using the following command.

```
$ sudo bpftool p t
```

If the process finished successfully, the command should show packet transmission and whether they were dropped or not.

Cleanup

On exiting (and pod removal in the cluster), cleanup is done. This includes removing temporary artifacts as well as unloading and deattaching eBPF programs - even though they automatically get cleaned up if the process they are attached to stop. This process is done for the sake of completeness.

These processes happen in reverse order - meaning first the program is deattached from the cgroups, after which they are unloaded, and finally, the artifacts (the object files) are deleted.

3.3.2 Implemented rules

Although the thesis project is extensible with custom user-made eBPF programs, it comes with prebuilt ones that cover the most frequently required features. The following sections describe the inner working of these programs.

Bandwidth limit

The bandwidth manager's sole purpose is to limit the bandwidth used by a specific Pod's ingress or egress. To reach this goal, it employs an algorithm called TBF which drops packets that are too big compared to the time elapsed since the last one. This is a rather simple solution, keeping the average bandwidth below a specified threshold.

The pros of TBF include simplicity and high processing speed. This can be useful in cases where low latency is an absolute necessity, for example serving multimedia content on the web. The high speed of the algorithm is due to its simplicity; the low latency is guaranteed by the fact that packets are either instantly let-through, or dropped. Below-limit bandwidth is always instantly forwarded.

The cons arise from the nature of the algorithm - packets above the bandwidth limit get dropped. This means that only TCP transmissions are reliable, as they resend packets that were not acknowledged. UDP does not do this, requiring proper user-made netcode to handle dropped packets.

Staying at the topic of transport layer protocols, TCP also shapes the speed of the data transmission dynamically. This is helpful because as packets get dropped, TCP throttles the speed, requiring the bandwidth manager to drop less packets. UDP does not have any similar features.

The TBF algorithm uses a bucket that gets filled with "tokens". Packets take tokens to send, and if a packet would completely drain the bucket, it gets dropped, otherwise let through. More formally:

1. Tokens are added to a bucket with 1/ rate interval.
2. A bucket can hold at most b tokens.
3. When a packet arrives (of length p):
   (a) If the bucket has more than p tokens, the same amount is removed and the packet is let through.
   (b) Otherwise, the packet is discarded.

eBPF is event-driven - in this case, these events are packet arrivals on either the ingress or the egress interface - therefore this algorithm requires slight modifications to work properly, namely the token addition is the part that requires discrete time steps instead of being continuous. The implemented algorithm is the following.

1. When a packet arrives:
2. Calculate the time between the last packet and the current packet's arrival as dT.
3. Add dT * rate amount of tokens to the bucket
4. If the bucket has more than p tokens, the same amount is removed and the packet is let through.
5. Otherwise, the packet is discarded.
6. Update the time of the last packet's arrival

The algorithms may slightly differ, but in essence, both implement the same idea: only let through packets through where the average bandwidth calculated as `packet size * (time of last packet - time of current packet)` is below the threshold. This limits the maximal available bandwidth.

There are also other TBF algorithms that extend the basic idea of it, though they are not implemented in the project as they are not required for the original purpose. Examples include the leaky bucket and the hierarchical token bucket algorithms.

Packet loss

Simulating packet loss is useful in cases where network-congestion testing is important. It can be configured for Pods' ingress or egress interfaces. There are two variations provided by the implementation, one using uniform and the other using exponential distribution.

Simulating probability-based things is very difficult without floating-point numbers or an usable standard library. Unfortunately, neither of those are available in the Linux kernel. To further complicate things, a PRNG is also required, not provided by the kernel by default.

For random-numbers, both implementations use the function `bpf_get_prandum_u32()`. This generates a random 32 bit value which is then used as a base for both distributions. The absence of floating-point numbers is circumvented by only using integer calculations; this does lose precision but for the use case, it is good enough.

Using the **uniform-distribution**, a percentage of packets can be dropped. This comes useful simulating congestion that happens on a regular basis, such as crowded Wi-Fi. This version of the loss manager is pretty straightforward, requiring little explanation. The random 32 bit number is interpreted as an unsigned integer modulo 100, and the output is compared to the given percentage provided in configuration. The packet is then passed through or dropped based on this.

The **exponential-distribution** has a threshold above which it throws packets. Using this distribution, edge-cases can be simulated - as the top-end of the distribution is rather rare.

Exponential distribution can be calculated from uniform one. Given a variable U conforming to uniform-distribution between [0,1), the following formula can be used to calculate the exponential-distributed value:

$$\xi = -\frac{1}{\lambda}\ln(1-U)$$

The implementation is somewhat awkward because of the absence of floating point integers and math libraries, this equation can not be used. Instead, since no better tools are available inside the kernel, a lookup table containing a predefined exponential curve is used. This is somewhat of a hack-ish solution, but after careful consideration, it was chosen not only because of the lack of better methods, but also for its speed. Very few - if any - other solutions beat the speed of lookup tables.

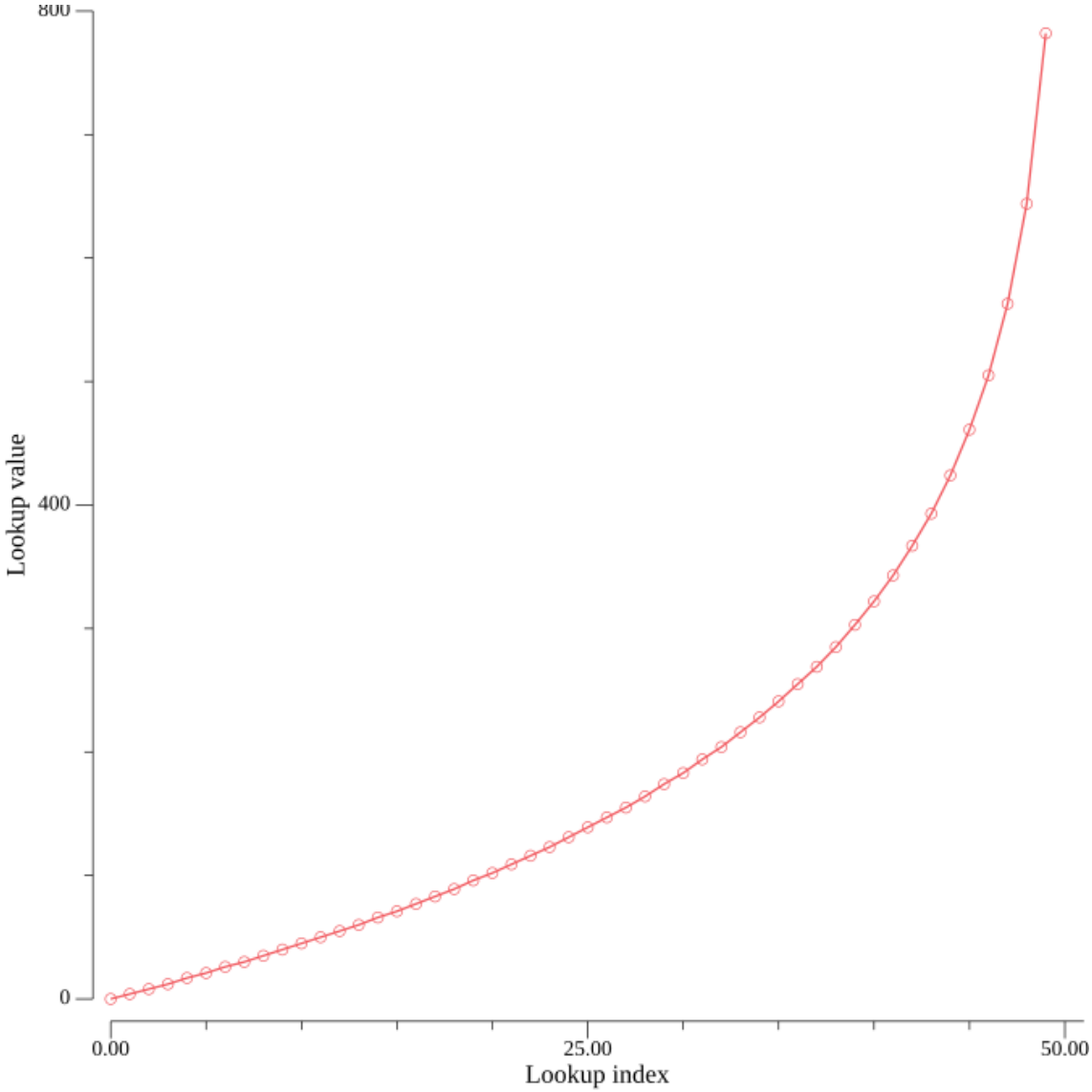

Figure 3.6: Lookup table for exponential distribution

The lookup table itself is implemented with a switch statement, instead of using an array. This is due to the eBPF verifier, which will not let the program offset an array with a random value - not even in a sanitized and bound-checked way.

The exponential distribution code is straightforward as well. Given the random 32 bit unsigned integer modulo 50 (this number was chosen for keeping the size of the lookup table manageable), the corresponding exponential value can be found. This is then compared to a threshold, and the packet is passed through or dropped based on the comparison.

EDT

Another way of bandwidth limiting is the EDT algorithm. Instead of dropping, this algorithm delays packets to ensure the fixed bandwidth threshold is not exceeded. Due to its nature, this algorithm only works for egress interfaces, and it is not fit for our use case.

Nevertheless, it is implemented as it may come useful later in the future. It can be found in the `bpf/edt.c` file.

The algorithm uses a variable called `t_next` that is set in the previous packet transmission. This variable is used to determine the earliest point the next packet can leave. The delay of a packet is calculated as `packet_size * bitrate`. If the packet is sent after `t_next`, it is passed through and `t_next` is set to the current time plus the delay.

If the packet is before `t_next`, the delay is added to the packet's timestamp and it is passed through - meaning it will be resent at a later date.

3.4 Testing

The correctness of the software bundle is ensured with both tests and benchmarks. The tests are performed on the code, while benchmarks are measured on the running systems.

3.4.1 Unit tests

Since the main application is kept simple, meaning it has neither complex structure or user-interface, testing the code base is simple as well, as there are only a few things that can be tested.

Due to how Go implements the type system, mocking is rather verbose. Polymorphism is achieved through interfaces, which are types having only methods. Actual types can implement these, and therefore the interface itself, and can be passed as such. This means mocking for example the Kubernetes API would mean all of its functions need to be re-implemented, which is simply not feasible. For this reason, dependencies such as the Kubernetes API are not tested; they should be tested in a stand-alone manner anyway.

The following basic features are tested:

- Compiling programs
- Deleting programs
- Loading programs
- Unloading programs

The eBPF programs themselves cant be tested in any way due to their specific format and the fact that tests can not be run in the Linux kernel.

3.4.2 Benchmarks

The benchmarks are run on both pre-written managers to ensure proper functionality.

Bandwidth manager

The bandwidth manager limits the maximum usable bandwidth for a given pod, therefore the sustained bandwidth is measured to ensure these metrics remain below threshold. For measurement, iperf3 [32] was used in TCP mode, the tests were run for one minute for each limit. The bandwidth limiter was attached to the egress interface of the sending process.

The findings can be found in Figure 3.7. As it can be seen, the bandwidth conformed to the limits pretty closely, with errors here and there. The maximum measured error was 4.7%, which occurred in the 600 Mbps run.

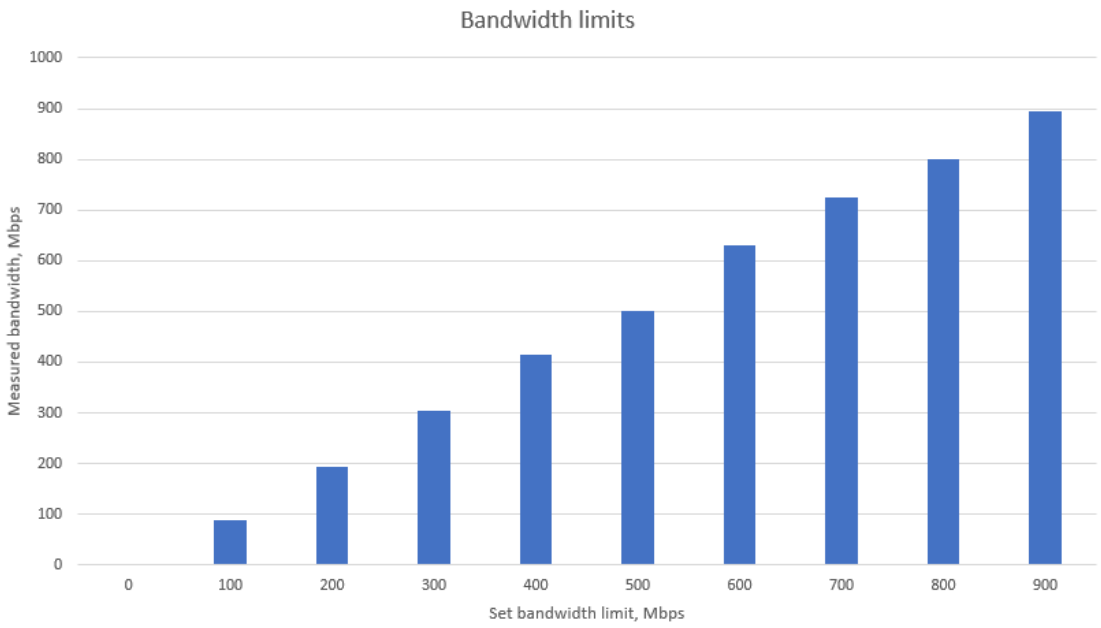

Figure 3.7: Bandwidth benchmarks

Loss manager

The loss manager has two different mechanisms to calculate packet drop chances. The common thing between these two is that both are using random numbers in a way or another to decide which packet to drop. Both are measured using a small

utility that tries to send 100.000 UDP (this is important because TCP will try to re-send non-acknowledged packets) packets. Similarly to the bandwidth manager, the eBPF programs are attached to the egress interface to the sending process.

The first version of loss manager is using uniform distribution to drop a percentage of packets. The accuracy of the method can be seen in Figure 3.8. Since there is no time or packet size data in play here, and the random generator is reasonable reliable, the measured packets follow the targets very well.

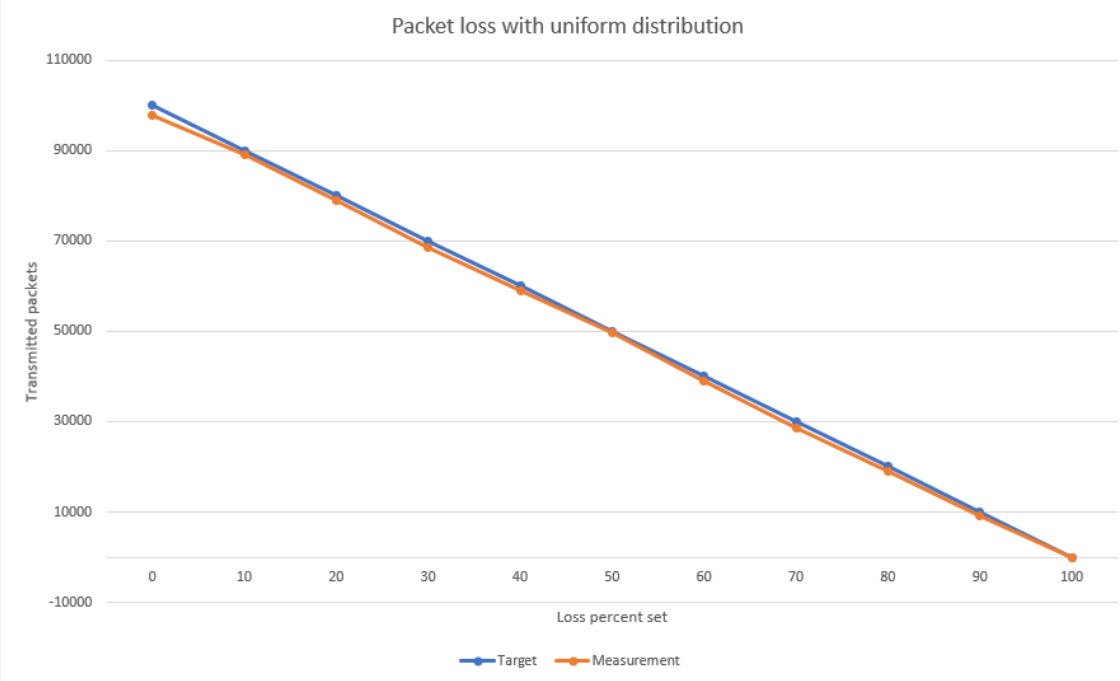

Figure 3.8: Loss benchmarks with uniform distribution

The second version uses exponential distribution - enabling the users to simulate events that occur in edge cases. The measurements can be seen in Figure 3.9. There is a slight deviation from the target in the middle of the plot, this is due to the usage of lookup tables instead of a continuous function - a little precision is sacrificed for speed. However, the measurements are still relatively close to the target values, making the exponential distributed packet loss an usable feature.

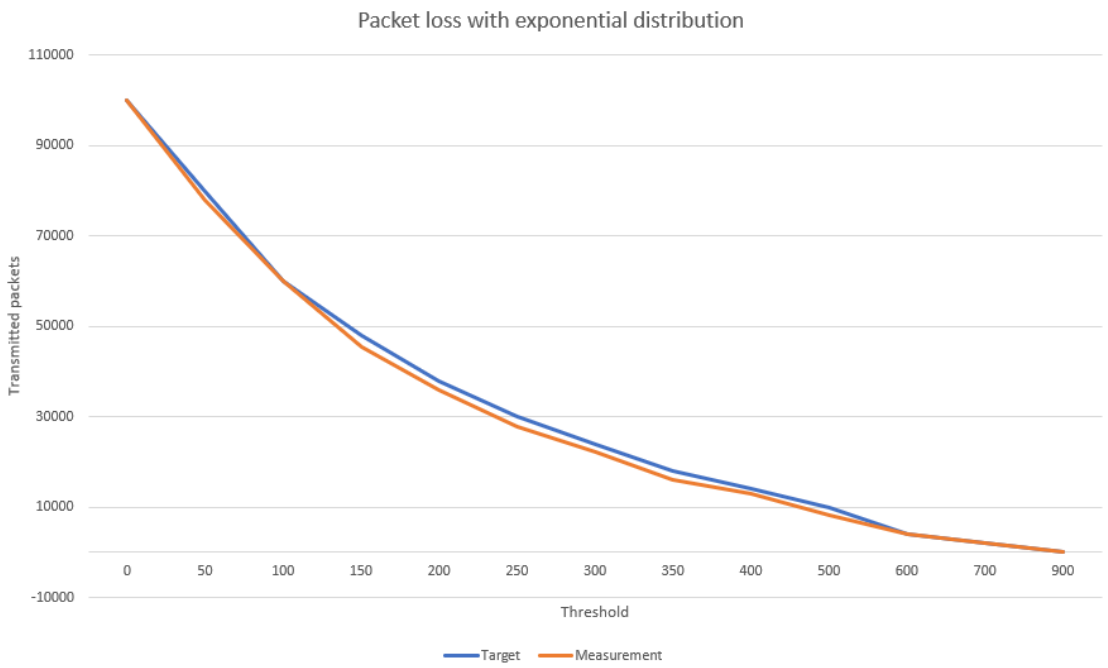

Figure 3.9: Loss benchmarks with exponential distribution

Chapter 4

Summary

The implemented application simulates various conditions on service or pod level by applying eBPF filters on the pods' ingress or egress socket buffers. This is done through the Kubernetes API.

Based on the benchmarks, the implementation is conforming to the proposed specifications, requiring no further fine-tuning. The application does its job - simulating various network conditions - and it is also extensible, having potential for more complex filters.

The application can be extended in various ways in the future. Two main possibilities are a graphical user interface and adding robustness (more portability, not requiring root privileges).

Glossary

API Acronym for Application Programming Interface, a system that lets two applications communicate. 25

compiled A type of programming language that requires the source code to be translated to machine-readable code before execution. 34

CPU Acronym for Central Processing Unit, referring to the processor of a machine. 35

CRI Acronym for Container Runtime Interface, an unified interface for container runtimes. 26

EDT Acronym for Earliest Departure Time, an algorithm to limit the bandwidth of a given data stream by delaying packets. 33

egress A network interface where data exits a machine or process and gets transmitted. 40

floating-point numbers A rational number stored as exponent and mantissa. 42

FP Acronym for frame pointer. A highly specialized type of CPU register used for managing function calls. 32

garbage collector A programming language feature that automatically cleans up unused variables on the heap. 34

IDE Acronym for Integrated Development Environment, software with built-in features to support development. 14

ingress An network interface where transmitted data enters a machine or process. 40

kprobe Hooks that allow to run eBPF code when specific kernel functions are called. 33

latency The delay between two hosts on a network. 40

load balancer Distributes requests between given machines. This is used to scale the capacity of a system. 26

memory-safe A type of language with safety measures that prevent various memory-related errors such as buffer overflows, out-of-bounds writes and the like. 34

ML Acronym for machine learning, which is a branch of computer science. 26

mock A program object that pretends to behave like another one but has bare-bones implementation. Used for testing. 45

netcode A part of an application that deals with the data exchange over a network. It has to ensure data is reliably and correctly transmitted. 40

on-premises A setup where software runs on the own machines of a user or organization, instead of cloud providers. 7, 25

OOP Acronym for Object Oriented Programming, a popular paradigm built around the concept of data hiding - also called encapsulation. 34

PC Acronym for program counter. A specialized CPU register that points to the next instruction to be executed. 32

PRNG Acronym for pseudo random number generator, an algorithm that outputs numbers that are seemingly - but not really random. The randomness depends on the seed used to initialzie the generator. 42

RAM Acronym for Random Access Memory, used to store all temporary data in a machine. 35

read-only A register or memory location that can't be written to, only read from. 32

RISC Acronym for Reduced Instruction Set Computing which refers to an instruction set architecture that uses relatively few instructions. 29

socket buffer A structure to hold data from a packet. It enables easy access and modification from code. 33

SSL Acronym for Secure Sockets Layer, or more properly, TLS (Transport Layer Security). A set of technologies allowing channels to be encrypted, preventing man-in-the-middle attacks. 26

stack A general-purpose register that holds information about the call stack, which describes subroutine call order.. 32

statically typed A type of language where the type of a variable is known at compile-time. 34

TBF Acronym for Token Bucket Filter, which is an algorithm to limit bandwidth of a data stream. This is achieved by dropping non-conformant packets. 40

TC Acronym for Traffic Control, an user space program used for configuring the Linux packet scheduler. 33

TCP Acronym for Transmission Control Protocol, which is a reliable protocol for data transmission, but slower than UDP. 40

transport layer A conceptual layer in the network stack of the OSI model. Protocols in this layer deal with host-to-host communication and data multiplexing between processes. 40

tree A hierarchical data structure made up from nodes ordered in layers. Each node can have multiple children but only one parent. 35

tuple A finite ordered list. A tuple of n elements is called n-tuple. 28

UDP Acronym for User Datagram Protocol, an unreliable but fast protocol for data exchange. 40

uprobe Hooks that allow to run eBPF code when specific user space functions are called. 33

verifier The eBPF verified is used to inspect the code before loading it into the kernel, making sure it doesn't contain unsafe code that can harm the system. 32

List of Figures

Bibliography

[1] *Calico project, May 2021.* URL: https://www.projectcalico.org/.

[2] *Cilium project, May 2021.* URL: https://cilium.io/.

[3] *Flannel project, May 2021.* URL: https://github.com/flannel-io/flannel.

[4] *Weave Net project, May 2021.* URL: https://www.weave.works/docs/net/latest/overview/.

[5] *Linux kernel, May 2021.* URL: https://www.kernel.org/.

[6] *Kubernetes orchestration engine, May 2021.* URL: https://kubernetes.io/.

[7] *K3S, a lightweight Kubernetes distribution, May 2021.* URL: https://k3s.io/.

[8] *Clang - C Language Family Frontend for LLVM, May 2021.* URL: https://clang.llvm.org/.

[9] *BPF Developer Tools, May 2021.* URL: https://archlinux.org/packages/community/x86_64/bpf/.

[10] *LibBPF, a library for loading eBPF programs and manipulating eBPF objects, May 2021.* URL: https://github.com/libbpf/libbpf.

[11] *The Go Programming Language, May 2021.* URL: https://golang.org/.

[12] *Traefik edge-router, May 2021.* URL: https://traefik.io/.

[13] *systemd init system, May 2021.* URL: https://systemd.io/.

[14] *GoLand: A CLever IDE to Go, May 2021.* URL: https://www.jetbrains.com/go/.

[15] *Cilium BPF Reference Guide, May 2021.* URL: https://docs.cilium.io/en/latest/bpf/.

[16] *Google, May 2021.* URL: https://www.google.com/.

[17] *LAMP Software Bundle, May 2021.* URL: https://en.wikipedia.org/wiki/LAMP_(software_bundle).

[18] *etcd key-value storage, May 2021.* URL: https://etcd.io/.

[19] *Docker container ecosystem, May 2021.* URL: https://www.docker.com/.

[20] *containerd container runtime, May 2021.* URL: https://containerd.io/.

[21] *NGINX web and reverse-proxy server, May 2021.* URL: https://www.nginx.com/.

[22] *extended Berkeley Packet Filter, May 2021.* URL: https://ebpf.io/.

[23] *Berkeley Packet Filter, May 2021.* URL: https://www.kernel.org/doc/Documentation/networking/filter.txt.

[24] *Linux Kernel version 3.1.8, May 2021.* URL: https://lwn.net/Articles/474637/.

[25] *eBPF instruction set, May 2021.* URL: https://github.com/iovisor/bpf-docs/blob/master/eBPF.md.

[26] *Rob Pike, programmer, May 2021.* URL: https://en.wikipedia.org/wiki/Rob_Pike.

[27] *Ken Thompson, computer scientist, May 2021.* URL: https://en.wikipedia.org/wiki/Ken_Thompson.

[28] *Golang first release, 2012.* URL: https://golang.org/doc/devel/release.html#go1.

[29] *The C programming language, May 2021.* URL: https://en.wikipedia.org/wiki/C_(programming_language).

[30] *control group v2, May 2021.* URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[31] *BPFTool, an eBPF utility, May 2021.* URL: https://man.archlinux.org/man/bpftool.8.en.

[32] *iPerf - The ultimate speed test tool, May 2021.* URL: https://iperf.fr/.