POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Towards Hybrid Network-Oriented Workloads on Edge Data Centers

Supervisors

Prof. Fulvio RISSO

Ing. Federico PAROLA

Ing. Giuseppe OGNIBENE

Candidate

Francesco CAPPA

ACADEMIC-YEAR 2021-2022

Abstract

With the advent of Cloud Computing, general-purpose workloads have been transformed into a set of cooperating micro-services, meeting the rapid, frequent and reliable delivery of very large and complex applications. Telcos have been trying to apply the same approach for network-oriented workloads, keeping into account the complexity and the high requirements needed. Among the leading technologies in this direction, eBPF represents an interesting option. eBPF is an in-kernel framework that also allows to implement data-plane services but it poses some limitations that prevents its applicability in some cases, which can be found for example in the Rate Limiter network function, which should be always present if policies are to be applied on the traffic. This thesis studies the possibilities of designing and implementing such a network function by using two technologies which could run simultaneously on a Linux machine: eBPF and AF_XDP. A hybrid prototype is proposed, highlighting the advantages and challenges coming from this integration. Finally, performance are compared with traditional Traffic Control in Linux at first and then by changing the amount of traffic handled by the two technologies. Results show that AF_XDP, besides overcoming limits coming from eBPF, could even achieve higher performance thanks to its capability of bypassing the Linux kernel. This might be an interesting solution to be considered when running virtual network functions at edge data centers, where both the flexibility and scalability requirements should be fulfilled and performance should be kept reasonably high.

Table of Contents

List of Figures

Acronyms

VM
Virtual Machine

NFV
Network Function Virtualization

LXC
Linux Container

CNF
Cloud native Network Function

NIC
Network Interface Card

TCP
Trasmission Control Protocol

IP
Internet Protocol

eBPF
extended Berkeley Packet Filter

XDP
eXpress Data Path

TC
Trasmission Control

AF_XDP

Address Family eXpress Data Path

POSIX

Portable Operating System Interface for Unix

OS

Operating System

JIT

Just-In-Time

RX

Reception

TX

Trasmission

COTS

Commercial Off-the-Shelf

API

Application Programming Interface

DPDK

Data Plane Development Kit

CNI

Container Network Interface

RSS

Receive Side Scaling

Chapter 1

Introduction

Traditionally, network functions such as routers, switches, firewalls and many others were provided as physical appliances, which connected people worldwide.

Due to its lack of scalability, flexibility, and programmability, this approach has been seen as discouraging: setting up all the needed infrastructure could be time consuming and expensive. Through the use of software services, a paradigm shift has taken place in the last few years.

First, there are *Virtual Machines (VMs)* which simplify and speed up the way in which those services can be provided, overcoming the aforementioned issues. Virtualized network functions have been deployed as VMs under the name *Network Function Virtualization (NFVs)*, with OpenFLow as one of the leading technologies. As a result, VNFs are not very flexible, since they would still require dedicated physical appliances, something that might not always be possible in data centers, where general-purpose servers are usually utilized for heterogeneous services.

Containers have been widely adopted to overcome these issues: lightweight virtual machines that can be deployed on a traditional Linux server, sharing the system resources and enabling multiple applications to run at the same time, without interfering with one another. There are several containerization technologies adopted. The most common one is *Docker* which comes from *Linux containers (LXC)*. When network functions are deployed as containers, they are called *Cloud-Native Network Functions (CNFs)*. In cloud environments, where availability, resilience, and robustness are highly demanded, the adoption of micro-services patterns, which are applications made up of independent services, has become very common, due to the fact that containers are applicable to general-purpose workloads. Following this pattern, each service can be handled independently and automatically by powerful container orchestration tools such as Kubernetes.

The idea of containerizing network functions is one of building robust service chains, made up of different functions, so they can be automatically handled by orchestration tools upon traffic loads.

However, networking tasks usually come with very strict requirements, such as taking exclusive access to the underlying system resources (e.g. NICs). Moreover, whenever an application runs on a server, sooner or later it would require to interact with the kernel of the system which costs the execution of system calls; those would trigger *context switches*, usually coming with performance degradation which could not be affordable in many cases (e.g. traffic handling).

A technology which already addressed those issues is *eBPF (extended Berkely Packet Filter)*, an in-kernel virtual machine which allows a programmer to push down in the kernel customized tasks at run-time. In order to do this, eBPF can execute programs on different levels of the kernel, where it could provide less or more features as well as be more or less performing, depending on which level is attached. This approach sounds promising when dealing with networking tasks since it could save system calls as they would only run inside the kernel, avoiding the communication overhead already mentioned. It also relies on services which are already present in a traditional Linux kernel. Furthermore, eBPF can attach different programs at different hook points, giving the possibility of building robust service chains which could address complex real-world scenarios.

On the other hand, eBPF is a very restricted environment that does not allow all possible operations to be performed. This is due to the presence of a verifier. This verifier analyzes the injected eBPF code in order to decide whether it could continue the execution or if it should be stopped because considered unsafe. Additionally, a slow down in performance could be experienced when relying on kernel-provided services, due to the allocation of resources which might not be useful in all cases. Those data structures are given by the *Linux network stack* which is composed of different layers, each one characterized by complex data structures which can be used to parse the traffic.

1.1 Goal of the Thesis

To overcome the eBPF limitations mentioned before, this thesis studies how to use *AF_XDP*, a new high-speed packet processing technology coming from the Linux kernel, to design and implement a rate limiter network function, which is in charge of applying policies to the upcoming traffic.

A comparison between AF_XDP and eBPF follows, exploring how it is possible to combine those two technologies in order to achieve a complete solution. A prototype is proposed and evaluated, highlighting all the limits posed by the technology, and the advantages and drawbacks of the solution. Then, an evaluation section will follow to analyse how the solution reacts under different scenarios.

Finally, an analysis of why this technology could be promising for CNFs is considered, giving suggestions for possible future directions.

Chapter 2

Background

This chapter both explores the evolution of Networking in Linux and describes the main key elements and the technologies which have been exploited to carry out the final result.

Firstly, an overall view about the progress and the changes that have been adopted in Linux Networking since its early days is given. Then, an introduction on the functionalities and features provided by the rate limiter is given, highlighting its importance in traffic management.

Lastly, an overview of related works focusing on the adoption of such solutions is proposed in order to better understand why they are promising and important for the upcoming network infrastructures.

2.1 Traditional Networking

Firstly, the deficiencies of the POSIX sockets API and their in-kernel implementations are described. Then, an analysis on kernel-bypass networking and programmable packet processing is given, including offloading to SmartNICs, to understand how the network stack is changing to meet the needs of contemporary hardware and workloads.

2.1.1 POSIX sockets

POSIX sockets are the standard programming interface for networking, adopted by most commodity operating systems from Linux to Windows. In the POSIX socket model, applications create a socket, which represents a flow, and use the file descriptor of the socket to send and receive data over the network. [1]

Most in-kernel network stacks implement POSIX socket operations as system

calls. That is, for both control plane operations and data plane operations, applications transfer control to the kernel using a system call. System calls are a problem for network-intensive applications because they have significant overheads. [2] The POSIX socket interface is also oblivious of multicore CPU and multi-queue NIC architecture. The application can access a socket on a different CPU that is managing the packet queue, which requires the OS to move packets between CPU cores. The socket API also pushes the OS to adopt a design, which demands dynamic memory allocation and locking. When a packet arrives on the NIC, the OS first wraps the packet in a buffer object, called a socket buffer (skb) in Linux and network memory buffer (mbuf) in FreeBSD. The allocation of the buffer object puts much stress on the OS dynamic memory allocator. Once allocated, the OS then passes the buffer object down the in-kernel network stack for further processing. The buffer object lives until the application consumes all the data it holds with the recvmsg() system call. As the buffer object can be forwarded between CPU cores and accessed from multiple threads, locks must be used to protect against concurrent access.

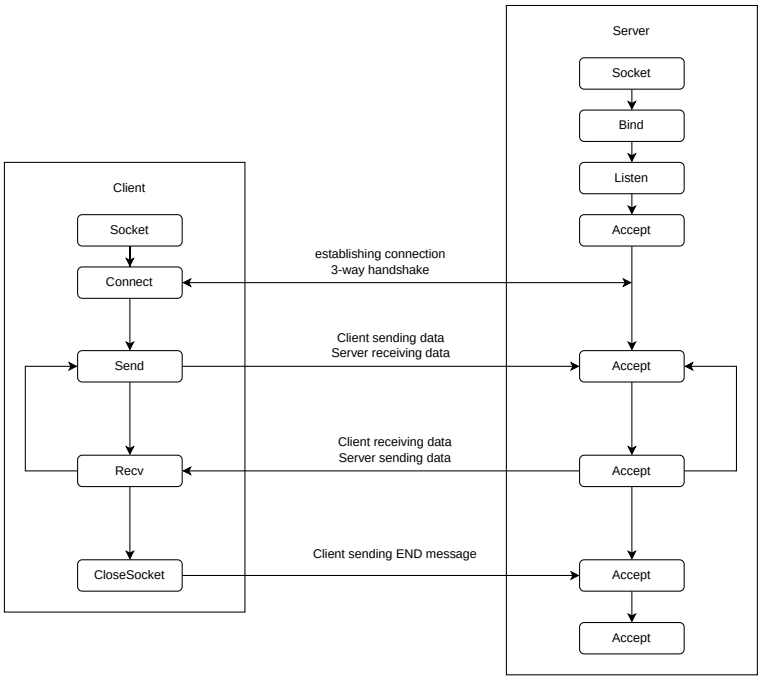

Figure 2.1: POSIX sockets.

2.1.2 Kernel-bypass networking

Kernel-bypass networking eliminates the overheads of in-kernel network stacks by moving protocol processing to userspace.

The packet I/O is either handled by the hardware, the OS, or by userspace, depending on the specific kernel-bypass architecture in use. For example, RDMA provides interfaces for directly accessing the memory of a remote machine, bypassing the OS for data plane operations altogether. In other words, an application receives messages in an RDMA-managed memory region without any inference from the OS. For Ethernet, the OS can dedicate the NIC to an application [3], which programs it from userspace, or the OS can continue to manage the NIC by allowing applications to map NIC queues to their address space [4]. Either way, packets flow from the NIC to userspace with minimal interference by the OS.

With the Operating System limiting itself to managing packet I/O, user-space applications are responsible for implementing the rest of the network stack. In practice, this means that user-space must at least implement the TCP/IP protocol suite and provide interfaces for applications to access messages carried over by the protocols.

Various userspace network stacks exist, but none of them have become a standard. Development and testing of the stacks are therefore fragmented, which limits their usefulness. Also, while it is possible to implement POSIX sockets API as a library, most userspace stacks provide their interfaces, which limits adoption and compatibility.

Programmable Packet Processing

Programmable packet processors are emerging as another technique to address the limitations of the in-kernel network stack. They allow execution of user-defined code either in the OS or the hardware. XDP is Linux's programmable packet processor. It allows a user-defined eBPF program to process a packet before it enters the in-kernel network stack. The eBPF program can either process the packet in full, perform some preprocessing and forward it to the in-kernel stack, or, with AF_XDP, forward the packet to userspace memory buffer after processing. Also, some SmartNICs, such as the Netronome Agilio CX, are capable of running eBPF programs directly on hardware. Offloads packet processing from the CPU to the NIC can reduce packet processing latency and improve energy-efficiency.

2.2 Rate Limiter

A network function like the rate limiter is of great importance in nowadays infrastructures due to possible bursts of traffic or cyber-attacks. As such, its main feature is to limit the concurrent number of traffic flows and the rate at which those are established.

2.3 eBPF (Extended Berkeley Packet Filter)

eBPF (Extended Berkeley Packet Filter) [5] is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner. It is used in a number of Linux kernel subsystems, most prominently networking, tracing and security (e.g. sandboxing).

eBPF was introduced in Kernel 3.18 and is the evolution of the classic Berkeley Packet Filter (cBPF), once simply known as BPF.

cBPF is known to many as being the packet filter language used by tcpdump. Nowadays, the Linux kernel runs eBPF only and loaded cBPF bytecode is transparently translated into an eBPF representation in the kernel before program execution.

eBPF, then, could be used whenever kernel functionalities need to be expanded.

2.3.1 vCPU

eBPF is a *general purpose* RISC instruction set and was originally designed for the purpose of writing programs in a subset of C which can be compiled into eBPF instructions through a compiler back end (e.g. LLVM), so that the kernel can later on map them through an in-kernel JIT compiler into native opcodes for optimal execution performance inside the kernel.

eBPF consists of eleven 64 bit registers with 32 bit subregisters, a program counter and a 512 byte large eBPF stack space. Registers are named r0 - r10. The operating mode is 64 bit by default, the 32 bit subregisters can only be accessed through special ALU (arithmetic logic unit) operations. The 32 bit lower subregisters zero-extend into 64 bit when they are being written to.

Register r10 is the only register which is read-only and contains the frame pointer address in order to access the eBPF stack space. The remaining r0 - r9 registers are general purpose and of read/write nature.

2.3.2 Verifier

Since eBPF runs inside the kernel, it should guarantee that no operation could affect the operability of the underlying system.

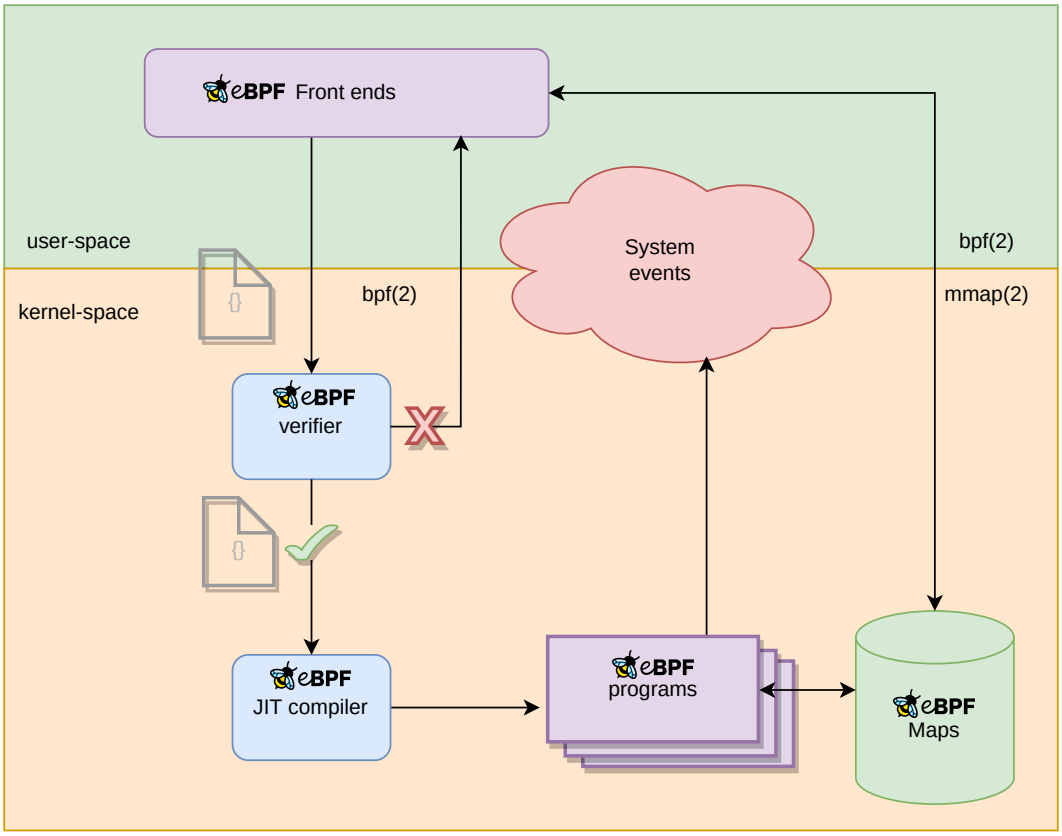

Figure 2.2: eBPF Architecture.

For this reason, eBPF programs, are checked by an in-kernel *verifier* before being executed, which is part of the whole eBPF framework.

The eBPF verifier is in charge of analyse the eBPF code and to whether it could continue the execution or it should be stopped.
Here, some restrictions on the eBPF code syntax and semantic are reported:

- The *maximum* instruction limit per program is restricted to 4096 eBPF instructions, which, by design, means that any program will terminate quickly. For kernel newer than 5.1 this limit was lifted to 1 million eBPF instructions.
- Although the instruction set contains forward as well as backward jumps, the in-kernel eBPF verifier will forbid loops so that termination is always guaranteed. In newer versions of the kernel, this has been lightened, allowing backward jumps as well as *limited* loops.However, unlimited loops are still forbidden due to safety.
- There is also a concept of tail calls that allows for one eBPF program to jump

into another one. This, too, comes with an upper nesting limit of 32 calls, and is usually used to decouple parts of the program logic, for example, into stages.

2.3.3 Helper Functions

Helper functions are a concept which enables eBPF programs to consult a core kernel defined set of function calls in order to exchange data with the kernel. Available helper functions may differ for each eBPF program *type*.

Each helper function is implemented with a commonly shared function signature similar to system calls. They allow to handle the life-cycle of an eBPF program as well as providing more complex operations which could have been forbidden by the verifier. However, since they are part of the kernel, they could not be modified. The list of available eBPF helper functions is rather long and constantly growing.

2.3.4 Maps

Maps are efficient key / value stores that reside in kernel space. They can be accessed from a eBPF program in order to keep state among multiple eBPF program invocations. They can also be accessed through file descriptors from user space and can be arbitrarily shared with other eBPF programs or user space applications.

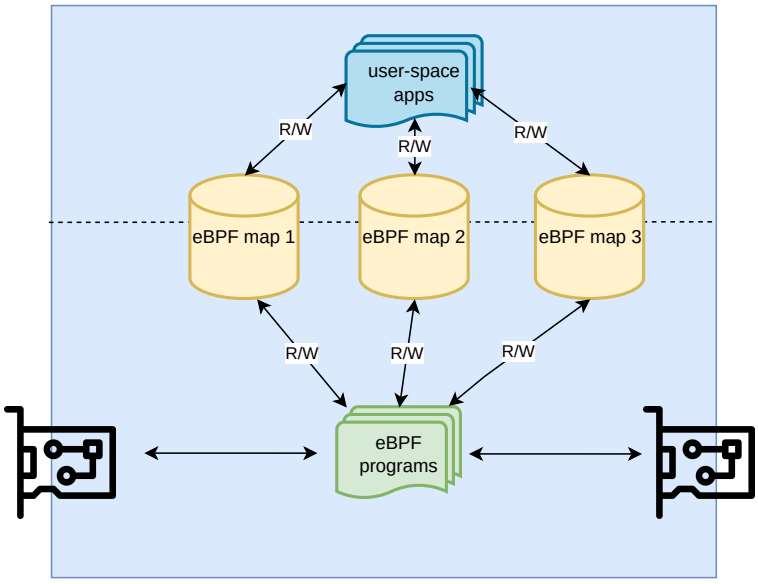

Figure 2.3: eBPF maps shared by kernel and user programs.

Map implementations are provided by the core kernel. There are generic maps that can read or write arbitrary data, but there are also a few non-generic maps

that are used along with helper functions. eBPF maps can broadly divided up to two categories:

- *per-CPU maps*: each map is owned and accessible by a single CPU (core).
- *shared maps*: a map could be shared across different CPUs (cores).

Non-generic maps that are particular types of maps whichtackle a specific issue which was unsuitable to be implemented solely through a eBPF helper function since additional state is required to be held across eBPF program invocations.

2.3.5 Object Pinning

eBPF maps and programs act as a kernel resource and can only be accessed through file descriptors, backed by anonymous inodes in the kernel. Advantages, but also a number of disadvantages come along with them.

User space applications can make use of most file descriptor related APIs, file descriptor passing for Unix domain sockets work transparently, etc, but at the same time, file descriptors are limited to a processes' lifetime, which makes options like map sharing rather cumbersome to carry out.

Thus, it brings a number of complications for certain use cases such as iproute2, where tc or XDP sets up and loads the program into the kernel and terminates itself eventually. With that, also access to maps is unavailable from user space side, where it could otherwise be useful, for example, when maps are shared between ingress and egress locations of the data path. Also, third party applications may wish to monitor or update map contents during eBPF program runtime.

To overcome this limitation, a minimal kernel space BPF file system has been implemented, where eBPF map and programs can be pinned to, a process called object pinning.

The eBPF system call has therefore been extended with two new commands which can pin *(BPF_OBJ_PIN)* or retrieve *(BPF_OBJ_GET)* a previously pinned object.

2.3.6 Tail Calls

Another concept that can be used with eBPF is called tail calls. Tail calls can be seen as a mechanism that allows one eBPF program to call another, without returning back to the old program. Such a call has minimal overhead as unlike function calls, it is implemented as a long jump, reusing the same stack frame.

Such programs are verified independently of each other. Only programs of the same type can be tail called, and they also need to match in terms of JIT

compilation, thus either JIT compiled or only interpreted programs can be invoked, but not mixed together.

Tail calls can be used to overcome the limited number of instructions per program, especially with older versions of the kernel, but most importantly they enable the creation of complex and dynamic service chains. Modular programs performing basic tasks can be developed independently and can then be combined to create rich functions, sharing data through maps. Thanks to the atomicity of the update operation on the PROG_MAP, programs can be swapped at run-time, re-configuring the chain without loosing any packet. This feature also enable the dynamic optimization of the code, allowing to inject refined programs based on run-time parameters such as the current configuration.

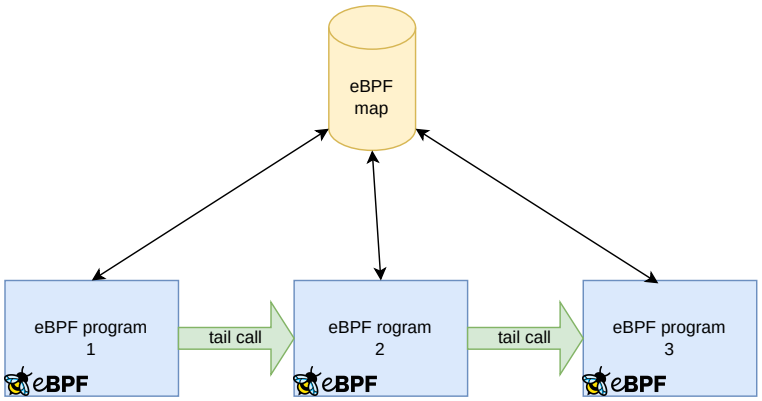

Figure 2.4: Tail calls.

2.3.7 Program Types

The execution of an eBPF program could be triggered by a kernel event. Depending on the type of the program, a specific kernel event, which goes under the name of *Hook Point*, would be in charge of the execution. Different Hook Point exist, addressing various kernel events. A short list of those is reported below:

- Networking
- System call execution
- Disk access
- Memory Management

There are two program types which are associated to Networking event: *XDP (eXpress Data Path)* and *TC (Traffic Control)*. Both types are in charge of packet processing.

eXpress Data Path (XDP)

The eXpress Data Path [6] provides a framework for eBPF that enables high-performance programmable packet processing in the Linux kernel. It runs the eBPF program at the earliest possible point in software, namely at the moment the network driver receives the packet.

At this point in the fast-path the driver just picked up the packet from its receive rings, without having done any expensive operations such as allocating a socket buffer for pushing the packet further up the networking stack.

Thus, the XDP BPF program is executed at the earliest point when it becomes available to the CPU for processing where only little data is provided to the program: the **struct xdp_md passed** to the main function contains pointers to the begin and end of the packet buffer, a pointer to a memory region to store additional metadata and indexes of the receive interface and receive queue. The return code of the program defines how the packet must be processed by the kernel. It can be dropped (**XDP_DROP** or **XDP_ABORTED**), can be redirected to another interface using helper functions **bpf_redirect()** and **bpf_redirect_map()** that return code **XDP_REDIRECT**, can be sent back to the same interface (**XDP_TX**) or can continue is path in the networking stack (**XDP_PASS**). XDP has three operation modes:

- *Native XDP*: it is the default mode where the XDP BPF program is run directly out of the networking driver's early receive path. Most widespread used NICs for 10G and higher support native XDP already.

- *Offload XDP*: the XDP eBPF program is directly offloaded into the NIC instead of being executed on the host CPU. Thus, the already extremely low per-packet cost is pushed off the host CPU entirely and executed on the NIC, providing even higher performance than running in native XDP.

- *Generic XDP*: for drivers not implementing native or offloaded XDP yet, the kernel provides an option for generic XDP which does not require any driver changes since run at a much later point out of the networking stack.

Traffic Control (TC)

This program type allows to bring the traffic in upper layers of the network stack (Traffic Control layer). Here, packets are already parsed and copied in pre-allocated memory structures named *socket buffer (skb)*, which contains additional useful metadata such as the protocol, the priority, the reception timestamp, VLAN associated metadata and layer 3 and 4 information. Although, TC programs do not have the same performance of XDP, they do bring some advantage:

- They do not require any driver changes since they are run at hook points in generic layers in the networking stack. Therefore, they can be attached to any type of networking device.
- They can be triggered out of ingress and also egress points in the networking data path as opposed to ingress only in the case of XDP.
- Since they have additional metadata, a richer set of functionalities is provided in order to perform more complex operations.

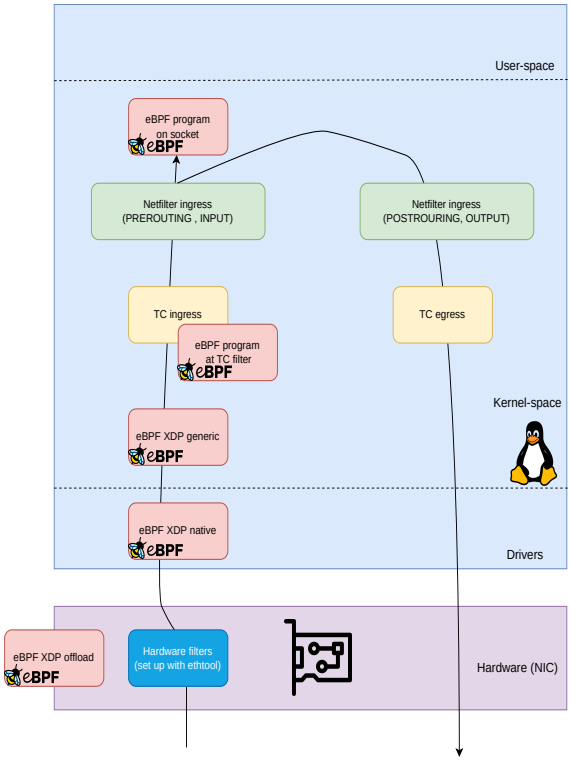

Figure 2.5: eBPF program types: XDP and TC hook points.

2.3.8 Toolchain

eBPF programs can be written using *restricted* C code so it can safely be executed within the kernel. Any violated contraints would trigger the Verifier to stop the program execution. LLVM is currently the only compiler suite providing a eBPF back end. gcc does not support eBPF untill now. The typical workflow is that eBPF programs are written in restricted C, compiled by LLVM into object or ELF files, which are parsed by user space eBPF ELF loaders (e.g. iproute2), and

pushed into the kernel through the *BPF* system call [7]. The kernel verifies the eBPF instructions and JITs them, returning a new file descriptor for the program, which then can be attached to a subsystem (e.g. networking). If supported, the subsystem could then further offload the eBPF program to hardware (e.g. NIC).

2.4 AF_XDP

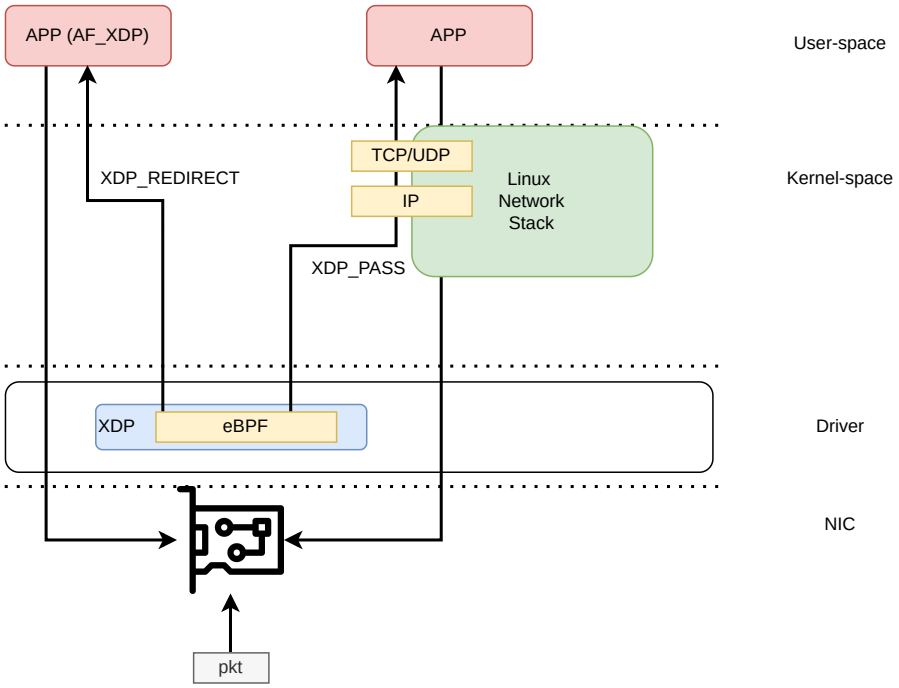

Figure 2.6: AF_XDP.

AF_XDP [8] is an address family that is optimized for high performance packet processing.

Using the *XDP_REDIRECT* action from an XDP program, the program can redirect ingress frames to other XDP enabled netdevs, using the bpf_redirect_map() function. AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in a user-space application.

Associated with each XSK are two rings: the RX ring and the TX ring. A socket can receive packets on the RX ring and it can send packets on the TX ring. These rings are registered and sized with the setsockopts XDP_RX_RING and XDP_TX_RING, respectively. It is mandatory to have at least one of these rings for each socket. An RX or TX descriptor ring points to a data buffer in a memory area called a UMEM. RX and TX can share the same UMEM so that a packet

does not have to be copied between RX and TX. Moreover, if a packet needs to be kept for a while due to a possible retransmit, the descriptor that points to that packet can be changed to point to another and reused right away. This again avoids copying data.

AF_XDP can operate in two different modes: *XDP_SKB* and *XDP_DRV*. If the driver does not have support for XDP, or XDP_SKB is explicitly chosen when loading the XDP program, XDP_SKB mode is employed that uses SKBs together with the generic XDP support and copies out the data to user space. A fallback mode that works for any network device. On the other hand, if the driver has support for XDP, it will be used by the AF_XDP code to provide better performance, but there is still a copy of the data into user space.

2.4.1 Concepts

In order to use an AF_XDP socket, a number of associated objects need to be setup. These objects and their options are explained in the following sections.

UMEM

UMEM is a region of virtual contiguous memory, divided into frames of equal size. It is created and configured (chunk size, headroom, start address and size) by using the XDP_UMEM_REG *setsockopt* system call. A UMEM is bound to a network device (*netdev*) and queue identifier, via the **bind()** system call.

An AF_XDP is socket linked to a single UMEM, but one UMEM can have multiple AF_XDP sockets.

The UMEM has two single-producer/single-consumer rings that are used to transfer ownership of UMEM frames between the kernel and the user-space application.

Rings

There are a four different kind of rings: *FILL, COMPLETION, RX and TX*. All rings are single-producer and single-consumer, so the user-space application need explicit synchronization of when multiple processes or threads access them. The UMEM always needs to use just two rings, a FILL and a COMPLETION ring whereas each socket needs its own RX or TX ring, or both. If a UMEM region is shared among four different sockets, which all handle both reception and trasmission traffic, then there would be one FILL ring, one COMPLETION ring, four RX rings and four TX rings. The size of the rings need to be of size power of two.

UMEM Fill Ring

The FILL ring is used to transfer ownership of UMEM frames from user-space to kernel-space. The UMEM addresses are passed in the ring. Frames passed to the kernel are used for the ingress path (RX rings). The user application produces UMEM addresses to this ring.

UMEM Completion Ring

The COMPLETION Ring is used transfer ownership of UMEM frames from kernel-space to user-space. As in the case of the FILL ring, UMEM addresses are passed in this ring. Frames passed from the kernel to user-space are frames that has been sent (TX ring) and can be used by user-space again. The user application consumes UMEM addresses from this ring.

RX Ring

The RX ring is the receiving side of a socket. Each entry in the ring is a struct xdp_desc descriptor. The descriptor contains a memory address as a UMEM offset and the length of the data.
If no frames have been passed to kernel via the FILL ring, no descriptors will (or can) appear on the RX ring. The user application consumes *struct xdp_desc* descriptors from this ring.

TX Ring

The TX ring is used to send frames. The struct xdp_desc descriptor is filled and passed into the ring. The user application produces struct xdp_desc descriptors to this ring.

2.4.2 Libbpf

libbpf [9] is a helper library for eBPF and XDP that easy the usage of these technologies. It also contains specific helper functions in *tools/lib/bpf/xsk.h* for facilitating the use of AF_XDP. It contains two types of functions: those that can be used to make the setup of AF_XDP socket easier and ones that can be used in the data plane to access the rings safely and quickly.

2.4.3 XSKMAP (BPF_MAP_TYPE_XSKMAP)

On XDP side there is a eBPF map type BPF_MAP_TYPE_XSKMAP (XSKMAP) that is used in conjunction with *bpf_redirect_map()* to pass the ingress frame to a socket. The user application inserts the socket into the map, via the bpf() system

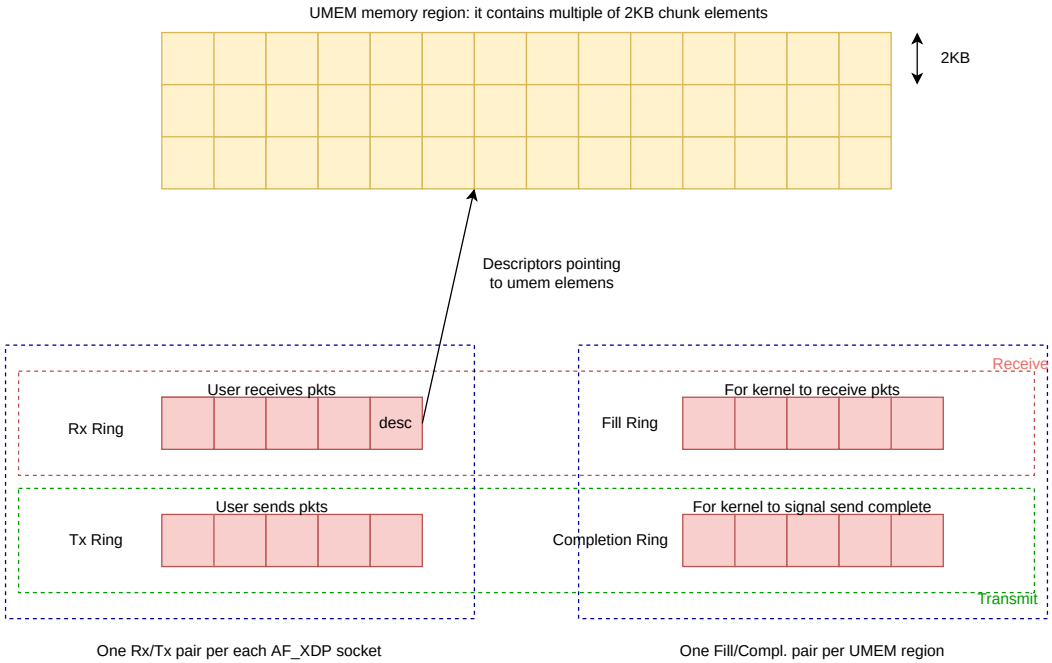

Figure 2.7: UMEM and Rings.

call. If an XDP program tries to redirect to a socket that does not match the queue configuration and netdev, the frame will be dropped.

2.5 Why AF_XDP

AF_XDP is not the only technology which allows to bring traffic directly to user-land applications, bypassing the kernel intervention which eventually increases the overall performance.

One of the most common alternative is surely *DPDK (Data Plane Development Kit)* [3], an open-source project which provides a set of data plane libraries and network interface controller polling-mode drivers for offloading TCP packet processing from the operating system kernel to processes running in user space.

This offloading achieves higher computing efficiency and higher packet throughput than is possible using the interrupt-driven processing provided in the kernel.

On the other side, though, this polling-mode drivers come with high resource requirements since they need to take over a complete subset of the CPU cores, which could not be suitable in some cases especially if considering cloud environment.

AF_XDP provides both polling-mode and interrupt-driven drivers, giving the flexibility of choosing which one depending on the scenario, requirements and

run-time needs.

Moreover, DPDK could only be used with the appropriate hardware: it supports only a subset of all available NICs as well requiring specific drivers to compatible with. This brings the best performance as being optimized for specific technologies from one side, and being less flexible by requiring the user specific setup from the other side.

This, once again, encourages the adoption of AF_XDP as a solution supporting a wider range of NICs and drivers, which better suits cloud scenarios where *COTS (Common Off The Shelf)* hardware are usually used.

Moreover, even if is not as powerful as DPDK, its performance are still high.

2.5.1 Libbpf-bootstrap

libbpf-bootstrap is a scaffolding playground setting up the infrastructure needed to easy writing BPF programs and providing APIs to handle their life-cycle. It takes into account best practices developed in eBPF community over last few years and provides a modern and convenient workflow with, arguably, best eBPF user experience to date.

```
.
├── ...
├── ...
├── LICENSE
├── README.md
├── src
│   ├── bootstrap.bpf.c
│   ├── bootstrap.c
│   ├── bootstrap.h
│   ├── Makefile
│   ├── minimal.bpf.c
│   ├── minimal.c
│   ├── vmlinux_508.h
│   └── vmlinux.h -> vmlinux_508.h
└── tools
    ├── bpftool
    └── gen_vmlinux_h.sh
```

libbpf-bootstrap bundles libbpf as a submodule in libbpf/ sub-directory to avoid depending on system-wide libbpf availability and version.

tools/ contains bpftool binary, which is used to build eBPF skeletons of the eBPF code. Similarly to libbpf, it's bundled to avoid depending on system-wide bpftool availability and its version being sufficiently up-to-date.

Makefile defines the necessary build rules to compile all the supplied (and your

custom ones) BPF apps. It follows a simple file naming convention:

- <app>.bof.c files are the BPF C code that contain the logic which is to be executed in the kernel context;
- <app>.c the user-space C code, which loads BPF code and interacts with it throughout the lifetime of the application;

2.6 Related Works

The interest of having reliable and efficient network functions has been increased thanks to the performance constraints which came with new technologies like the 5G, where the virtualization of data plane component plays a crucial part to meet the flexibility concerns of those architectural novelties. Many works have been advanced in this direction, where the usual target has been chaining different simple network functions to address real-world scenarios.

Before analysing proposed solutions of rate limiters applied in the context mentioned before, it is worth to mention the possibility of applying QoS in Linux.

tc [10] command is used to configure Traffic Control in the Linux Kernel. Traffic Control consist in:

- **SHAPING**: When the traffic is shaped, its transmission rate is under control. Shaping may be more than lowering the available bandwidth - it is also used to smooth out bursts in traffic for better network behaviour. Shaping occurs on egress.
- **SCHEDULING**: By scheduling the transmission of packets it is possible to improve interactivity for traffic that needs it while still guaranteeing bandwidth to bulk transfers. Reordering is also called prioritizing, and happens only on egress.
- **POLICING**: Whereas shaping deals with transmission of traffic, policing pertains to traffic arriving. Policing thus occurs on ingress.
- **DROPPING**: Traffic exceeding a set bandwidth may also be dropped forthwith, both on ingress and on egress.

Processing of traffic is controlled by three kinds of objects: qdiscs, classes and filters. All those aspects are to be of great importance.
In [11], the authors emphasized on the possibility to build a complex service chain, starting from smaller and simpler network functions. They designed and implemented a prototype of Mobile Gateway, an important component in 5G Mobile Packet Core (MPC), in charge of interconnecting mobile users to Data

Packet Network like Internet. The solution has mainly build within an open source software framework for Linux that provides fast and lightweight network functions, Polycube. [12]. This work mainly relies on eBPF as underlying technology, where a special attention has also been given on a function like a rate limiter, which is definitely of extreme importance and whose behaviour could heavily affect the overall performance.
The solution implement and compare three different algorithms for the rate limiter component:

- Window Counter
- Token Bucket
- Sliding Window

providing a benchmark both on the overall performance and on the solution accuracy.

[13] has an eBPF Package Repository where, among different network functions, there is also a simple prototype of rate limiter. Quoting their documentation [14], "*Adding the connection and rate-limiting functionality to our edge proxies and load-balancer protects our compute resources from getting overwhelmed when there is a sudden burst of traffic that is beyond what our resources are capable of handling.*" This emphasizes the importance of such a component in modern networks. This solution mainly relies on the XDP hook point, since it allows to drop packets at a very high rate.

Open Virtual Switch (OVS) has also implemented a rate limiter based on its technology [15]. The solution is based on Policing, which drops any packet beyond the specified rate. The author suggests that, "*specifying a larger burst size lets the algorithm be more forgiving, which is important for protocols like TCP that react severely to dropped packet*".

For TCP traffic, setting a burst size to be a sizeable fraction (e.g., $> 10\%$) of the overall policy rate helps a flow come closer to achieving the full rate. If a burst size is set to be a large fraction of the overall rate, the client will actually experience an average rate slightly higher than the specific policing rate. For UDP traffic, set the burst size to be slightly greater than the MTU and make sure that your performance tool does not send packets that are larger than your MTU (otherwise these packets will be fragmented, causing poor performance).

Chapter 3

Prototype Architecture

3.1 General Architecture

This prototype has been conceived to explore the possibility of combining two different technologies in implementing the same network function: eBPF and AF_XDP. This experiment has carried out in order to understand whether network-oriented tasks could run within the same machine even if relying upon different technologies. The underlying target is that, there might be the need of running network tasks on general-purpose servers, where there would also be running other general-purpose applications. However, network functions usually come with strict requirements since they needs to spread the traffic as soon as possible. Here, the idea is to leave the management of *local traffic* targeting local applications to eBPF then in charging of redirecting the packets to the final destination, whereas giving the responsibility of managing the *remote traffic* to AF_XDP. This choice has been done since AF_XDP could achieve higher performance in traffic forwarding as it is going to be detailed later on in the evaluation chapter, due to its capability of bypassing the kernel, saving additional overhead in all those cases where kernel-provided services are not strictly needed by applications. In any case, an eBPF program is strictly needed to redirect packets to user-land applications through AF_XDP sockets. This is achieved thanks to the XDP action *XDP_REDIRECT*, which is aimed to redirect network traffic to other network devices.

The XDP program could take several action, but for testing this prototype only three out of them are used:

- XDP_REDIRECT: This action is mainly used by XDP program to redirect traffic to the AF_XDP application.

- XDP_TX: This action is mainly used by the XDP program to forward back the traffic.

- XDP_DROP: This action is mainly used by the XDP program to drop unmanaged traffic.

Those actions are depicted in blue in the following picture, whereas all the other ones are reported for the sake of completeness.

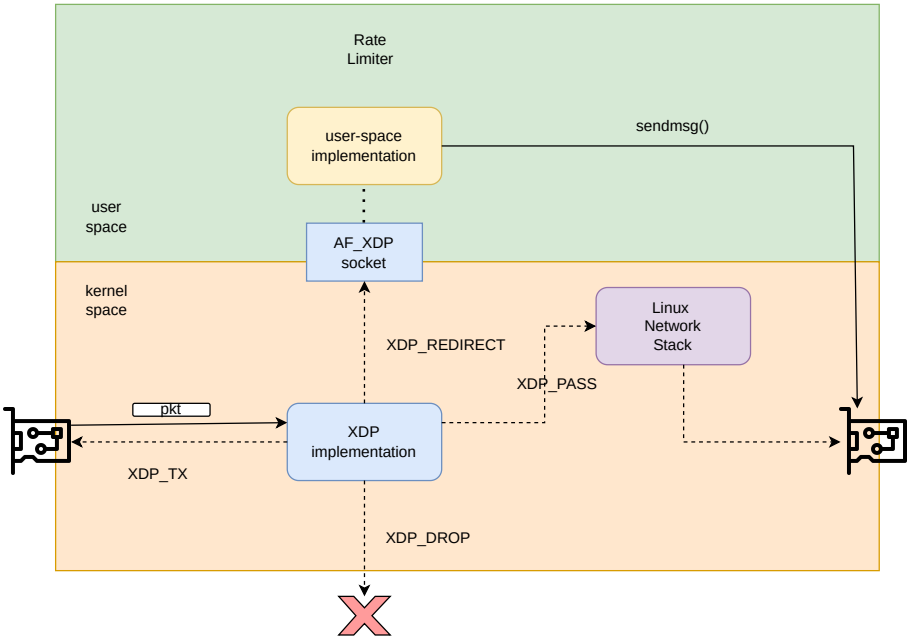

Figure 3.1: Rate limiter general architecture: eBPF and AF_XDP path.

Given this need, an opportunity could be taken in order to double the implementation of the rate limiter; depending on the receiving traffic profile, it would be handled whether by the eBPF implementation or by the AF_XDP counterpart.

3.2 Rate Limiter

The Rate Limiter is a network function which mainly provides rate limiting functionalities. In this implementation, it also provides access control. The service is configured with a list of contracts, which define how a certain class of traffic must be handled. The actions that could be applied are:

- PASS: Let the packet pass.
- DROP: Drop the packet.
- LIMIT: Apply rate limiter.

The classification logic is not implemented into this prototype, which relies on packet metadata to decide what to do with the traffic. This allows the model to correctly process both up-link traffic (from a user to the Internet) and the down-link traffic (from Internet to the final user).

Furthermore, two techniques could be considered when it comes to rate limiting functionalities: traffic shaping and traffic policing.

3.2.1 Traffic Shaping and Traffic Policing

Shaping and Policing are two bandwidth management techniques which adopt the traffic behaviour to a desired profile.

Traffic policing propagates bursts. When the traffic rate reaches the configured maximum rate, exceeding traffic is dropped. The result is an output rate which grows and shrink over the time.

On the other side, traffic shaping keeps exceeding packets in queue and then schedules them for later transmission. The results is a smoother output rate.

However, shaping requires the presence of a queue and as such there is the need of sufficient memory to buffer the delayed traffic, while policing does not.

Queuing is an outbound concept and as such only policing can be applied to inbound traffic on a interface. In addition, shaping requires a scheduling function for later transmission of any delayed packets

Traffic Shaping

Traffic Shaping is the most flexible approach as it only delays packets which would break the established *traffic profile*, by putting them in a buffer. This techinque allows to handle bursty traffic without loosing the packets as far as they do not exceed the buffer capacity. Shaping is commonly applied at the network edge to control traffic entering the network, in order to avoid congestion and latency increase. As such, the single parameter it needs is the average rate. One of the disadvantages of this technique is that, since it only sends packets at a fixed rate, it can cause under-utilization of network resources when traffic volume is low and resources could be consumed in a bursty way without contention

Traffic Policing

On the other side, Policing is a stiffer approach to bandwidth management. For this reason, it requires two parameters: the average rate and the maximum burst size. Packets exceeding one of the metrics are either dropped or marked as non compliant. An alternative implementation described in RFC 2697 provides a more granular control and requires three parameters: the *Committed Information Rate* (CIR), the *Committed Burst Size* (CBS) and the *Excess Burst Size* (EBS). Packets

can then be split in three different categories identified by a color: a packet is "green" if it doesn't exceed the CBS, "yellow" if it does exceed the CBS, but not the EBS, and "red" otherwise.

Overall, Traffic Policing is recommended for Voice, Video and Rich media traffic where generally UDP based communication takes place. On the other hand Traffic Shaping is recommended for TCP based applications which can bear delay in traffic but need high data transfer rate like SAP etc.

3.2.2 Rate Limiting algorithms

Token Bucket

The algorithm requires two input parameters for each flow: the desired average bit rate and the maximum burst size.

A bucket for every class of traffic is used and filled with tokens. A token represents one bit of information. Two parameters are associated to every bucket:

- The maximum number of tokens it can contain, equal to the maximum burst size.
- The refill rate (expressed in tokens per second), equal to the desired average bit rate (in bits per second).

Every time a packet is processed it needs to consume a number of tokens from the corresponding bucket equal to its size. In case there aren't enough tokens the packet is discarded.

When the packet rate is below the desired one the output is not influenced by the algorithm, since tokens are inserted into the bucket faster then they are consumed.

When the rate grows above the desired threshold initial packets are still forwarded,producing a burst whose size can be at most equal to the size of the bucket, and further packets are limited to the desired rate.

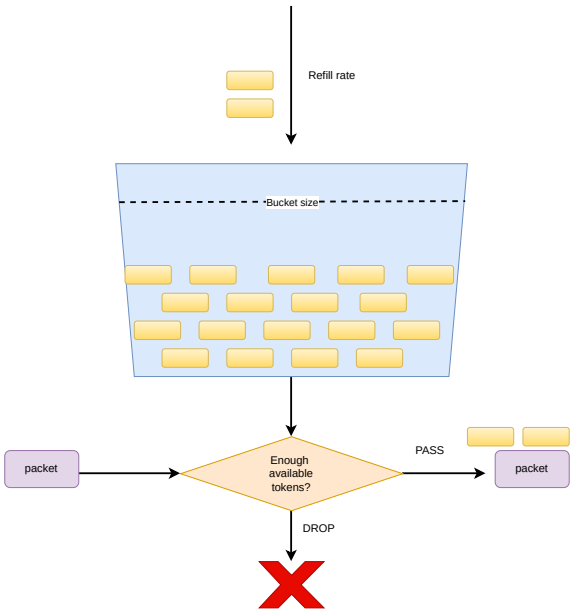

Figure 3.2: Token Bucket algorithm.

3.2.3 Private State vs Shared State

Taking in account the double presence of the rate limiter in two different layers, eBPF and AF_XDP, two possibilities of keeping the state of the application are possible: private state and shared state.

Private State

With the *private* approach, the two applications, eBPF XDP and AF_XDP, have their own state where to manage the traffic: reading the policies to be applied and saving the bandwidth consumption.

The main advantage of this approach is that the two applications do not interfere with each other, saving the management of synchronization accesses.

However, there are a couple of drawbacks which should seriously be considered when dealing with data-centers' environment:

- *The double usage of resources*: it could be a big problem in real scenarios.
- *The lack of flexibility*: static setup of which portion of traffic is handled by eBPF and AF_XDP.

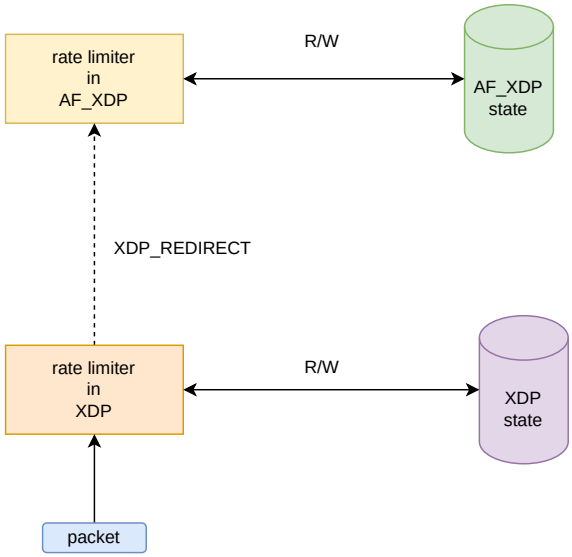

Figure 3.3: Private state.

Shared State

The two applications could have a unique and *shared* state, which they access concurrently both for applying policies and for saving bandwith consumption.

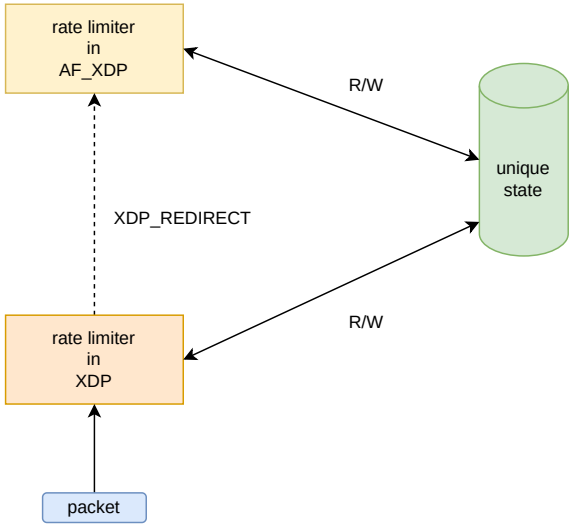

Figure 3.4: Shared state.

This approach solves the issues coming from the stand-alone solution: waste of resources and lack of flexibility.

However, this solution brings potential problems with concurrent accesses: there are no synchronization primitives to coordinate kernel threads and user threads.

A workaround to this problem has been found, which would be detailed in the *Implementation* section, which is the next chapter.

Chapter 4

Prototype Implementation

This chapter explains how the components introduced in the *Architecture* section have been implemented, showing also some fragments of code to better understand the logic behind it, and seeing some workaround used to solve problems faced along the process.

The only programming language used in this solution is C, both for the thread handling the control plane and for the fast-path of the data plane, the latter being implemented both in kernel and in user-space.

4.1 Rate Limiter

The solution has been implemented mainly in two source files:

- **rate_limiter_kern.c**: this source implements the eBPF program running at the XDP hook point.
- **rate_limiter_user.c**: this source implements both the AF_XDP programm running in user-space and the thread managing the control plane.

4.1.1 Private State vs Shared State

Before going into details of the prototype implementation, it is useful to analyze which has been the path that led to the final solution and what had been the issues that were faced.

4.1.2 Private State

The first implementation of this prototype followed the private design which has been introduced in the *Architecture* chapter.

As the name states, it requires the XDP and AF_XDP programs to operate on two different memory regions. Moreover, since these two programs run within different execution environments, as XDP being an in-kernel solution whereas AF_XDP running in user-space, they would need different kind of memory.

Here, an eBPF Hash Map is used for the XDP program:

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct session_id);
    __type(value, struct contract);
    __uint(max_entries, MAX_CONTRACTS);
} contracts SEC(".maps");
```

Listing 4.1: eBPF Hash Map for XDP

whereas for the AF_XDP application, a user-customized data structure should be adopted. A hash-table-like structure named *khashmap* has been implemented:

```
struct khashmap {
    uint32_t key_size;
    uint32_t value_size;
    uint32_t max_entries;
    struct khashmap_bucket *buckets;
    void *elems;
    void *next_free;
    atomic_int count;    /* number of elements in this hashtable */
    uint32_t n_buckets; /* number of hash buckets */
    uint32_t elem_size; /* size of each element in bytes */
    uint32_t hashrnd;
};
```

Listing 4.2: *khashmap*: a customized data structure for AF_XDP

In both cases, a key-value association is needed where the key is represented by the session ID and the value is the contract to be applied to the traffic.

```
struct session_id {
    uint32_t saddr;
    uint32_t daddr;
    uint16_t sport;
    uint16_t dport;
    uint8_t proto;
} __attribute__((packed));
```

Listing 4.3: Session ID used as a key

```
struct contract {
  int8_t action;
  int8_t local;
  struct bucket bucket;
```

```
};
```

Listing 4.4: Contract used as a value

The *bucket* struct within the *contract* stores the parameter which model the *Token Bucket* algorithm implementation:

```
struct bucket {
  int64_t tokens;           /* tokens currently available */
  uint64_t refill_rate;     /* refill rate/ms */
  uint64_t capacity;        /* maximum bucket size */
  uint64_t last_refill;     /* timestamp last refill */
};
```

Listing 4.5: The *bucket* struct

In this way, the desired network function could run both in XDP and AF_XDP at the same time, giving the possibility of handling traffic in both levels.

However, as already mentioned in the previous chapter, this is not a very efficient design choice for the drawbacks that it has behind: static setup and double usage of resources.

This could not be acceptable in cloud environments, where flexibility is among the highest requirements and resources should be managed efficiently.

4.1.3 Shared State

The implementation choices and issues of adopting the *shared state* approach are detailed here.

Two different strategies could be used to have a shared state:

- Use of a single eBPF map also accessible from the user-level application
- Use of a single shared portion of memory accessible from both program.

An analysis of both approach is going to be described in order to understand which one would fit better for the original goals.

A single eBPF map for both application

One single eBPF map could be used for managing both the traffic which flows in XDP and in AF_XDP. This is possible thanks to the presence of libbpf helper facilities which allow user-land applications to access eBPF maps. There are basically two helper functions which provides the possibility of exchanging data between a user-level and an eBPF program:

- **int bpf_lookup_elem(int fd, const void *key, void *value)**: it looks up an element with a given key in the map referred to by the file descriptor

fd. If an element is found, the operation returns zero and stores the element's value into value, which must point to a buffer of value_size bytes. If no element is found, the operation returns -1 and sets errno to ENOENT.

- **int bpf_update_elem(int fd, const void *key, const void *value, uint64_t flags)**: it creates or updates an element with a given key/value in the map referred to by the file descriptor fd. On success, the operation returns zero. On error, -1 is returned and errno is set to appropriate values.

However, using a single eBPF map which is accessible both from both parts at the same time is not the best solution since it would require the user-space program to trigger a system call whenever it access the map.

This could not be acceptable in all those cases where performance is an important factor: traffic management is one of those and so an alternative solution would be more appropriate.

Another strategy has also been considered: the usage of a Private approach with a periodic comparison between the eBPF map used by XDP and the khashmap used by AF_XDP. However, this solution would be rather imprecise because of the latency spent in reading the data and then comparing them.

A Single Shared Memory Area

Since the usage of a single eBPF map for both XDP and AF_XDP program could be an overkill for the overall prototype performance, a different model of shared memory is required.

The basic idea is to find a solution which does not require the user-level application to trigger a system call for accessing the data. Here, many solutions could be taken in account but in general implementing and handling a shared memory model between user threads and kernel threads is not easy.

Moreover, another problem to be seriously taken in account is the lack of synchronization primitives which would manage concurrent access between kernel threads and user threads.

All of these problems have been addressed and solved thanks to the support of a scaffloading playground technology named *libbpf-bootstrap*, whose usage in the prototype is going to be detailed in the next section.

4.1.4 Rate Limiter application

The code of the prototype is going to be showed in order to understand how it has been implemented with the support of the *libbpf-bootstrap* library.

The XDP side

Here is some part of the eBPF XDP side code is reported:

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
...
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct session_id);
    __type(value, int);
    __uint(max_entries, MAX_CONTRACTS);
} positions SEC(".maps");
struct contract contracts[MAX_CONTRACTS] = {};
SEC("xdp")
int rate_limiter(struct xdp_md *ctx) {
    /* additional instructions */
    ...
    int *position = bpf_map_lookup_elem(&positions, &key);
    if(!position){
        return XDP_DROP;
    }
    volatile int array_index = *position;
    int safe_index = array_index;
    if(safe_index < 0 || safe_index >= MAX_CONTRACTS){
        return XDP_DROP;
    }
    struct contract *contract = &contracts[safe_index];
    switch (contract->action) {
        case ACTION_PASS:
        return XDP_PASS;
        break;
        case ACTION_LIMIT:
        return limit_rate(ctx, contract);
        break;
        case ACTION_DROP:
        return XDP_DROP;
        break;
    }
    return XDP_TX;
}
```

Listing 4.6: The XDP rate limiter program

One important part to analyse is an interesting eBPF feature: the usage of global variables. ***struct contract contracts[MAX_CONTRACTS];*** defines a global variable which eBPF code can read and update just like any user-space C code would do with a global variable. In the rate limiter case, it stores the policies to be applied to the upcoming traffic.

It is extremely convenient and also performing to use eBPF global variables for maintaining the state of the eBPF program. Additionally, such global variables can be read and written from the user-space side. This feature is available starting from Linux 5.5 version.

It is frequently used for things like configuring eBPF application with extra settings, low-overhead stats, etc. It can also be used to pass data back-and-forth between in-kernel eBPF code and user-space control code.

The eBPF program, firstly performs some header checks in order to understand if the traffic is legit. This part has been skipped in the above figure for the sake of brevity. Then, starting from the packet information, the program looks for the policy to be applied: here, there is actually an intermediate step which goes through the use of an eBPF map. The latter, has been introduced to solve hash collisions which are going to be detailed later.

After this step, if a policy is retrieved from the array global variable *contracts*, the program will take the appropriate action to manage the traffic. In case the traffic need to be limited, an inline function *limit_rate* is called:

```
1  static inline int limit_rate(struct xdp_md *ctx, struct contract *
       contract) {
2      void *data = (void *)(long)ctx->data;
3      void *data_end = (void *)(long)ctx->data_end;
4      //  Consume tokens
5      int64_t needed_tokens = (data_end - data + 4) * 8;
6      uint8_t retval;
7      if (contract->bucket.tokens >= needed_tokens) {
8          __sync_fetch_and_add(&contract->bucket.tokens, -needed_tokens
       );
9          retval = XDP_TX;
10     } else {
11         retval = XDP_DROP;
12     }
13     return retval;
14 }
```

Listing 4.7: *limit_rate* inline function

In this code, there is the presence of atomic hardware instructions which turns to be essential in managing concurrent access between the XDP and AF_XDP programs. The synchronization issues that have been faced will be described in one of the last section of this chapter.

The AF_XDP side

From the AF_XDP program, which runs in user-space, all things are tied together thanks to the use of a special header:

```c
#include "rate_limiter.skel.h"
```

Listing 4.8: including eBPF skeleton header in user-space program

This includes a eBPF skeleton of the eBPF code in rate_limiter_kern.c. It is auto-generated by bpftools and reflects the high-level structure of rate_limiter_kern.c.

It also simplifies the eBPF code deployment logistics by embedding contents of the compiled eBPF object code inside the header file, which gets included from the user-space code. No extra files to deploy along the application binary are needed.

The eBPF skeleton is purely a libbpf construct, it is not kernel-related. However, it provides useful facilities to easy the developement of eBPF applications. [16]

```c
#include <stdlib.h>
#include <bpf/libbpf.h>
struct rate_limiter_kern {
    struct bpf_object_skeleton *skeleton;
    struct bpf_object *obj;
    struct {
        struct bpf_map *xdp_stats;
        struct bpf_map *xsks;
        struct bpf_map *positions;
        struct bpf_map *bss;
    } maps;
    struct {
        struct bpf_program *rate_limiter;
    } progs;
    struct {
        struct bpf_link *rate_limiter;
    } links;
    struct rate_limiter_kern__bss {
        struct contract contracts[20];  /* 20 is the value of
    MAX_CONTRACTS */
    } *bss;
};
static void
    rate_limiter_kern__destroy(struct rate_limiter_kern *obj);
static inline int
    rate_limiter_kern__create_skeleton(struct rate_limiter_kern *obj)
    ;
static inline struct rate_limiter_kern *
    rate_limiter_kern__open(void);
static inline int
    rate_limiter_kern__load(struct rate_limiter_kern *obj);
static inline struct rate_limiter_kern *
    rate_limiter_kern__open_and_load(void);
static inline int
    rate_limiter_kern__attach(struct rate_limiter_kern *obj);
static inline void
    rate_limiter_kern__detach(struct rate_limiter_kern *obj);
```

Listing 4.9: eBPF skeleton header file

It has the struct bpf_object *obj; which can be passed to libbpf API functions. It also has maps, progs, and links sections, that provide direct access to BPF maps and programs defined in the eBPF code.

These references can be passed to libbpf APIs directly to do something extra with eBPF maps, programs and links. Skeleton can also optionally have bss, data, and rodata sections that allow **direct** access to eBPF global variables from user-space, without triggering any system call. In this case, the contracts array eBPF global variable corresponds to the *bss->contracts* field.

There are also a set of methods in the eBPF skeleton, used to handle the lifecycle of the eBPF application:

- <eBPFprog-name>___create_skeleton: to create the skeleton object starting from the *object* file, which is the result of the compilation phase.
- <eBPFprog-name>___destroy: to destroy the eBPF skeleton object.
- <eBPFprog-name>___open: to open a file associated to the eBPF program.
- <eBPFprog-name>___open: to load an open eBPF file.
- <eBPFprog-name>___attach: to attach the loaded eBPF program to a specific hook point.
- <eBPFprog-name>___detach: to detach an eBPF program from a specific hook point.

```
skeleton = rate_limiter_kern__open();
err = rate_limiter_kern__load(skeleton);
if (err) {...} /* error */
int if_index = if_nametoindex(config.interfaces[0]);
if(!if_index){..} /* error */
skeleton->links.rate_limiter = bpf_program__attach_xdp(skeleton->
    progs.rate_limiter, if_index);
if(!skeleton->links.rate_limiter){...} /* error */
```

Listing 4.10: handling eBPF life-cycle with eBPF skeleton

From the code above, it is useful to mention that the life-cycle of the eBPF XDP program has been managed with the support of the eBPF skeleton. The only exeption is the attachment phase, which has been done through the helper function *bpf_program___attach_xdp(...)* because of issues of faced with the skeleton's counterpart.

Accessing eBPF global variables in user-space application

```
struct rate_limiter_kern *skeleton;
...
void *pkt_end = pkt + len;
    struct session_id key;
    int *position;
    /* packet handlers management */
    ...
    position = khashmap_lookup_elem(&positions, &key);
    struct contract *contract = &skeleton->bss->contracts[*position];
    switch (contract->action) {
    case ACTION_PASS:
        return 0;
        break;
    case ACTION_LIMIT:
        return limit_rate(pkt, len, contract);
        break;
    case ACTION_DROP:
        return -1;
        break;
    }
    return 0;
```

Listing 4.11: The AF_XDP rate limiter program

The logic of AF_XDP program is a copy of the XDP side. However, they differ in accessing the data needed for traffic policing.

In AF_XDP, there is the direct access to the array defined as eBPF global variable *contracts* thorugh the eBPF skeleton, which was previously initialized with a proper call to the function *rate_limiter_kern__open()*.

4.1.5 eBPF skeleton generation

The eBPF skeleton header file is generated though the kernel tool *bpftool*. The process starts from the eBPF source code written in C which, after being compiled, it generates an object file. The latter, is taken as an input by the bpftool in order to generate the eBPF skeleton as an header file. This header file, is then imported into the user-level application, in this case the rate limiter in AF_XDP, to directly access the date specified in the eBPF program.

This process, as already mentioned, is needed in order to save system call whenever the XDP and AF_XDP program need to interact. The syntax for eBPF skeleton generation is: **bpftool gen skeleton file.obj**. All the setup needed for skeleton generation is automatically handled with instructions reported in a Makefile belonging to the project:

```
1 $(EXAMPLES_SKEL): %.skel.h: %_kern.o $(EXAMPLES_KERN)
2     ./bpftool gen skeleton $< > $@
```

Listing 4.12: Lines in Makefile for skeleton generation

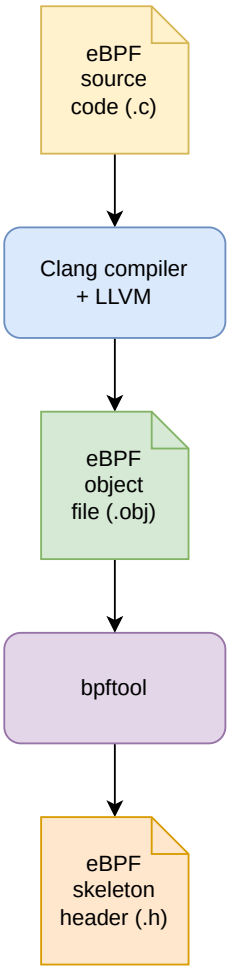

Figure 4.1: eBPF skeleton generation.

4.1.6 Hash Collisions

Problem

This solution adopts the use of a single memory region to access data used by both sides of the application, eBPF and AF_XDP. As already seen, it is an array of ***struct contract***, which stores the policies specification to be applied to the traffic.

Moreover, there is the need of associating the arriving traffic to the policy to be applied: here, the solution relies on a further data structure ***struct session*** which represents the session ID of the handled packets.

However, this requires to map a session ID to a specific index of the array. For achieving this goal, the kernel library *jhash* could be used.

The idea is taking the session ID when the packet arrives, giving it as an input to the jhash library, and using the result of such a computation as the index for looking up within the array.

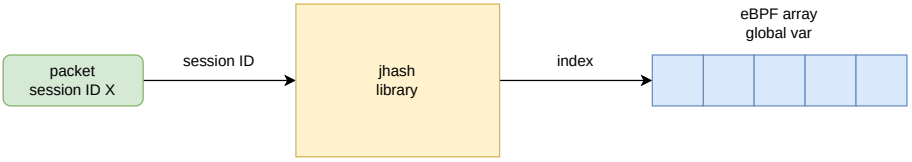

Figure 4.2: Use of jhash library.

This solution works as expected but there is still a problem to be addressed in the XDP side. Due to the presence of the eBPF verifier, the following instructions could raise some issues and the execution of the XDP program would be stopped.

```
int *position = bpf_map_lookup_elem(&positions, &key);
if(!position){
    return XDP_DROP;
}
```

Listing 4.13: The XDP rate limiter program

To solve this problem, the following workaround has been adopted:

```
int *position = bpf_map_lookup_elem(&positions, &key);
if(!position){
    return XDP_DROP;
}
volatile int array_index = *position;
int safe_index = array_index;
if(safe_index < 0 || safe_index >= MAX_CONTRACTS){
    return XDP_DROP;
}
```

Listing 4.14: The XDP rate limiter program

In this way we avoid some compiler optimizations triggering the eBPF verifier, which would stop the program execution.

Although all the problems related to the programs' executions have been solved, another issue comes from the implementation point of view: the hash values computed on the session ID should be truncated on the array capacity because of the association with its indexes, but this could potentially cause high-ratio

collisions. This is even a bigger problem in eBPF context since it runs in restricted environments and so arrays could not host many data. As a result, the smaller is the data structure holding data the higher is the collision-ration.

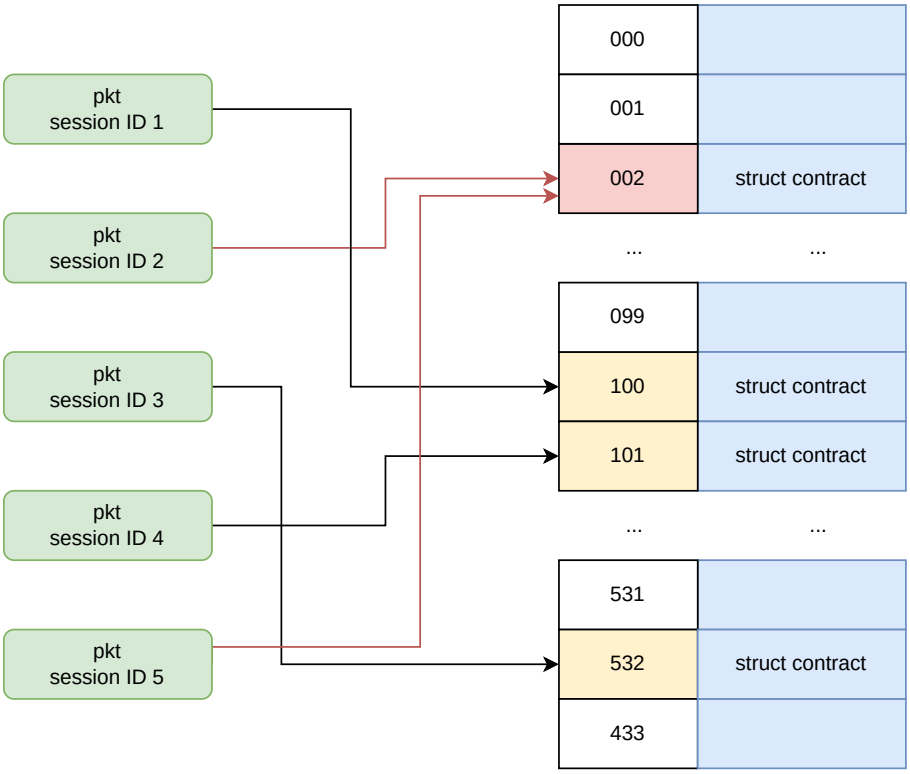

Figure 4.3: Example of hash collision.

Possible solutions

A naive solution to the problem mentioned before is to handle collisions with chaining. This requires to have a linked list of buckets for each hash value which allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. However, this approach might still require a lot of memory which is not compliant with eBPF restrictions. Moreover, the lookup-phase could be rather expensive in all those cases where the chains are very long.

To overcome the lookup overhead, a more sophisticated algorithms could be implemented. A possible solution is the *Cuckoo hashing* algorithm, which is a scheme for resolving hash collisions of values of hash functions in a table, with worst-case constant lookup time [17].

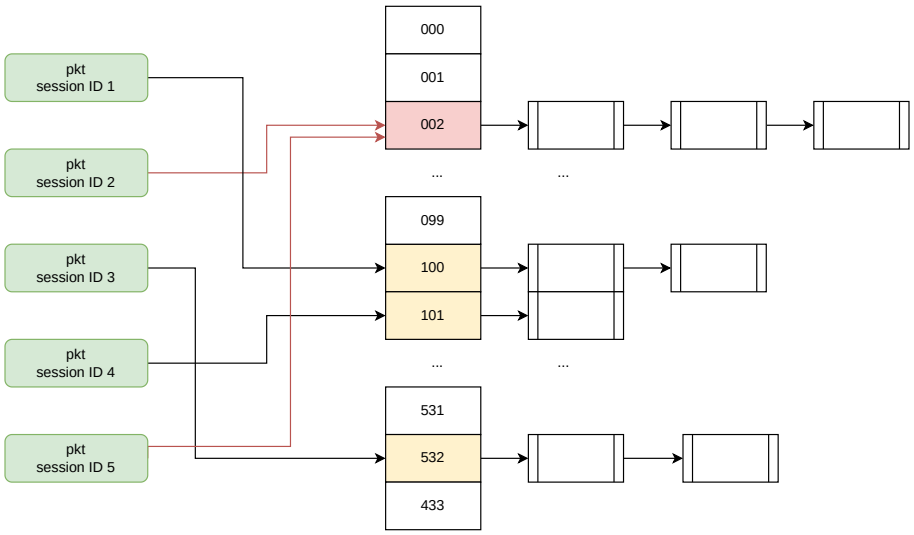

Figure 4.4: Hash Chaining.

Furthermore, the problem of using many resources still persists as well as the complexity of the algorithm could not fit the eBPF restricted environment since it would require a lot of cycle loops, instructions and memory.

Final solution

To overcome the issues mentioned in the previous subsections, the solution has leveraged on the facilities already present in the system: eBPF maps and the khash map previously seen during the AF_XDP analysis. However, this structures do not store the policies directly due to the need of having a shared state. They are instead in charge of storing the position within the eBPF global array variable, where the actual policies data are stored. Basically, they function as an intermediate step in the data access. Even an intermediate step has been added in the data access phase, no additional overhead has been experienced from the performance point of view. Moreover, this approach take advantage of the optimised data structures designed for eBPF as well as leaving the user from the implementation burden of complex algorithms.

4.1.7 Refilling thread

One of the last component to be addressed is how this solution refill the token bucket in order to allow upcoming traffic to get transited.

The main concern is that this task should update data which are shared between user-space application, the AF_XDP side of the rate limiter, and a kernel-space

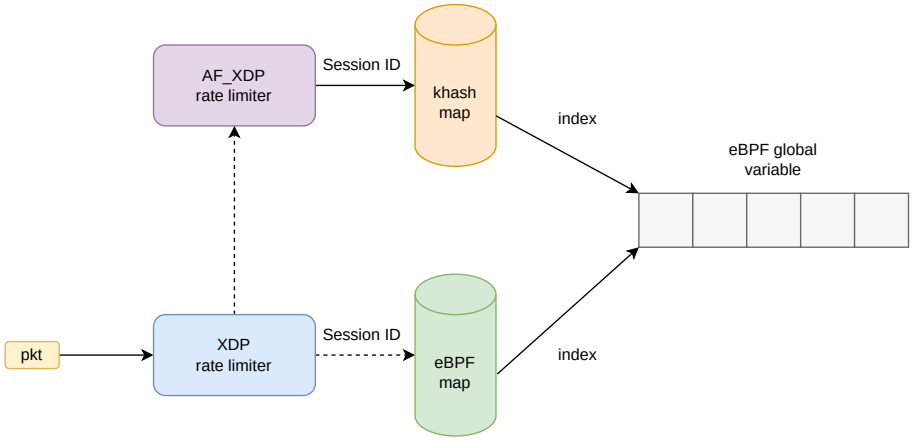

Figure 4.5: Final solution for hash collisions.

application, the XDP side. This creates some potential race conditions which should be properly handled.

In XDP, since it is an eBPF program, the prototype could rely upon *eBPF spinlocks* [18], which allows to synchronize access within eBPF maps. However, this is not is not the best choice for AF_XDP applications since it would require to interact with maps and, as already seen, this is something that must be avoided to save high performance drops.

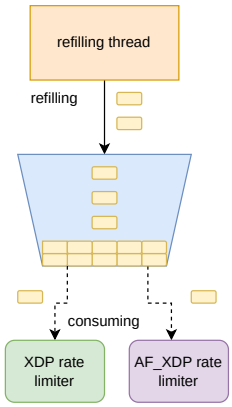

Figure 4.6: Refilling thread.

The final solution has been relying on the use of the eBPF global variable already mentioned and, all the access to it both by the AF_XDP program and XDP one as well as by the refilling thread has been managed with the only use of atomic operations which have been tested to be working even when threads belonging to user and kernel level try to access concurrently the same data.

```
void *refill_thread(void *args){
    ...
    while(1){
        for(i=0; i < nrules; i++){
                position = khashmap_lookup_elem(&positions, &entries[
    i].key);
                amount = skeleton->bss->contracts[*position].bucket.
    refill_rate;

                clock_gettime(CLOCK_MONOTONIC_RAW, &time);

                now = time.tv_sec * 1000000 + time.tv_nsec/1000;

                /* time elapsed used to weight the refilling rate */
                difference = (now - skeleton->bss->contracts[*
    position].bucket.last_refill);

                amount = difference*(skeleton->bss->contracts[*
    position].bucket.refill_rate/1000);

            ...
        usleep(1000);
    }
}
```

Listing 4.15: The refilling thread

This refilling task has been implemented to be as much precise as possible. However, some imprecision have been experienced due to the behaviour of the *usleep()* function which allows to specify how many microseconds a thread should sleep for, unlike the *sleep()* function which is limited up to seconds precision. This happens because the ulseep function can go to sleep for a grater number of microseconds than the specified one. For patching this imprecision, the elapsed time in tems of microseconds is computed among different refilling phases so that whenever the refill should be done it will be weighted to this difference. The refilling rate is set to be working every milliseconds which corresponds to 1000 microseconds.

Chapter 5

Evaluation

This chapter analyses a set of tests which have been carried out to evaluate both the performance and the precision of the proposed solution.

First of all, both the test environment and the set of tools which have been used are introduced. Then, the results obtained from the different test scenarios are evaluated.

5.1 Testbed Setup

Tests have been carried out on two physical machines which are connected through a direct link, using Intel X540-AT2 10-Gbps NICs.

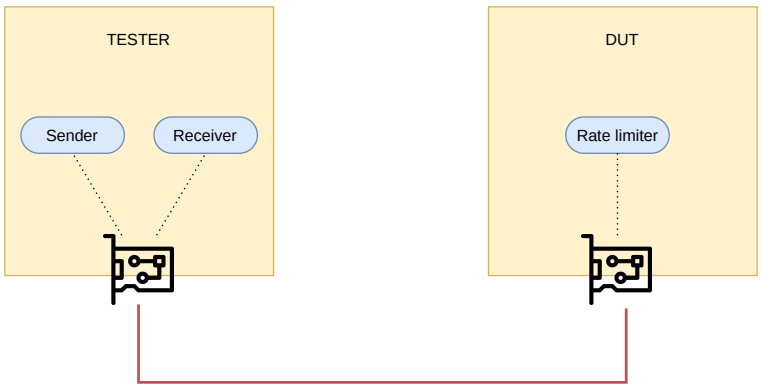

Figure 5.1: General Configuration.

One machine operates as DUT (Device Under Test), executing the prototype under test which receives and forward back the traffic from its interface, whereas the other acts as tester, generating test packets and measuring the processed traffic from its own interface.

5.1.1 DUT Characteristics

The machine which executes the prototype under evaluation has the following characteristics:

- Processor: Intel Core i7-4770 @ 8x 3.9GHz
- Memory: 32 GiB of DRAM
- Operating System: Ubuntu 20.04 focal
- Kernel version: 5.16.9

5.1.2 Tester Characteristics

The machine which sends and receives back the traffic has the following characteristics:

- Processor: Intel Core i7-3770 @ 8x 3.9GHz
- Memory: 16 GiB of DRAM
- Operating System: Ubuntu 20.04 focal
- Kernel version:5.4.0-121-generic

5.2 Tools

5.2.1 MoonGen

MoonGen [19] is a scriptable high-speed packet generator built on libmoon. The whole load generator is controlled by a Lua script: all packets that are sent are crafted by a user-provided script. Thanks to the incredibly fast LuaJIT VM and the packet processing library DPDK, it can saturate a 10 Gbit/s Ethernet link with 64 Byte packets while using only a single CPU core. MoonGen can achieve this rate even if each packet is modified by a Lua script. It does not rely on tricks like replaying the same buffer.

MoonGen focuses on four main points:

- High performance and multi-core scaling: $>$ 20 million packets per second per CPU core
- Flexibility: Each packet is crafted in real time by a user-provided Lua script
- Precise and accurate timestamping: Timestamping with sub-microsecond precision on commodity hardware

- Precise and accurate rate control: Reliable generation of arbitrary traffic patterns on commodity hardware

MoonGen architecture is depicted in figure 5.2, where it shows that is built on libmoon, a Lua wrapper for DPDK. Users can write custom scripts for their experiments. It is recommended to make use of hard-coded setup-specific constants in your scripts. The script is the configuration, it is beside the point to write a complicated configuration interface for a script. Alternatively, there is a simplified (and less powerful) command-line interface available for quick tests.

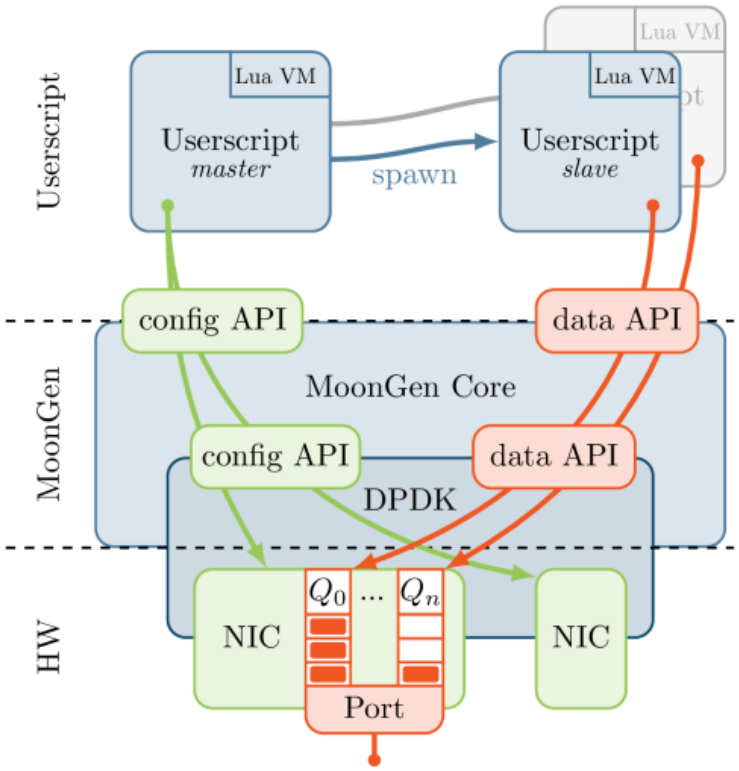

Figure 5.2: MoonGen architecture.

Execution begins in the master task that must be defined in the userscript. This task configures queues and filters on the used NICs and then starts one or more slave tasks.

Lua does not have any native support for multi-threading. MoonGen therefore starts a new and completely independent LuaJIT VM for each thread. The new VMs receive serialized arguments: the function to execute and arguments like the queue to send packets from. Threads only share state through the underlying library.

5.2.2 iperf3

iperf3 [20] is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, protocols, and buffers. For each test it reports the measured throughput as bit-rate, loss, and other parameters.

5.3 Performance tests

This set of tests evaluates the performance of the Rate Limiter under different circumstances and it also analyses its scalability when distributing the traffic over different instances. Preliminary tests showed that packet size can impact the overall performance, where the system is stressed more when dealing with small packets.

As a consequence following benchmarks has been performed using minimum sized packets to see how the solution behaves when it is under pressure. Both in the down-link and in the up-link direction, the size would be 64 bytes.

In those performance tests, UDP traffic has been generated with MoonGen without limiting the rate. In this way, the prototype can forward back traffic with the highest throughput.

The Rate Limiter prototype performance has been evaluated and compared at different levels: XDP, AF_XDP and hybrid. For the sake of completeness, the comparison is done also at the TC level where a custom eBPF program, with the support of the appropriate qdisc rule, is attached.

The custom TC eBPF program

The following code shows the custom and simple eBPF program that is attached to the NIC when evaluating the TC throughput.

```
#include <arpa/inet.h>
#include <linux/bpf.h>
#include <linux/pkt_cls.h>
#include <bpf/bpf_helpers.h>
SEC("ingress")
int tc_ingress(struct __sk_buff *skb){
        bpf_clone_redirect(skb, skb->ifindex, 0 );
        return TC_ACT_REDIRECT;
}
```

Listing 5.1: A simple eBPF program to send back all incoming traffic at the TC level

Then, a *Token Bucket Filter (tbf)* queuing discipline of the kernel is used as a reference:

```
tc qdisc add dev eth0 root tbf rate 100mbit burst 10mbit latency 400
    ms
```

Listing 5.2: TC command to apply bandwidth limit with Token Bucket Filter

MoonGen is able to generate a traffic rate of 14.88 Mpps on the hardware platform previously introduced, which can be seen as up-link traffic. In the figure 5.3 is reported the down-link traffic, to evaluate and compare the maximum rate that can be achieved with the different solutions.

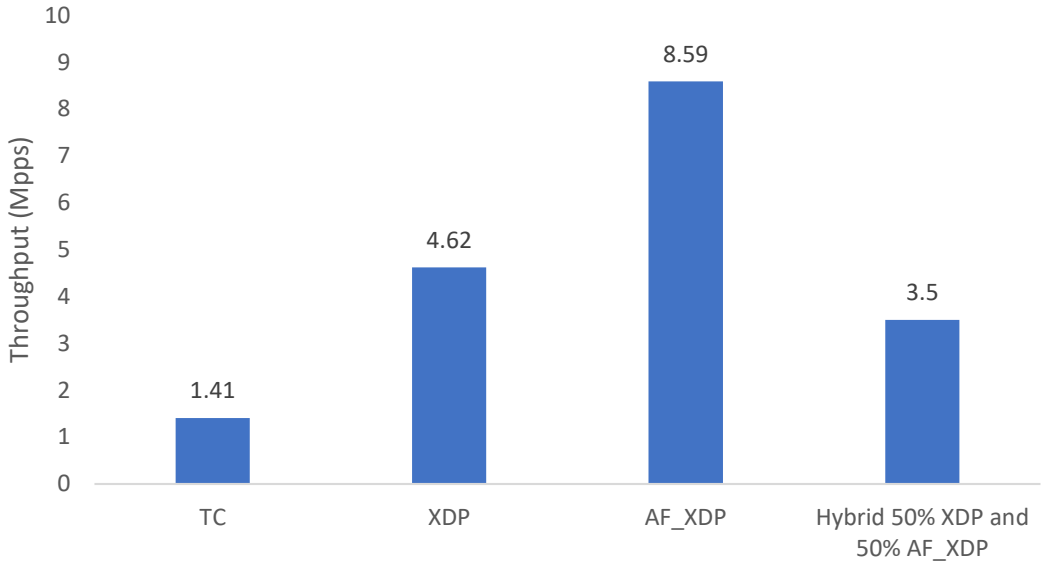

Figure 5.3: Performance test results.

Results show that TC gets the worse performance in the evaluation, due to its execution at a higher level of the network stack.

XDP achieves higher performance thanks to the possibility of deploying the associated eBPF program directly to the driver of the NIC. This high performance improvement is motivated by the fact that the NIC of the host machine has a direct support for XDP Native [21].

The situation drastically changes when the traffic is completely managed by AF_XDP. Here, the best performance are achieved and this is due to the capability of the technology to bypass entirely the Linux kernel.

As a result, the traffic does not undergo to a series of kernel tasks which slow down the overall process. Moreover, the kernel no longer allocates data structures that could be useless in many cases.

Then, the prototype has also been evaluated with a *hybrid* solution, where the rate limiter is run both with XDP and AF_XDP.

A first result shows that this configuration has an inferior throughput compared to the solutions running either XDP or AF_XDP. Here, the traffic is equally distributed among the two technologies. This performance drop is mainly caused by concurrent access by the two parts of the solution as well as by a copy semantic applied to the traffic when attaching an AF_XDP socket to an eBPF XDP program.

However, this combination is still meaningful if the traffic is partitioned differently. As the chart in Figure 5.3 shows, the *hybrid* solution achieve higher performance when most of the traffic is handled in AF_XDP and only a small portion in XDP. Indeed, results show that if 10% of the packets are managed in XDP whereas the remaining 90% portion in AF_XDP, the performance are almost doubled with respect to the ones achieved with the XDP solution.

5.3.1 Multiple flows scalability

In this tests, the number of flows handled has been ranged from 1 to 100 to see how performance can vary. This variation is propagated both on the tester side, where a different number of flows is generated, and on the DUT side, where a different number of policies is applied on the upcoming traffic.

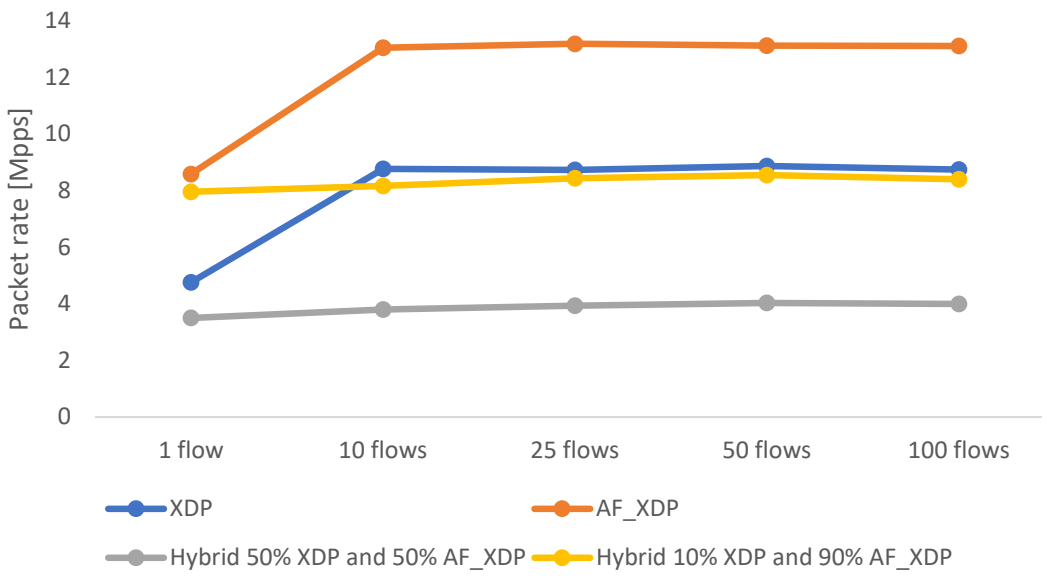

Figure 5.4: Scalability performance.

Results in Figure 5.4 shows that performance in XDP and AF_XDP have a significant increase when passing from one flow to multiple flows. This is due to the allocation of multiple queues on the NIC where thanks to the support of *RSS (Receive Side Scaling)*, the traffic will be fairly distributed among the CPU cores.

On the other side, when running the model as a hybrid solution, the throughput improvements is slighter than the previous cases when passing from single to multiple flows. This happens because there will always be two working CPU cores, one for handling the interrupts and the other one for running the AF_XDP application. However, it is interesting to notice that the performance improve remarkably if AF_XDP side handles more packets when running the model as a hybrid solution. Finally, the performance growth starts to drop as the flows to be managed increase in almost all solutions.

5.3.2 AF_XDP: Interrupt-driven vs Busy Polling

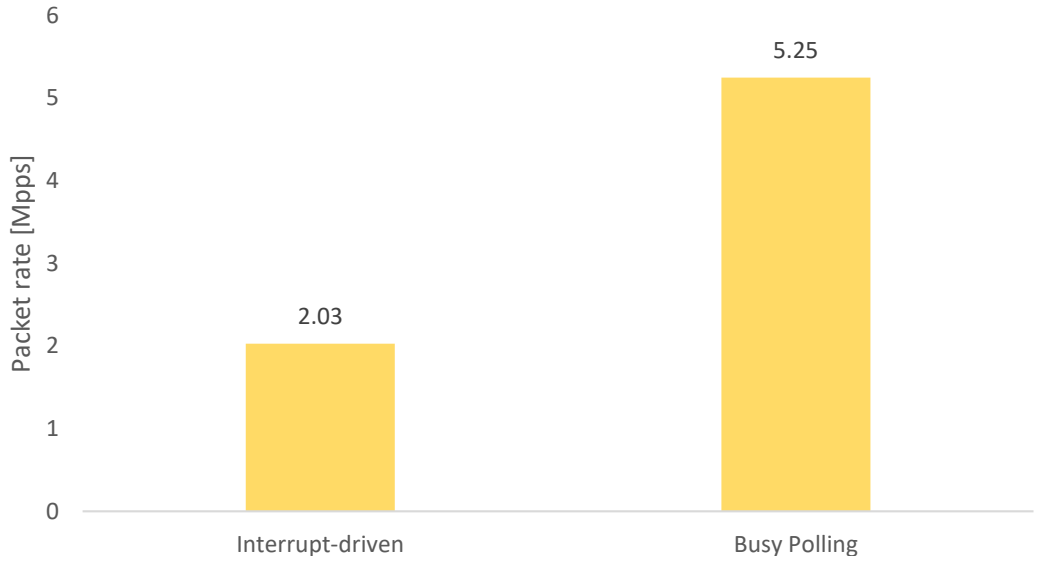

Figure 5.5: AF_XDP: interrupt-driven vs busy polling.

Figure 5.5 shows a single-core test comparison between two AF_XDP modalities. The *interrupt-driven* mode is the traditional notification of a system when an event should be notified though an interrupt service routine. In this case, this routine is forwarding the packets to the AF_XDP applications. However, the presence of interrupt handling could affect remarkably the performance. For this reason, AF_XDP offers the possibility to get rid of interrupts by performing busy-waiting packets lookup. This mode is called *Busy Polling*. However, this execution mode is meaningful only when AF_XDP is exclusively in charge of the packet management. Becouse of the event-driven nature of eBPF, if disabling interrupts on the NIC the execution of the XDP application becomes useless.

Furthermore, when running applications in busy-waiting, they usually end up in consuming more CPU cycles. This, with the prospective of balanced resource

management in Cloud environments, could not be feasible in several cases.

5.4 Precision tests

In the precision test, the evaluation of the Rate Limiter prototype is done both with UDP and TCP traffic. *MoonGen* has been used for UDP data, whereas TCP traffic is generated with *iperf3* tool. In both cases, the packets are generated with the smallest size of 64 Bytes. Moreover, the token bucket algorithms is set with the same parameter under all the test cases, where the burst-size is about 1/10 of the refill rate.

In this test-bed, traffic is equally distributed among XDP and AF_XDP when evaluating the hybrid model.

5.4.1 UDP tests

In this evaluation, the prototype will fistly be set with an increasing rate limit, from 100 Mbit/s to 700 Mbit/s.

This range has been choosen to compare the performance of the solution with a token bucket rate limiter done with the TC Linux tool.

Afterwards, an evaluation with higher bit-rates is considered only within the implemented prototype going form 1Gbit/s to 3 Gbit/s.

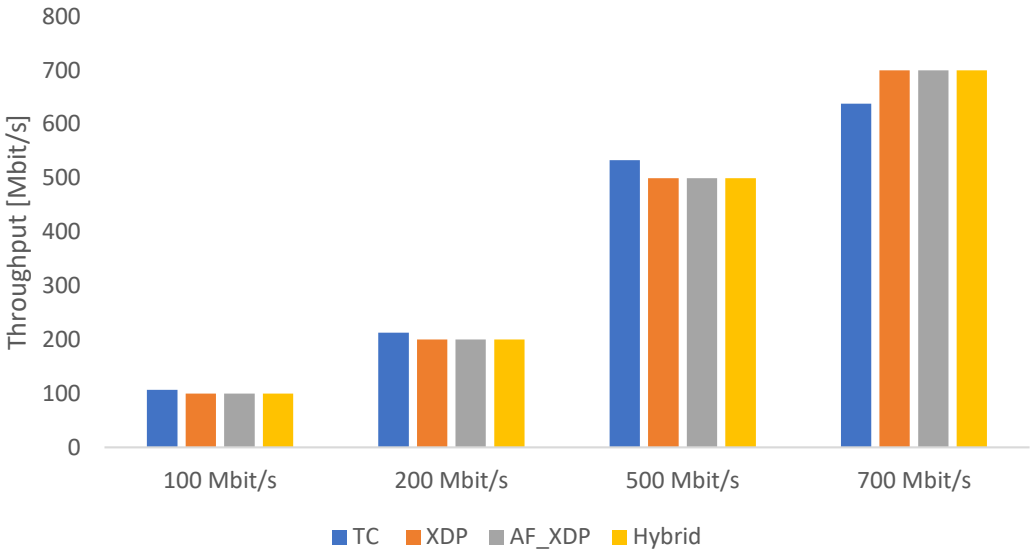

Figure 5.6: UDP precision test.

In Figure 5.6, results show that the prototype achieves an accurate rate whereas

TC is less precise. Particularly, TC gets a throughput which is higher than the expected one in the 100 Mbit/s 200 Mbit/s and 500 Mbit/s.

This might be due to the presence of intermediate buffers which could hold temporarily data, transmit them later possibly exceeding the expected rate.

On the other hand, when testing a token bucket at the TC level, the throughput obtained is smaller than what expected. This is very likely to be thanks to the major complexity added by the network function, which usually turns to decrease the maximum rate.

Instead, for XDP, AF_XDP and hybrid, the throughput obtained is the one that is expected under all the test cases.

UDP at highest rates

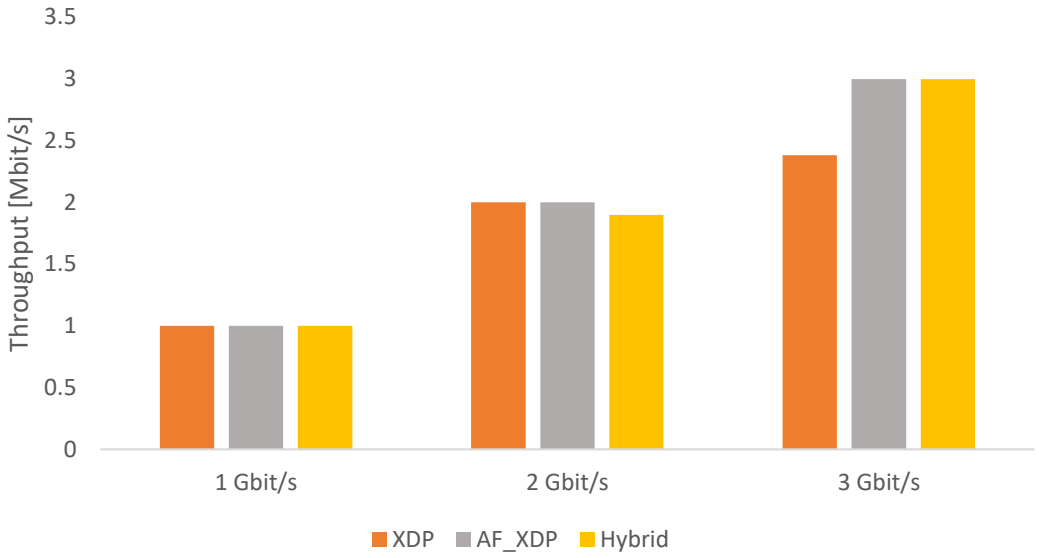

Figure 5.7: UDP precision test at higher rates.

In Figure 5.7, a test is done only with the proposed solution evaluated only with XDP, AF_XDP and hybrid model to analyse how it performs with higher rates.

The rate limiter is evaluated with a thoughput of 1 Gbit/s, 2 Gibt/s and 3 Gbit/s. Results show that both technologies, XDP and AF_XDP, perform quite well under all the cases. The only exception is done by XDP in the 3 Gbit/s, which could reach only about 2.4 Gbit/s. This is due to the technology limitation of XDP, as it could not achieve a higher rate. On the other side, the hybrid model is still rather precise but it loses performance as the rate increases.

This behaviour has been already presented in the performance test.

5.4.2 TCP traffic

Also here, tests are firstly carried out by increasing the rate limit from 100 Mbit/s to 700 Mbit/s to be able to evaluate also the performance at TC level.

Again, there will be an evaluation only with XDP, AF_XDP and hybrid model with higher rates.

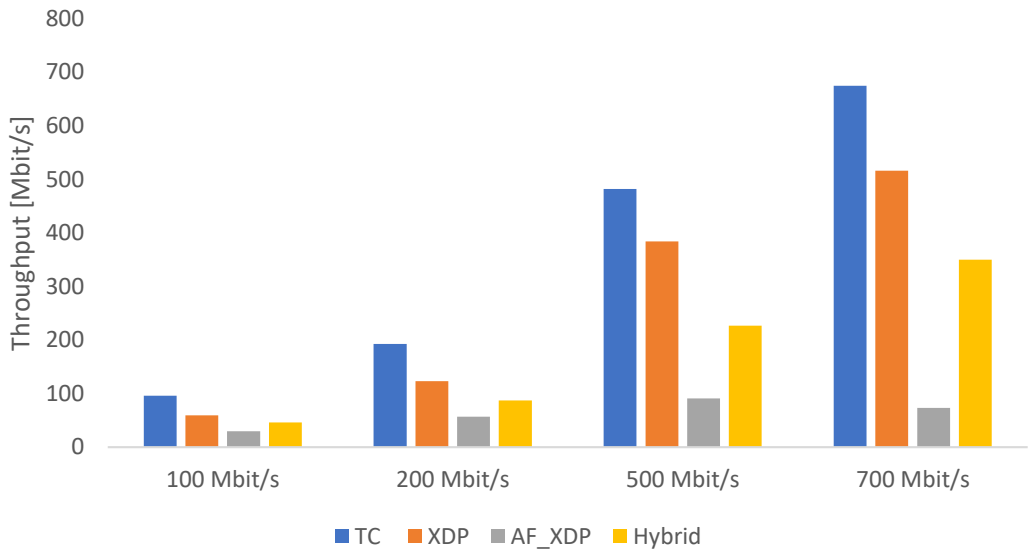

Figure 5.8: TCP precision test with buffer-less solution.

In Figure 5.8 shows that the Token Bucket is not able to produce the desired rate, this happens because it is configured with a burst limit smaller than the desired rate, 1/10 of the desired rate.

The Token Bucket Filter (tbf) queuing discipline performs quite well as it gets close to the desired rate in all cases. This happens because it has been designed keeping into account that TCP protocols relies upon the presence of big intermediate buffers.

The low performance given by the solution is mainly motivated by its buffer-less design, which heavily affect the correct behaviour of TCP traffic.

To emulate the buffering mechanisms needed by TPC, the burst size can be increased up to the rate desired in the prototype.

Results in Figure 5.9 show that the solution reach the expected rate. This workaround adopts the buffer-oriented needs of TCP with the buffer-less nature of the prototype.

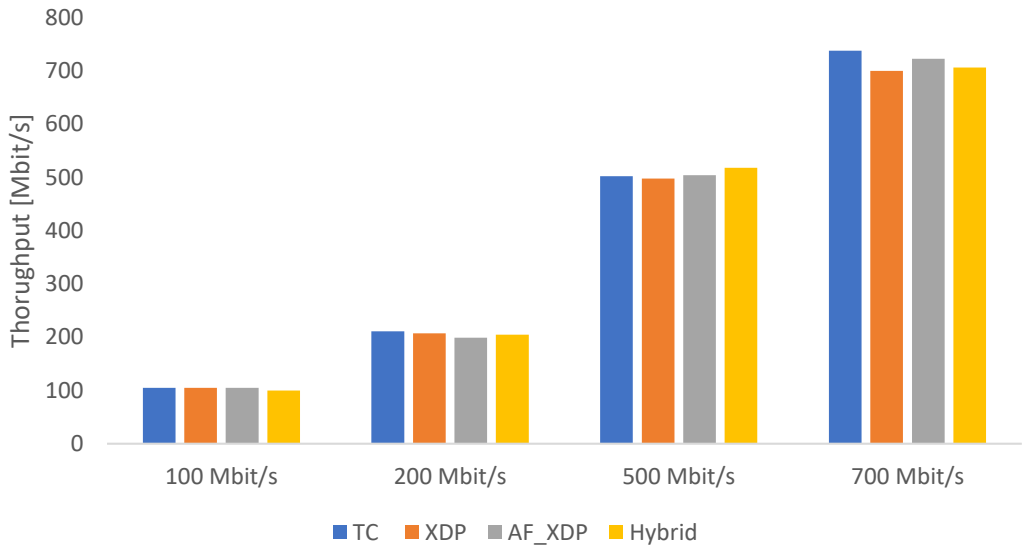

Figure 5.9: TCP precision test with buffered solution.

TCP at highest rates buffer-less

Results in Figure 5.10 shows how the model reacts when dealing with higher rate traffic. The performance downsize is still due to the buffer-less nature of the solution.

TCP at highest rates buffer

As shown in Figure 5.11, the performance improves when bringing the burst size up to the refill rate.

Among the solutions, AF_XDP results to be the least precise one. In some cases, it could achieve a higher rate compared to the desired one, while in other cases it is not able to reach the one expected.

Yet, this imprecision is mainly due to the emulation of the buffering approach by increasing the burst size of the rate limiter.

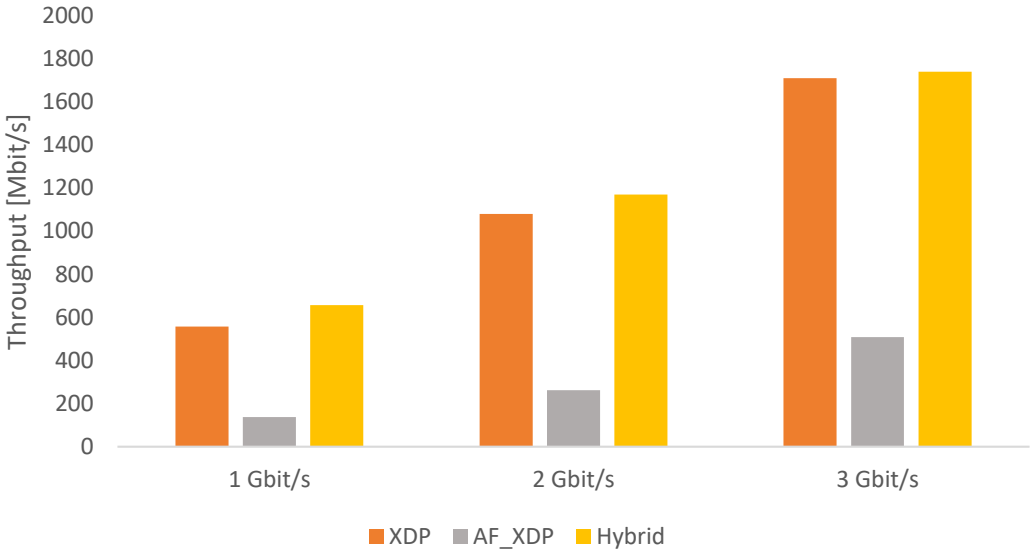

Figure 5.10: TCP precision test at higher rates with buffer-less solution.

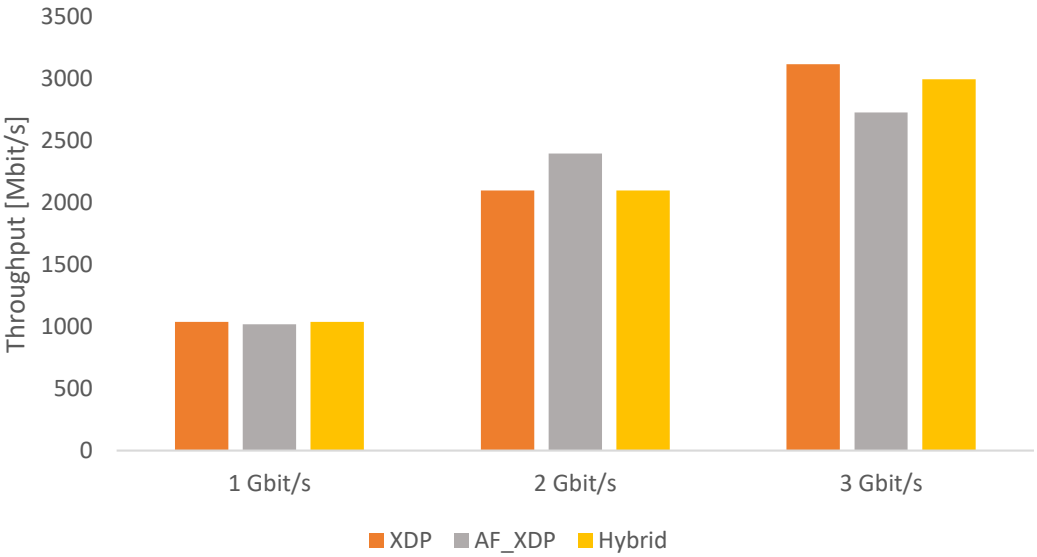

Figure 5.11: TCP precision test at higher rates with buffered solution.

Chapter 6

Conclusions

This thesis aimed in building a prototype of a hybrid Rate Limiter network function which could run concurrently in kernel space, thanks to the eBPF/XDP framework, and in user space with high-performance AF_XDP sockets, pointing out how the cooperation of these two technologies could be an interesting solution for high demand telcos oriented data plane functions, becoming very important in modern architectures like the 5G.

The main objective of this study was to explore if AF_XDP could overcome some limitations of the eBPF and how those two technologies could be combined.

In the implementation phase, many challenges have been faced mainly due to the lack of synchronization primitives. This would have allowed to synchronize kernel and user tasks when accessing the same data, as well as providing the possibility to access shared data without resulting in huge performance drops.

Several solutions and workarounds have been found to overcome these problems, such as relying upon atomic instructions and building a shared state without the potential overhead. To adopt the last solution, the BPF skeleton structure was extremely useful and important.

This allows a user-space application to manage eBPF data without issuing any system call, which ends up in a faster exchange of data since the access is direct.

The evaluation of the prototype shows that the hybrid Rate Limiter could achieve higher throughput compared to the Token Bucket qdisc kernel discipline which could be set in Linux with TC tool

The Token Bucket qdisc kernel discipline has been mainly taken as a comparison reference.

The model has been evaluated under different levels:

- XDP
- AF_XDP
- Hybrid: traffic is split between XDP and AF_XDP

The best performance is achieved by AF_XDP, thanks to its capability of bypassing the Linux kernel.

However, when scaling out to different instances, the hybrid model is the only one which improves the results.

In the precision test, results have shown that the prototype has rather high accuracy when dealing with UDP traffic.

There has been a first comparison with the /Token Bucket Filter] queuing discipline of the kernel, where the proposed solution, under the different models, seemed to be more precise when the limit was increased.

Afterwards, UDP traffic tests have been carried out comparing only the models of the solution, setting up higher rates. Here, all the models are precise but the XDP one starts to decrease its rate at a certain point due to its performance limitations.

TCP tests highlighted the difficulties that the solution can find as being completely bufferless. In fact, the solution could not reach the desired rate due to the absence of intermediate buffers.

Here, to overcome this issue, the burst size of the token bucket has been increased up to the target rate to emulate a buffer-oriented model.

6.1 Possible Improvements

This model could be further developed by adopting a Traffic Shaping technique, which could be useful for many purposes.

Although this is not possible in eBPF due to its restricted environment, it is still possible in user-space as AF_XDP take traffic out of the kernel. Indeed, eBPF could cooperate with AF_XDP to circumvent its run-time limitations.

Moreover, a deeper analysis of how to integrate the two technologies would be interesting. This is so that different types of traffic could be managed by one of the two depending on their profile. A classifier can be used in this situation to understand how traffic should be managed.

Lastly, a further investigation may be done in order to analyse how it is possible to combine this solution with other modules. This would be very interesting in real-world scenarios, where simple single workloads are usually combined to build a more complex service chain.

6.2 Potential challenges and future directions

Cloud computing has changed the way applications are built and deployed thanks to the flexibility and agility provided.

Following this approach, services are usually deployed and run on general-purpose servers. Here, it is no longer necessary to have a physical machine with a given set of characteristics. This is thanks to the possibility of creating dynamically a virtual server with the requested characteristics based on the actual requirements. As a consequence, users can buy tons of equivalent servers, with exactly the same hardware characteristics, and aggregate them together in a data center. Computing hardware becomes a commodity, thus COTS (Commercial Off-the-Shelf) is requested. For this reason, already available facilities could be useful to build new services. If network functions are considered, the Linux Kernel would be the component which could be used. It has a solid implementation, support for a variety of protocols and network devices, well-defined API, efficient resource consumption and efficient sharing of resources.

On the other side, it comes with many drawbacks. Among the most considerable ones, there are:

- Heavy-weight
- It may introduce unnecessary overheads
- Difficult to customize
- It may slows down innovation

Those issues encouraged the possibility of finding other solutions. Here, Kernel-Bypass Networking came across.

One of the most common technologies to overcome the kernel overhead is DPDK. It is a rather powerful framework since it provides high-performance and it is also easy to customize and as such it supports innovation.

However, DPDK adopts an inefficient resource consumption as it relies upon the busy polling technique, taking exclusive access to a CPU core. As a consequence, it causes an energy consumption disproportion. Furthermore, it has a scarce system integration as it requires a complete set of custom drivers and possibly appropriate NICs. Then, it results to be difficult to use when sharing resources is needed, as it assumes full control of the network devices to which they are attached. Last but not least, all kernel security and isolation features are completely bypassed.

With AF_XDP, the actual application APIs interacting with sending and receiving packets are decoupled from the low-level infrastructure. Moreover, the fact that XDP and AF_XDP work with any driver, as they can be injected along different layers of the networking stack, makes the technology really portable and flexible. Therefore, it is compliant with the needs of cloud environments.

So, in this way traffic can land on any platform and in any Cloud provider taking advantage of the same APIs. The performances, as has been shown previously, are really promising and it is very interesting how the technology has been growing over the last few years. This is why it is considered an excellent fit for cloud-native networking.

On the other hand, there are still some challenges in deploying micro-service applications based on AF_XDP. Among those, there is the NICs sharing and the efficient use of network devices.

The idea is that AF_XDP applications could share A NIC taking different queues to meet a good level of resource management and scalability which are typical of cloud-native applications.

However, if those applications are considered to be wrapped into containers, this sharing becomes difficult due to the need of moving the network device into the container network namespaces, thus making network devices and queues inaccessible outside the containers.

With this approach, each container consumes a full network device, which turns out to be not a scalable solution. The target here is to find a way of partitioning a network device.

The Linux Kernel Devlink API [22] for subfunction management is a solution that has been proposed by the community. This is a new light-weight PCI function, where a physical function is sliced into netdev-queue pairs. This allows allocating a much more granular portion of A NIC to containers, receiving a netdev-queue pair rather than the full network device. Those APIs would facilitate all the necessary resource management that would be needed in those contexts.

Another aspect to be considered when using AF_XDP is the needed privileges. In fact, up to 5.9 kernel version, root privileges are required to create AF_XDP sockets. Furthermore, Ethtool net filter rules, allowing to query or control network driver and hardware settings, also require privileges as for redirecting packets to sockets in this case.

The solutions to those issues have been the separation of AF_XDP socket from BPF program loading in 5.9 Linux kernel version and leaving the programming of Ethool net filter rules to a CNI which runs outside the containers, following the least privilege property typical of cloud-native applications.

Other issues to be addressed are that Ethtool rules are persistent across network namespaces. Here, traffic could still be forwarded to containers that are no longer up. That is why, a well-designed CNI would play an important role in this scenario, setting and cleaning up all those rules upon run-time needs.

Moreover, interface indexes are not unique across different network namespaces and this complicates the loading and unloading of eBPF programs. An appropriate and consistent naming of AF_XDP network devices across different namespaces is important to be able to retrieve the right interface index.

6.3 Final considerations

AF_XDP opens the possibility of building up packet processing applications as containers, considering aspects of performance along with portability and scalability.

On the other hand, eBPF remains an advantageous technology from an infrastructure platform perspective since it provides monitoring facilities that would be essential in system administration.

eBPF has been an essential playground for innovation in the last few years. But, upstreaming the kernel to apply modifications is a rather long task. This could slow down the innovation process when trying to take advantage of the flexibility and scalability of this technology.

From the application side, it is preferred to stay at the user-land level with an AF_XDP socket attached to it. AF_XDP is at the beginning of its journey and as such, there is still room for improvements, growth and innovation. It is migrating out the kernel, with plans to wrap it up into a library [23].

As a conclusion, a combination of both can end up in the best approach as eBPF would manage the low-level part of applications whereas AF_XDP would consume the packets directly out of the in-kernel fast-path with all the possible complexity needed.

Bibliography

[1] Linux manual page. *socket syscall*. `https://man7.org/linux/man-pages/man2/socket.2.html`, Last accessed on 2022-05-16 (cit. on p. 10).

[2] Livio Soares and Michael Stumm. «{FlexSC}: Flexible System Call Scheduling with {Exception-Less} System Calls». In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010 (cit. on p. 11).

[3] Intel. *The DPDK Library*. `http://doc.dpdk.org/guides/index.html`, (cit. on pp. 12, 23).

[4] Luigi Rizzo and Matteo Landi. «Netmap: Memory mapped access to network devices». In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 422–423 (cit. on p. 12).

[5] The Cilium Authors. *BPF and XDP Reference Guide*. `https://docs.cilium.io/en/stable/bpf/`, Last accessed on 2022-05-16 (cit. on p. 13).

[6] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. «The express data path: Fast programmable packet processing in the operating system kernel». In: *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 2018, pp. 54–66 (cit. on p. 18).

[7] The Linux manual page. *BPF system call*. `https://man7.org/linux/man-pages/man2/bpf.2.html`, Last accessed on 2022-05-16 (cit. on p. 20).

[8] The Linux Kernel. *AF_XDP Documentation*. `https://www.kernel.org/doc/html/latest/networking/af_xdp.html`, Last accessed on 2022-05-16 (cit. on p. 20).

[9] The eBPF Documentation. *libbpf library*. `https://www.kernel.org/doc/html/latest/bpf/libbpf/index.html`, Last accessed on 2022-05-16 (cit. on p. 22).

[10] The Linux Kernel. *tc*. `https://man7.org/linux/man-pages/man8/tc.8.html`, (cit. on p. 25).

[11] Federico Parola, Fulvio Risso, and Sebastiano Miano. «Providing telco-oriented network services with eBPF: the case for a 5G mobile gateway». In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE. 2021, pp. 221–225 (cit. on p. 25).

[12] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. «A Framework for eBPF-Based Network Functions in an Era of Microservices». In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 133–151. DOI: 10.1109/TNSM.2021.3055676 (cit. on p. 26).

[13] Wallmart. *l3AF*. https://l3af.io/, Last accessed on 2022-05-16 (cit. on p. 26).

[14] Wallmart. *l3AF Documentation*. https://medium.com/walmartglobaltech/introducing-walmarts-l3af-project-xdp-based-packet-processing-at-scale-81a13ff49572 (cit. on p. 26).

[15] The OVS authors. *OVS Rate Limiter*. https://docs.openvswitch.org/en/latest/howto/qos/ (cit. on p. 26).

[16] Andrii Nakryiko. *BCC to libbpf conversion guide*. https://nakryiko.com/posts/bcc-to-libbpf-howto-guide/#bpf-skeleton-and-bpf-app-lifecycle, (cit. on p. 40).

[17] Wikipedia. *The Cuckoo Hashing algorithm*. https://en.wikipedia.org/wiki/Cuckoo_hashing, (cit. on p. 45).

[18] The Linux Kernel authors. *BPF Spinlocks*. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=d83525ca62cf8ebe3271d14c36fb900c294274a2, (cit. on p. 47).

[19] *The MoonGen Packet Generator*. https://github.com/emmericp/MoonGen, (cit. on p. 50).

[20] iperf authors. *iperf3*. https://github.com/esnet/iperf, (cit. on p. 52).

[21] *Pantheon.tech*. https://pantheon.tech/what-is-af_xdp/, (cit. on p. 53).

[22] The Linux Kernel authors. *The Linux Kernel Devlink APIs*. https://www.kernel.org/doc/html/v5.12/networking/devlink/index.html, (cit. on p. 64).

[23] XDPtool. *xdp tools*. https://github.com/xdp-project/xdp-tools, (cit. on p. 65).