POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Prototyping an eBPF-based 5G Mobile Gateway

Supervisors
Prof. Fulvio Risso
Ing. Sebastiano Miano, PhD

Candidate
Federico Parola

Academic year 2019-2020

Abstract

In the upcoming 5G network infrastructure, the virtualization of network function is a key component to meet the flexibility and scalability requirements imposed by the wide range of supported applications. The Mobile Gateway is the component of the 5G core network responsible of interconnecting the mobile user equipment located in the Radio Access Network to one or more Packet Data Networks (e.g. the Internet). This work studies advantages and challenges of using eBPF (Extended Berkeley Packet Filter), a novel technology that allows fast packet processing in the Linux kernel, to implement such a network function. A modular prototype providing a subset of the functionalities of the Gateway is proposed, and its performance is compared to equivalent network functions based on alternative data plane technologies. Results show that eBPF, thanks to its flexibility and integration with the Linux kernel, can be an interesting solution in scenarios such as small, distributed data centers for edge computing.

Contents

Chapter 1

Introduction

The upcoming fifth generation mobile network will bring a huge improvement in system capacity, access speed, latency and energy efficiency in order to face the ever growing requirements of an increasing number of services. 5G will support a wide variety of use cases [1], each one with specific requirements, that can be subdivided in three broad categories:

- Enhanced mobile broadband (eMBB): The evolution of the 4G broadband providing faster data rates, a more widespread connectivity and increased mobility, in order to enable advanced services like 360°high-resolution video streaming, immersive AR and VR applications and real time monitoring.
- Ultra-reliable and low latency communications (URLLC): A class of services with stringent requirements for reliability, latency and availability for mission critical applications such as industrial internet, remote surgery and autonomous driving.
- Massive machine type communications (mMTC): A family of applications providing connectivity to a huge number of devices (usually IoT devices) that typically produce sporadic small amounts of data, such as grid of sensors or smart home systems.

To support this range of use cases the 5G architecture has been completely redesigned, including concepts like Mobile Edge Computing (MEC), which provides cloud computing capabilities in close proximity to users, reducing the overhead on the core network and allowing the user equipment to offload complex tasks without latency penalties, and Network Slicing, allowing to instantiate overlay networks on a shared infrastructure in order to satisfy the needs of different vertical industries.

In this scenario the traditional mobile core architecture, based on hardware, fixed-function middleboxes, lacks in flexibility, scalability and programmability. As a consequence Network Function Virtualization (NFV) and Software Defined

Networks (SDN) has been identified as key technology enablers for realizing 5G networks [2]. Network Function Virtualisation allows to run network functions as software application on top of container or virtual machines executed on Commodity Off-the-Shelf (COTS) hardware. This enables the use of cloud computing practices to improve flexibility and scalability while reducing both capital (CAPEX) and operational (OPEX) expenditures for the service provider. Software Defined Networking complements this technology, providing the ability to programmatically reconfigure the network in order to achieve traffic engineering and steering.

1.1 Goal of the thesis

Despite its many advantages, this softwarized approach highlights the performance limits of general purpose hardware with respect to dedicated appliances. This limits are especially visible in the software data plane that, due to the demanding requirements of the 5G network, has to process packets at a very high rate. As a consequence technologies for high speed packet processing are required, like kernel bypass solutions such as Intel DPDK.

This thesis studies the possibility to use eBPF (Extended Berkeley Packet Filter), a novel technology that allows to run fast network functions in the Linux kernel, to implement a 5G Mobile Gateway, the component responsible of handling the User Plane Function in the Mobile Packet Core. A prototype is proposed and evaluated, highlighting the challenges posed by the technology and the advantages and drawbacks of the solution.

Chapter 2

Background

This chapter provides a description of the main elements and key technologies this thesis is based on.

First, the 5G Mobile Gateway is presented, highlighting its main features and the function it carries out inside the 5G Mobile Core network. Then, a detailed description of the two main technologies used in the project, eBPF and Polycube, is provided. In the end, an overview of related work studying the virtualization of the Mobile Gateway function is proposed.

2.1 5G Mobile Gateway

The 5G Mobile Packet Core (MPC) [3] is the network responsible of providing voice and data services to mobile user equipment. These devices are connected to the network through radio base stations inside the so called Radio Access Network (RAN). The 5G Core network is an evolution of the 4G Evolved Packet Core (EPC), it heavily relies on Network Function Virtualization (NFV) and Software Defined Networks (SDN) and is based on two key principles:

- Control and User Plane Separation (CUPS), that enables the User Plane Function to be deployed independently from the centralized control plane, in closer proximity to the user, increasing flexibility and scalability of the system.
- Service Oriented Architecture (SOA), that allows a flexible and extensible control plane based on independent and interchangeable services that communicated through HTTP based RESTful APIs.

Some of the main modules composing the control plane of the Mobile Packet Core are:

- The Access and Mobility Management Function (AMF), that handles all signaling messages needed to interconnect the RAN to the Mobile Core and is

responsible of authenticating the user equipment and managing its mobility between different base stations.

- The Session Management Function (SMF), that assigns and manages IP addresses of the user equipment, interacts with the RAN through the AMF to send QoS and policy information, identifies the User Plane Function that is more suited to serve a certain user and configures it with all parameters needed to handle the corresponding session.
- The Policy Control Function (PCF), that provides policy rules to other control plane functions, including policies related to QoS, network slicing, charging and mobility management.

All functions related to the user plane (the User Plane Function) are grouped into the Mobile Gateway, also known as Evolved Packet Gateway, that constitutes the point of interconnect between the mobile user equipment and external Packet Data Networks (PDN) such as the Internet or a corporate intranet. Multiple gateways can be used by a single user device, also in cascade, to provide connectivity to different or the same network. All the traffic that flows between a user device and the Data Network is grouped in a PDU Session, composed by one or more bi-directional radio bearers that connect the user to the Base Station and two unidirectional GTP-U tunnels that connect the Base Station to the Mobile Gateway. The GPRS Tunneling Protocol user plane (GTP-U) carries traffic in this last part of the PDU Session and is based on the encapsulation shown in figure 2.1:

Figure 2.1: GTP encapsulation.

- The external IP header contains the IP addresses of the Base Station and the Mobile Gateway.
- UDP is used as transport layer, with destination port 2152.
- The GTP-U header contains, among other information, the Tunnel Endpoint IDentifier (TEID), that uniquely identifies the uplink tunnel (BS to GW) on the Gateway and the downlink tunnel (GW to BS) on the Base Station.
- The inner IP header contains the IP addresses of the mobile user equipment and the remote destination on the Packet Data Network.

A simplified representation of the 5G Mobile Packet Core Architecture is shown in figure 2.2.

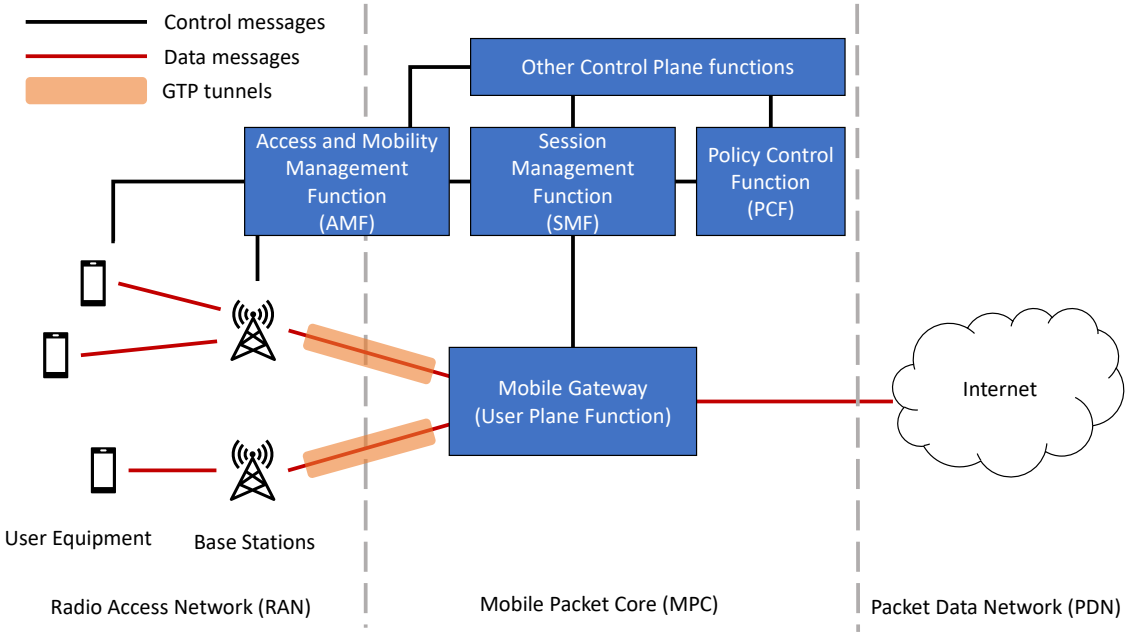

Figure 2.2: 5G Mobile Packet Core Architecture.

Functions carried out by the Mobile Gateway include:

- Packets encapsulation/decapsulation.
- Forwarding and routing of traffic to/from the Packet Data Network.
- Access control, allowing only certain classes of traffic to transit.
- Policy enforcement and QoS handling.
- Traffic inspection.
- Charging support.
- Support the mobility of user equipment between different base stations.

The QoS model in the 5G network is based on the concept of QoS Flow, that represents the finest granularity of QoS differentiation in the PDU Session. A default QoS flow is instantiated upon session initialization and additional flows can be requested (optionally with Guaranteed Bit Rate) to support specific services like a VoIP call. QoS is enforced both in the Access Network and by the Mobile Gateway and is managed by the SMF through dedicated rules. Every QoS Flow is identified by a QFI (QoS Flow Identifier) that is carried with the packet in encapsulation headers.

2.2 eBPF (Extended Berkeley Packet Filter)

Extended Berkeley Packet Filter (eBPF) [4] is a virtual machine integrated into the Linux Kernel that allows to execute custom bytecode injected at runtime in an event-based way. eBPF was introduced in Kernel 3.18 and is the evolution of the classic Berkeley Packet Filter (cBPF), once simply known as BPF.

cBPF was born in 1992 and was a very simple VM used to perform in-kernel packet filtering. The network TAP, a component in the lower layers of the networking stack, copied packets received by network interfaces to the BPF filter, where injected bytecode decided whether the packet needed to be sent to the user space. Matching packets were inserted into a buffer, that could be read by a user space program, such as Tcpdump, through a dedicated API.

eBPF extends BPF architecture making it general-purpose, therefore becoming an interesting technology not only for packet processing, but also for other aspects like security management and kernel monitoring.

This technology allows to extend the behaviour of the vanilla Linux kernel, without requiring the reboot of the system, the re-compilation of the kernel and avoiding the cost of system calls and expensive kernel/user context switching.

Main features of eBPF are discussed in following sections.

2.2.1 vCPU Architecture

eBPF virtual CPU consists of eleven 64 bit registers with 32 bit subregisters, a program counter and a 512 byte large stack. This vCPU executes a general purpose RISC instruction set.

Registers are named r0 - r10: r10 is read-only and contains the frame pointer to access stack space while other registers (r0 - r9) are general purpose. Upon entering execution of the program, register r1 is loaded with the context for the program. The context is defined by the program type, for example, a networking program can have a kernel representation of the network packet (skb) as the input argument. Register r0 is also the register containing the exit value for the BPF program, whose semantic is defined by the type of program.

2.2.2 Safety

eBPF allows the injection of custom code at runtime. While providing a great flexibility this feature is a source of potential risks for the stability of the kernel and the security of user data. Furthermore eBPF programs are executed with preemption disabled in a run-to-completion mode requiring the program to terminate in a brief time and return the control to the kernel.

As a consequence, all programs are analyzed by a Verifier, that can reject them in case one or more safety constraints are not respected. These constraints include:

- Valid instruction opcodes.
- Access to valid memory zones.
- Limited program size: the maximum instructions count was initially set to 4096 and has been raised to above 1 million in kernel 5.1. Besides guaranteeing the termination of the program in a limited time this constraint is also influenced by the desire to keep a low verification and injection time.
- Absence of unbounded loops: initially also bounded loops were forbidden but the constraint has been lifted in kernel 5.3 thanks to the improvement of the verifier. The same consideration made for the above point applies here.

Due to the presence of these constraints, eBPF does not implement a Turing-complete machine, meaning it can not execute arbitrary computation tasks.

2.2.3 Helpers

Helpers are functions provided by the kernel that can be leveraged by eBPF programs. They are executed outside the eBPF context and therefore are not subject to its constraints. Their main applications are:

- Overcoming eBPF limitations, like using the helper `bpf_xdp_adjust_head()` to extend or shrink the packet buffer.
- Interacting with kernel structures, for example the helper `bpf_fib_lookup()` allows to perform a lookup in the kernel routing table.
- Executing complex tasks, like handling VLAN encapsulation with `bpf_skb_vlan_pop()` and `bpf_skb_vlan_push()` helpers.

Available helper functions may differ for each BPF program type. New helpers are continuously added to the kernel, extending the system capabilities.

2.2.4 Maps

Maps are efficient key/value data structures that reside in kernel space. They can be accessed by eBPF programs through dedicated helper functions. Unlike other memory areas, maps are preserved among different executions and can be used to keep the state among one run and another. They can be shared between different programs and can also be accessed by the user programs, providing an efficient way to exchange data between kernel and user space. The kernel guarantees safe concurrent access to maps using the Read-Copy-Update mechanism.

Different types of maps are available, characterized by specific behaviour and structure, some of them are described below:

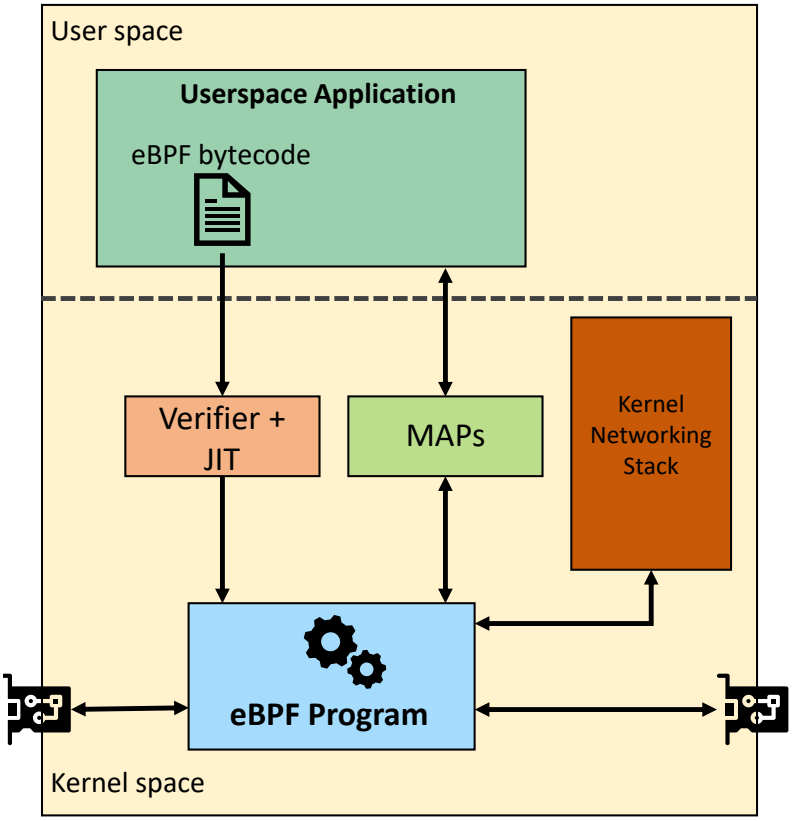

Figure 2.3: Use scenario of a networking eBPF program.

- BPF_MAP_TYPE_ARRAY: behaves like a classic sequential array, providing the fastest possible lookup. All elements are pre-allocated and zero initialized at init time.
- BPF_MAP_TYPE_HASH: data is stored using a hash table. Elements are split into buckets based on the result of the hash function applied to the key. Provides constant time access to data.
- BPF_MAP_TYPE_LPM_TRIE: this map allows to perform a longest prefix match lookup on the key. Useful to implement lookup in routing tables.
- BPF_MAP_TYPE_PROG_ARRAY: contains file descriptors of eBPF programs. Is used by the `bpf_tail_call()` helper to pass the control to another program.
- BPF_MAP_TYPE_PERF_EVENT_ARRAY: allows, among the other things, to push raw data to a perf ring using the `bpf_perf_output()` helper. This data can then be read by a program in user space polling the buffer.

Some of these maps also have a PERCPU implementation (PERCPU_ARRAY, PERCPU_HASH, etc.), that allows to have a different instance of the table for every CPU core providing a performance improvement, since there is no need for additional synchronization mechanisms and the map can be kept into the cache for faster access.

2.2.5 Tail Calls

The mechanism of tail calls allows an eBPF program to call another one with a minimum overhead. Unlike function calls, tail calls transfer the control to a different program and never return. To perform such a call the `bpf_tail_call()` helper must be used, passing it a PROG_MAP containing the file descriptor of the target program and its index inside the map.

In order to avoid undefined execution time the number of consecutive nested calls is limited to 32.

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.

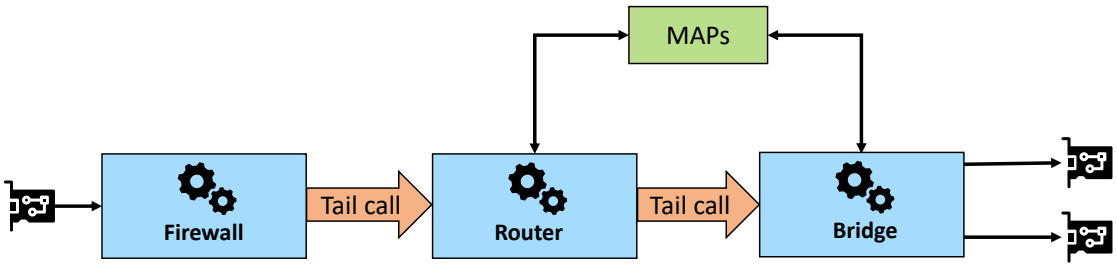

Figure 2.4: Example of a service chain implemented using tail calls.

2.2.6 Program Types

The execution of eBPF programs is triggered by specific kernel events that take the name of Hook Points. Different kernel events are handled by different program types, each one invoked with specific metadata carrying information about execution context. Hook points include all possible kernel events, for example:

- Reception or sending of a network packet.
- Invocation of a system call.
- Access to disk.
- A page fault in memory.

There are two program types related to packet processing: eXpress Data Path and Traffic Control.

eXpress Data Path (XDP)

The eXpress Data Path [5] enables high-performance packet processing in the Linux kernel, by running the eBPF program at the earliest possible point, as soon as the network driver receives the packet. No expensive operations have been executed by the kernel at this point, such as allocating the socket buffer.

As a consequence very 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`).

Thanks to its early processing XDP can bring huge performance improvements in applications that do not require (all) the packets to traverse the networking stack. For example it can be used for Firewalling or DDoS Mitigation, where part of the packets can be immediately dropped, or in Forwarding and Load Balancing, where the traffic can be redirected to other interfaces without requiring further processing of the kernel.

XDP has three operation modes:

- *Native XDP*: default operation mode in which the packet is processed in the driver of the NIC.
- *Generic XDP*: this mode can be used in case the driver does not support XDP and is useful for debugging. The packet is processed at a higher level therefore losing performance advantages of *Native XDP*.
- *Offloaded XDP*: allows the offload of the program to supported SmartNICs, reducing the overhead for the host CPU and further improving performance.

Traffic Control (TC)

This program type allows to process packets in the Traffic Control layer of the networking stack.

At this point the packet has been parsed and copied in a data structure called socket buffer and additional metadata are available to the eBPF program via `struct __sk_buff`, such as the protocol, the priority, the reception timestamp, VLAN associated metadata and layer 3 and 4 information.

While TC programs can't achieve XDP performance they come with some advantages:

- No driver support is required, allowing this kind of programs to be attached to any interface.
- Unlike XDP programs, TC ones can process packets in the egress path of the networking stack.
- Thanks to additional metadata available, a richer set of helpers is provided, allowing to perform complex operations like handling VLAN encapsulation or updating L3 and L4 checksums.

2.2.7 Tool chain

eBPF programs can be written using restricted C code, with the restrictions due to the safety constraints imposed by the Verifier. This code can then be compiled into eBPF assembly using the LLVM (Low Level Virtual Machine) compiler infrastructure: the CLANG fronted, extended to support restricted C, converts the program into the LLVM intermediate representation, LLVM core is then used to apply optimizations and the backend generates the eBPF bytecode.

In older kernel versions the vCPU was realized with an interpreter, while now the bytecode can be Just In Time compiled into native x64 code.

BCC (BPF Compiler Collection) is a toolkit for creating efficient kernel tracing and manipulation programs. It provides macros and structures to simplify the writing of eBPF C code, and includes frontends in Python and LUA to interact with eBPF programs in user space. While being mostly focused on tracing its infrastructure can also be used for network traffic management.

2.3 Polycube

Polycube [6] is an open source framework that enables the creation of fast and efficient in-kernel network functions chain based on eBPF and XDP technologies.

Polycube provides the user with a set of services, such as router, firewall, bridge, etc., that can be dynamically connected and configured to provide custom connectivity to namespaces, containers, virtual machines, and physical hosts. From the developer perspective, Polycube provides the infrastructure to create complete network functions, simplifying the implementation of control and user planes and the management of the interaction between the two. Two standalone applications are also available: *pcn-iptables*, a faster clone of iptables, and *pcn-k8s*, a CNI network plugin for Kubernetes.

Polycube adopts a centralized architecture, in which all management tasks are carried out by a userspace daemon, called *polycubed*. Interaction with the system can happen using a command line interface, called *polycubectl*, or through a RESTful API.

A schematic representation of Polycube architecture is shown in figure 2.5, main aspects are discussed in the following sections.

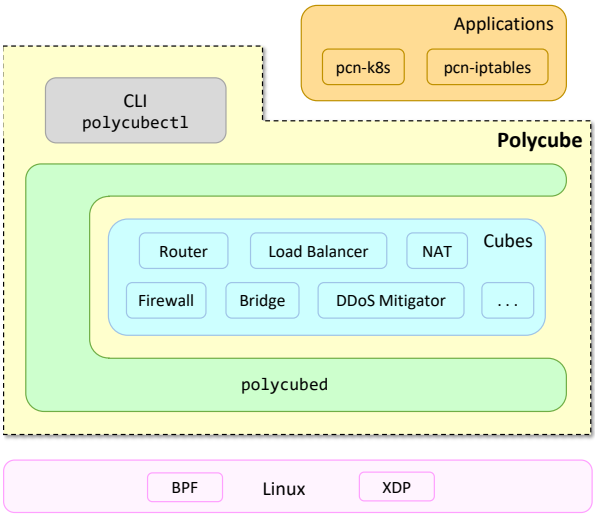

Figure 2.5: Simplified Polycube architecture.

2.3.1 Services

Polycube services represent virtual network functions. Each service is described by a YANG data model that defines its behaviour, the configuration it needs and the interface that must be used to interact with it.

Services are composed by a data plane and a control/management plane.

Data Plane

The data plane is responsible of the processing and forwarding of single packets and is composed by a fast and a slow path.

The *fast path* is executed at kernel level by or more eBPF programs and performs basic tasks like packet parsing and mangling and maps update.

The *slow path* is executed in user space and handles all that packets that require a more complex processing, that either can not be achieved in the fast path due to eBPF limitations or could cause an excessive slowdown. An example of slow path processing is the handling of a packet after an ARP lookup miss. An ARP request needs to be triggered and the packet needs to be buffered waiting for a response, an operation that can not be accomplished in eBPF. After a packet has been processed in the slow path it can be optionally sent back to the fast path, either to the next service of the chain or to the fast path of the same service as a new packet.

Control and Management planes

The control plane is responsible of defining the behaviour of the network function, handling control related protocols (e.g. Routing Protocols, Spanning Tree) and configuring the the data plane, either updating maps or the eBPF code.

The management plane interacts with external entities through the service API, receiving configuration parameters and exporting the status of the function.

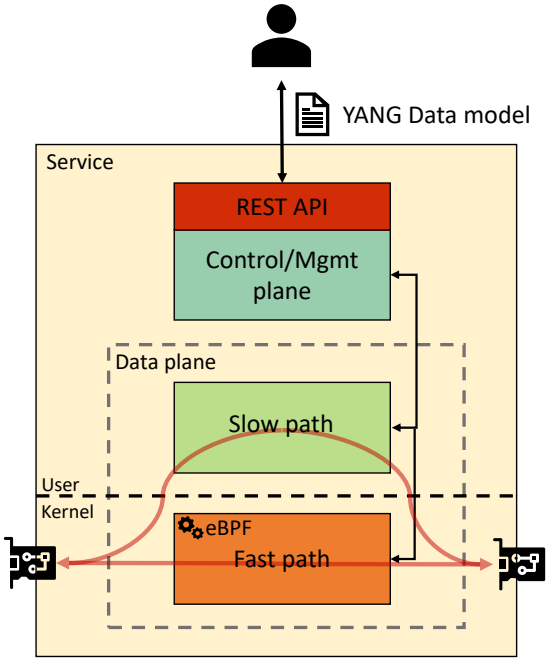

Figure 2.6: Polycube service architecture.

2.3.2 Cubes

Cubes are instances of Polycube services, that can be connected together to create complex service chains.

Before being injected into the kernel, eBPF data plane programs are augmented with additional code that supports Polycube facilities and provides some helpers to simplify the work of the developer. The main function of the program written by the user is inserted into a wrapper function, that provides it with additional metadata and handles its return codes in order to implement connectivity among cubes.

Data plane of services can be instantiated both as a TC and as a XDP (XDP_DRV for native XDP and XDP_SKB for generic XDP) program. To help the developer writing code that is not bound to the program type Polycube provides unified functions to perform tasks that needs a different implementation for TC and XDP programs, such as VLAN encapsulation and packet checksum update.

Two kinds of cubes are available:

Standard Cubes

Standard cubes have forwarding capabilities and can be used to implement services such as a Router or a Bridge.

Polycube introduces the concept of port, a connection point that can link a standard cube to another cube or to a network device. Information about the port on which the packet was received is carried in packet metadata.

Besides dropping the packet and sending it to the kernel networking stack with `RX_DROP` and `RX_OK` return codes, a standard cube can redirect it to another port using the `pcn_pkt_redirect()` function. The code of this function is dynamically generated every time a new port is connected to the cube, in order to either perform a tail call to the eBPF code of a peer cube, or to invoke the `bpf_redirect()` helper to send the packet out of an interface.

Standard cubes can also be instantiated in shadow mode. In this case the cube is associated to a Linux namespace and parameters are kept aligned among these entities.

Transparent Cubes

Transparent cubes do not have forwarding capabilities and have to be attached to a port of an existing standard cube or to a network device. They process packets flowing in or out the entity they are bound to through their set of ingress and egress programs, and can be used to implement services like a firewall or a NAT. Cubes of this type inherit the parameters of the port they are attached to (MAC, IPv4, etc.), multiple instances can be connected to the same port implementing a stack of functions.

Polycube wrapper code allows to correctly link programs in the ingress and egress chain of an interface. Every time a transparent cube is attached or removed this code can be updated injecting a new version of the program, to connect the cube to the correct next entity.

When an instance of transparent cube lets the packet pass with return code `RX_OK`, three situations may occur:

- There is another cube (transparent or standard) in the chain: in this case its eBPF code is executed with a tail call.
- The next entity is a networking device: in this case the packet is redirected using `bpf_redirect()` helper.
- The next entity is the networking stack of the host: the packet proceeds with return code `XDP_PASS` or `TC_ACT_OK`.

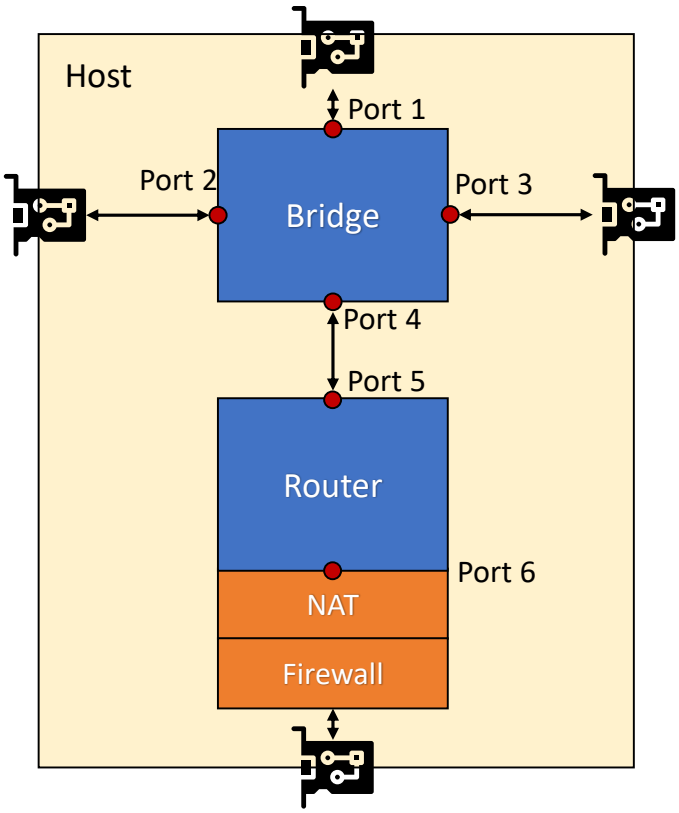

Figure 2.7: Example of chain of cubes.

2.3.3 Polycube Daemon

The Polycube system daemon, *polycubed*, implements the centralized point of control of the framework. It manages the lifecycle of cubes, handling their creation, update, connection, and deletion.

The interaction with the system happens through a REST API, that provides CRUD operations on available resources.

polycubed operates in a service agnostic way: it has no idea on how network functions are implemented internally. Services can be added through a registration phase in which, starting from their YANG data model, polycubed generates an internal representation and a set of REST endpoint that can be used to interact with the resources of the service. Every time a request is made to the API polycubed operates as a proxy: it performs ancillary tasks like validation of the payload, redirects the request to the corresponding method of the target service and sends the response back to the user.

Besides handling the communication between users and services *polycubed* also implements a *kernel abstraction layer* based on the BCC toolkit, used to manage the interaction among the user space component of services and their in-kernel data path. This layer provides methods to add, remove and update programs and to access eBPF maps, it manages the facility that enables logging from the data plane and handles the mechanism to exchange packets and metadata among the fast and the slow path.

Fast and Slow path interaction

eBPF programs can send a packet to the slow path using the function `pcn_pkt_controller()`. This method combines the content of the packet with additional metadata, including the unique identifier of the cube and the reason the slow path is invoked, and sends it to user space using a shared perf ring. Here a thread polling the ring receives the packet, performs demultiplexing identifying the target cube and invokes its `packet_in()` method.

On the other hand, packets directed to the fast path are sent to one of two dedicated TAP interfaces: `pcn_ctrl_xdp` for XDP programs and `pcn_ctrl_tc` for TC ones. An ARRAY_MAP is used to carry additional data needed to correctly forward the packet, this map is handled as a circular buffer to avoid overrides when multiple packets are sent to the fast path before being processed. Once the packet is received by one of the above mentioned interfaces a *Decapsulator* eBPF program is triggered: it retrieves data from the map, fills metadata of the packet and either invokes the program of a cube, sends the packet to the networking stack or redirects it to a net interface.

Architecture of polycubed and its interaction with services and the kernel are shown in figure 2.8.

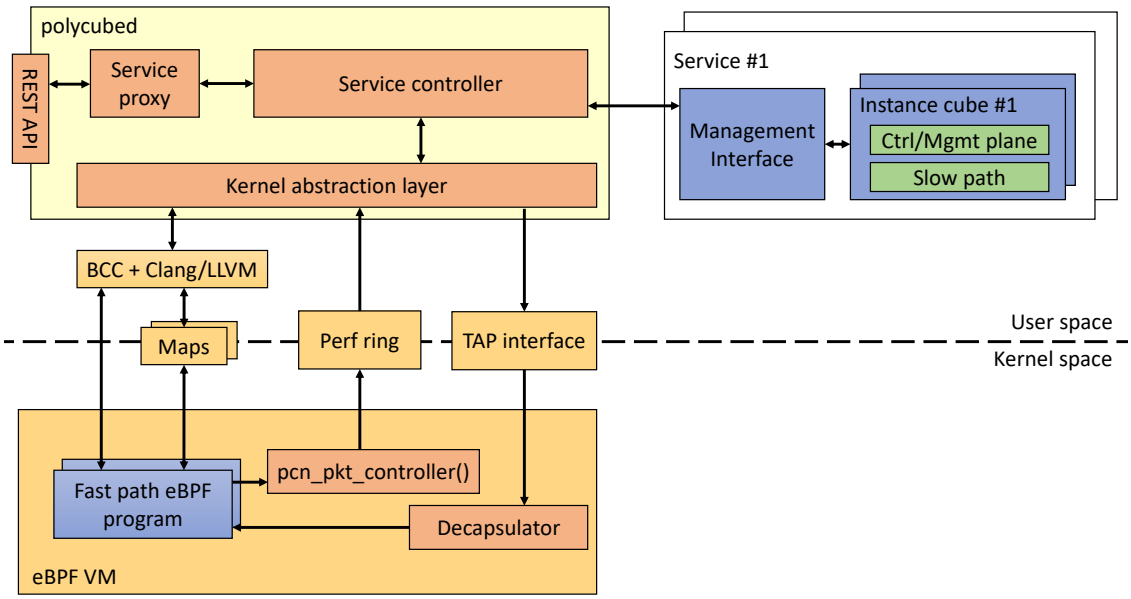

Figure 2.8: Polycubed architecture.

2.3.4 Polycube CLI

The Polycube CLI, *polycubectl*, provides a simplified way to interact with the system, mapping user commands to HTTP requests directed to the REST API. Thanks to its help command and auto-completion feature it can be used to quickly explore the structure of available services.

2.4 Related Work

Due to flexibility and performance constraints of the 5G core network, the virtualization of its data plane functions has been subject of different research articles.

In [7] the authors focus on the possibility to dynamically compose simple functions to create specific chains to handle the traffic of different users. They propose a prototype based on the Click modular router, able to run in containers and to process packets at high speed thanks to the acceleration provided by the kernel bypass technology Netmap. The architecture is based on a set of different nodes, each one running an instance of the Click router, that can exchange packets and associated metadata using a custom tunneling mechanism. Every node contains (besides tunnel encapsulation and decapsulation modules), a set of User Plane Functions (three simple functions are defined: a packet counter, a marker and a header compressor), and a Forwarding Element, that sends packets across related functions. An handover mechanism is also proposed, with buffers in every node to allow the movement of user equipment among different base stations without packet loss. The performance of the system is then studied, varying the number of connected users and the complexity of the of functions chain.

[8] targets those use cases of the 5G network where low latency and high data rates are crucial aspects. It shows how programmable switch ASICs are an effective instrument to handle these classes of traffic, allowing a higher performance with respect to other virtualization techniques based on commodity servers, while losing some flexibility. A mobile gateway pipeline is defined using the P4 language, and is then compiled both for the Barefoot Tofino programmable ASIC and as a ODP-DPDK program to run on a x86 server. Performance of the two solutions are then compared, showing how the hardware switch achieves better results and is not influenced by the number of configured flows. The paper highlights how a mix of different technologies could be the right approach to meet the broad range of requirements that 5G services ask.

In the end, [9] provides a study on scalability properties of the main programmable software switches currently available from a 5G vendor perspective. Authors of the paper first provide a taxonomy for data plane scalability, distinguishing among static and dynamic workloads and identifying three main scalability dimensions:

- *Concurrency*: the number of CPU cores dedicated to parallel packet processing.
- *Pipeline size*: the number of VNFs composing the chain, the static number of configured rules and the rate and type of updates applied at run-time.
- *Active flows count*: the number of individual transport layer sessions flowing through the network function.

The paper then presents *TIPSY (Telco pIPeline benchmarking SYstem)*, a tool to perform automated and reproducible tests on telco network functions. The system allows to tune scalability dimensions described above, automates the process of configuring the pipeline under test and provides different software switches as backends.

10 5G pipelines are defined:

- 8 micro-benchmarks: a function to simply forward packets between two interfaces, used to test base performance, l2 and l3 forwarders, a firewall, a NAT, a rate limiter and a tunnel encapsulator/decapsulator.
- 3 macro-benchmarks, based on a combination of the above ones: a Data Center Gateway, a 5G Mobile Gateway and a Boradband Network Gateway.

Available backends include:

- *Open vSwitch (OVS)*, the most popular programmable switch, allowing processing packets both in-kernel using a kernel module or in user space with DPDK.
- The *Berkeley Extensible Software Switch (BESS)*, a DPDK based framework for software switching that allows the composition of base modules into a VNF graph using a Python interface.
- Other software switches: *ESwitch* (Ericsson proprietary solution), *Lagopus* and *t4p4*.

Performance of these technologies are compared in different scenarios, highlighting their strengths and drawbacks.

In conclusion, the authors show how analyzed software switches (with the exception of ESwitch) can't match the scalability of fixed-function hardware appliances in terms of costs and energy consumption, and highlight how further research in this field is needed.

Chapter 3

Prototype Architecture

This chapter starts explaining the general architecture of the 5G Mobile Gateway prototype and the design principles that led to its definition. It then gets into more details on the functions carried out by the base modules composing the final solution.

3.1 General Architecture

The prototype of the Mobile Gateway has been designed following the principles of *Network Functions Composition*. A set of smaller modules carrying out basic functions has been identified and then combined to obtain the final pipeline. Mentioned modules have been designed and implemented with the goal of being general purpose and therefore reusable in different applications. Building basic blocks that are not bound to a specific use case (in this case the gateway) allows to combine efforts in their development and exploit solutions already available and tested. Moreover it pushes for further improvements of the solutions since the benefits can be shared by different applications.

A set of simplifying assumptions has been made for the prototype:

- The management of GTP tunnels and QoS Flows has been merged in order to have one tunnel for every QoS (in a similar way to how EPS Bearers in the LTE core network work). As a consequence the TEID identifies both the GTP tunnel and the QoS Flow.
- The same TEID is used for both the uplink (BS to MGW) and the downlink (GW to MGW) tunnels.

A subset of the functionalities provided by the 5G Mobile Gateway has been selected and mapped to different modules, that have then been connected to compose the chain shown in figure 3.1. These functionalities are:

- Encapsulation and decapsulation of GTP tunnels, carried out by the *GTP Handler* module.
- Access control and rate limiting for QoS enforcement, provided by the *Traffic Policer* module.
- Packets classification (mapping of packets the corresponding tunnel/QoS flow), provided by the *Traffic Classifier*.
- Routing and forwarding of packets, carried out by the *Router* module.

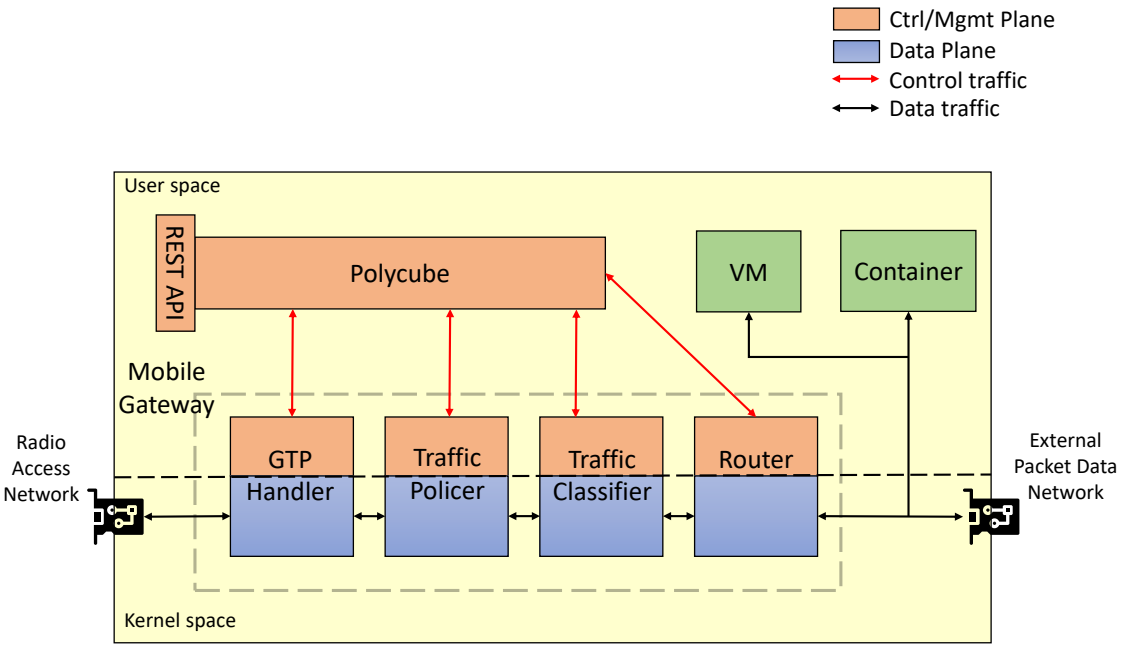

Figure 3.1: Mobile Gateway prototype architecture.

Packets flowing from the user equipment towards a remote host on the Packet Data Network, composing the *Uplink* traffic, traverse these blocks in the given order, while packets directed to the user equipment, *Downlink* traffic, traverse it in the reverse order. A single instance of the Router is needed, while a chain of GTP Handler, Policer and Classifier is needed for every interface leading to the user equipment.

Thanks to the modular approach additional functionalities of the Gateway, such as a counter to support charging or a *Deep Packet Inspector*, can be easily designed as independent modules and plugged into the chain.

Polycube provides the infrastructure to build and link the services together. Every service is composed by an in-kernel data plane and a user space control / management plane, is described by a YANG data model and can be accessed by a

RESTful API, directly with HTTP requests or through the Polycube CLI. Packet metadata is used by the GTP Hanlder, the Policer and the Classifier to share the information on the Tunnel Endpoint ID (TEID) associated to the packet currently processed.

The integration with the kernel provided by the eBPF technology allows to deploy the solution in a traditional data center scenario, where it can either route the packets towards different machines or external networks, or forward them to other functions running on the same host in containers or virtual machines, all of this without requiring a specific support from the virtualization infrastructure. This can provide benefits in emerging use cases like edge computing, where all packets coming from the user equipment could be handled on a single machine.

Following sections provide more details about building modules.

3.2 GTP Handler

This is a very simple module responsible of handling GTP tunnels, decapsulating packets coming from the user equipment and encapsulating the ones coming from the Packet Data Network.

The service is configured with:

- The set of user devices reachable through the current interface, each one identified by its IP address and associated to the IP address of the base station terminating the GTP tunnel (Tunnel Endpoint).
- The IP and MAC addresses of the interface the service is attached to.

Actions carried out for uplink traffic (UE to PDN) are:

1. Check if incoming packet is an IP packet directed to this interface.
2. Check if packet is GTP encapsulated: UDP header with destination port 2152.
3. Extract the TEID from the GTP header and save it into packet metadata.
4. Remove external IP, UDP and GTP headers.
5. Pass the packet to the next module.

For downlink traffic (PDN to UE):

1. Check if incoming packet is an IP packet with an associated TEID.
2. Lookup the destination address to find the IP of the tunnel endpoint.

3. Push external headers: the IP header with the IP of the current interface as source address and the IP of the remote base station as destination address, the UDP header with destination port 2152 and the GTP header with the TEID read from packet metadata.

4. Send the packet out of the interface.

For both directions of traffic if a packet does not match one of the checks it is passed to the next module without performing encapsulation / decapsulation. This allows protocols such as ICMP or ARP to continue to work properly.

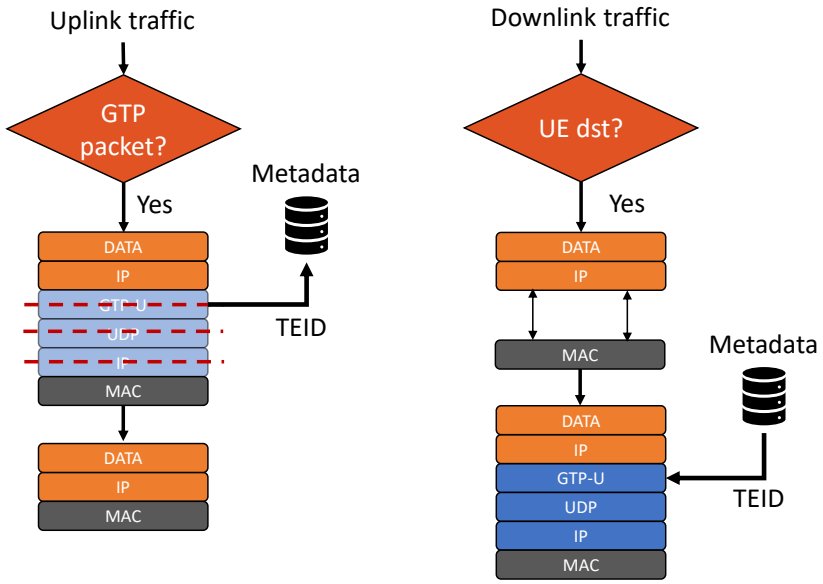

Figure 3.2: GTP Handler functions.

3.3 Traffic Policer

The Traffic Policer module provides access control and rate limiting functionalities. The service is configured with a list of contracts, each one defining how a specific class of traffic must be handled. Three actions can be applied:

- Let the packet pass.
- Drop the packet.
- Apply rate limit.

The classification logic is not implemented into the service, that relies on packet metadata to get this information. This allows the policer to correctly process both

the packets coming from the GTP handler, whose class has been set using the TEID of the GTP header, and the ones coming from the Packet Data Network, that are classified by a dedicated module.

For the rate limiting functionality two techniques have been considered: traffic policing and traffic shaping.

3.3.1 Traffic Shaping vs Policing

Policing and Shaping are two bandwidth management techniques used to guarantee that traffic complies with a desired profile.

Shaping

Traffic Shaping requires a single input parameter, the average rate, and operates by buffering incoming packets that exceed the given threshold and then sending them out at the desired rate. This technique allows to handle bursty traffic without loosing packets in case the size of the bursts doesn't exceed the buffer. Shaping is commonly applied at the network edge to control traffic entering the network, in order to avoid congestion and latency increase.

It can be implemented with a *Leaky Bucket* algorithm in its *bucket as a queue* version. This algorithm draws its name from its analogy with a bucket with a leak, where water poured in it in an intermittent way then leaks out with a constant rate. In a similar way, packets exceeding the desired rate are stored in a FIFO buffer, one for each separately shaped class, until they can be transmitted in compliance with the associated traffic contract.

One of the disadvantages of this technique is that, since it only sends packet 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.

Policing

On the other hand, Policing represents a more drastic approach to bandwidth management. This technique allows to control two parameters of the traffic profile: 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.

Different algorithms can be used to implement Policing [10]: *Fixed Window Counters*, the *Token Bucket* (also known as the *bucket as a meter* version of the *Leaky Bucket*), the *Sliding Log* and the *Sliding Window*.

Among the two techniques Traffic Policing has been chosen due to the limitations imposed by the eBPF data plane. The execution of fast path programs can only be triggered by the reception of a packet and therefore doesn't allow to buffer packets and then send them out at a fixed rate without relying on the user space slow path, that would bring an increase of overhead. Following section explains into details algorithms analyzed for the prototype.

3.3.2 Rate limiting algorithms

Fixed Window Counter

The Fixed Window Counter represents the most simple rate limiting algorithm.

The time is subdivided in a set of fixed windows of size w and a counter is used for every traffic class to store the number of bits forwarded in the current time span. Counters are associated with a threshold that represents the maximum number of bits that can be forwarded in every window, whose value is the product of the desired average rate r and the window size. Every time a packet is received a check is performed to verify that the counter can be increased by the size of the packet without exceeding the threshold. In case the check is positive the counter is updated and the packet is forwarded, otherwise the packet is dropped.

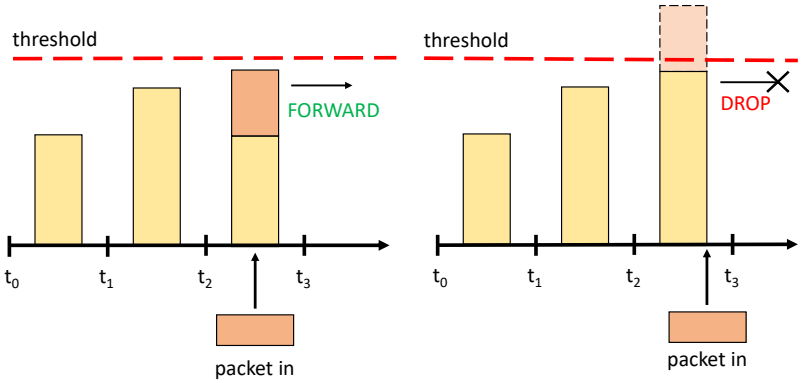

Figure 3.3: Fixed Window Counter.

This solution is very simple but suffers form different problems if compared to subsequent ones:

- The maximum burst size can't be set as an independent parameter but is the product of the desired rate r and the window size w.
- If the the window is too long and the input rate is high the output traffic could have a bursty profile, while if it is too short it could cause the maximum burst size to be too small, therefore dropping more packets than expected.

- If a burst of packets is forwarded at the end of a window and a new one is sent at the beginning of the next window the output could result in a burst of up to double the expected size.

Token Bucket

The algorithm requires, for every flow, two input parameters: 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.

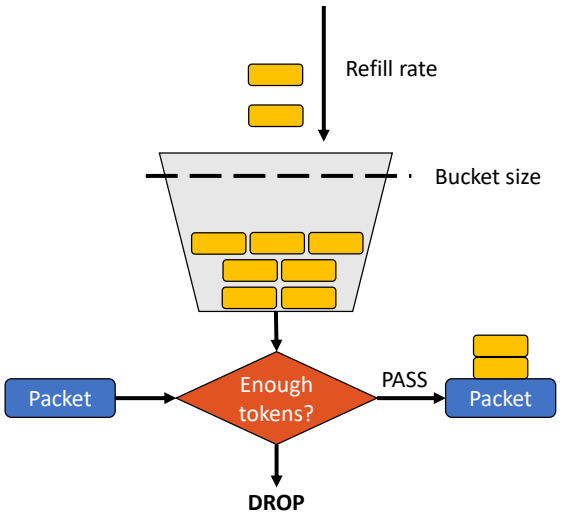

Figure 3.4: Token Bucket representation.

Sliding Window

The Sliding Window algorithm [11] also allows to configure both the desired average rate r and the maximum burst size B. These parameter are used to compute the size of the window $w = B/r$. In every point in time the portion of the window that precedes the point (located on its left on a temporal axis) represents the amount of time that can be spent to forward traffic at the desired rate r. As a consequence, every time a packet of size S is transmitted the window needs to be shifted forward of an amount of time equal to its transmission time $t = S/r$.

Depending on the arrival time of the packet three situations may occur:

- The portion of the window preceding the arrival time of the packet (on its left) is greater or equal to its transmission time: the packet can proceed and the window is moved forward of the corresponding time.

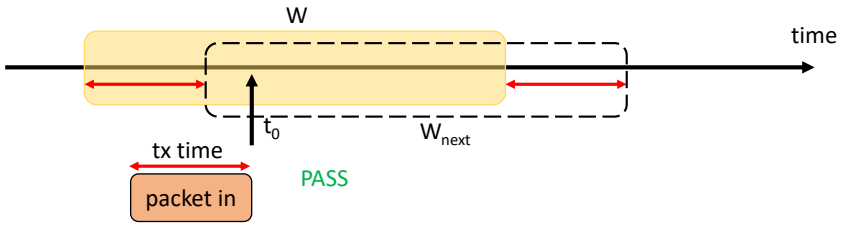

Figure 3.5: Sliding Window - Pass the packet.

- The portion of the window preceding the arrival time of the packet is shorter than its transmission time: the packets needs to be dropped and the position of the window is not changed.
- The packet arrives after the window (on its right): the window has not been moved for too long, the packet can be forwarded (assuming its size is lower the the maximum burst), and the window is moved forward in order to end at packet arrival time plus its transmission time.

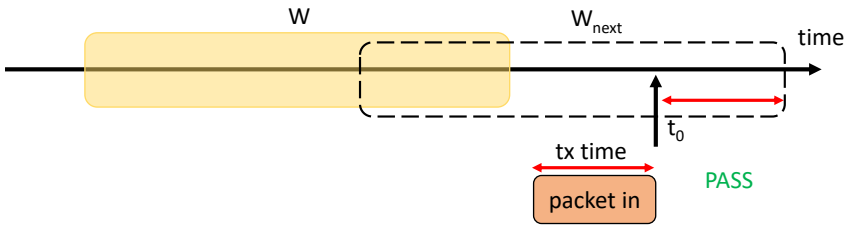

Figure 3.6: Sliding Window - Old window.

This algorithm can be seen as a different interpretation of the Token Bucket and, given the same input parameters, behaves in the same way.

3.4 Traffic Classifier

This module is responsible of classifying packet based on the fields of their headers. Every class is identified with an integer id written into packet metadata.

According to the principle of building general purpose services a wide range of fields is available for classification:

- Source and destination MAC addresses.
- MAC ethertype, supporting ARP and IPv4.
- Source and destination IP addresses. These values are expressed in address and prefix length format (e.g. 10.0.1.0/24), allowing to perform match both on single hosts (using /32 prefix) and on networks.
- Layer four protocol, supporting TCP, UDP and ICMP.
- Source and destination ports of UDP or TCP header.

The traffic class is also associated to the direction of the traffic, enabling to classify packets only in the ingress, egress or both the directions. This allows to avoid overwriting an existing class in case another service performing classification is present and to prevent additional overhead in case this information is not needed for packets flowing in a certain verse.

To avoid clashes among classes matching the same packet a priority is also used: in case there are multiple matches the class with the higher priority is chosen.

In the Mobile Gateway pipeline use case the service is configured to classify only the downlink traffic (egress packets), providing the value that is later used by the Policer to apply optional rate limit and by the GTP Handler to identify the ID of the tunnel (TEID). In the base configuration just a /32 destination address is used, to split traffic in one tunnel for every user device. Additional fields can be used to further subdivide traffic in classes related to different services, such as HTTP or VoIP, that can than be handled with a dedicated QoS.

Implementing packet classification in an efficient way in the kernel is a challenging task, due to the restricted environment in which eBPF programs are executed. The lack of support for unbounded loops, the limited size of the stack and the possibility to only use a predefined set of data structures makes the implementation of widespread matching algorithms, such as cross-producting and approaches based on decision trees, unfeasible. This problem has already been faced in [12], where authors propose and eBPF based clone of *iptables* and need an efficient way to classify packets. Solutions adopted in this work has proven to be applicable to the implementation of the Traffic Classifier as well. This solutions include the Linear Bit Vector Search algorithm and the dynamic generation of the classification pipeline.

3.4.1 Linear Bit Vector Search algorithm

A trivial approach to packet classification encompasses the sequential scan of all classification rules until a match is found. This technique has a linear complexity ($O(n)$) requiring in the worst case, with n rules and k classification fields, a total number of $k * n$ comparisons. As shown by Linux *iptables*, this algorithm leads to poor performance when the the number of rules grows and therefore does not suit the Mobile Gateway scenario, where the number of classes can go from a thousand to multiple tens of thousands.

The Linear Bit Vector Search algorithm presented in [13], while still having a linear complexity, allows to exploit the parallelism of CPU registers (64 bits in modern processors) to speed up the classification process.

The algorithm requires a bi-dimensional table for every header field used for matching. This table maps each value assumed by the field in the current set of rules to a bit-vector. The bit-vector contains one element for every class, set to 1 if the class is compatible with the current value and to 0 otherwise. An additional wildcard entry can be added to handle those values of the field that aren't explicitly specified by any rule. When the bit-vector is computed, classes are sorted by priority, therefore having the first (least significant) bit associated to the highest priority class.

The classification is performed according to following steps:

1. Headers of the packets are parsed, saving fields of interest for subsequent steps.

2. A bit-vector responsible of storing partial results of the matching process is associated to the packet. At this point the packet may match on all classes, therefore all bits are set to 1.

3. A matching step is performed for every field: the field is looked up in the corresponding table and the retrieved bit-vector is combined with the one of the packet with a bit-wise AND operation. At this point two early stop conditions may occur, allowing to immediately break the classification process since there are no matching classes:

   - The lookup on the table fails. This can happen when all classes specify a value for the current field and no wildcard is therefore present.
   - The result of the bit-wise AND operation is a bit-vector with all elements set to zero.

4. The least significant bit set to 1 is retrieved from the resulting bit-vector and its value is mapped to the corresponding class ID.

figure 3.7 shows a simplified example of successful LBVS classification while figure 3.8 show a possible case of early stop.

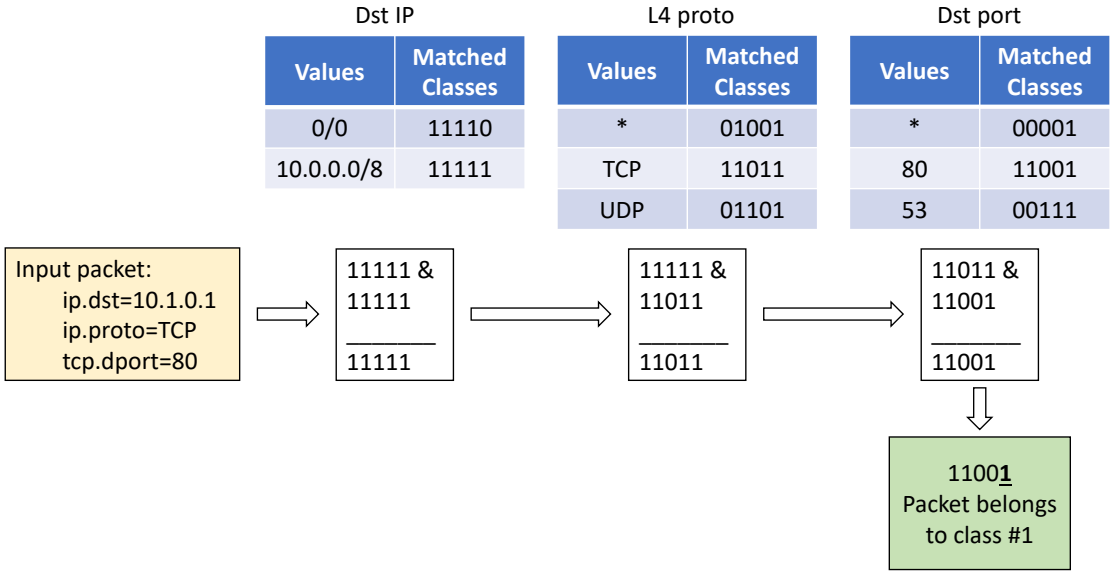

Figure 3.7: Example of successful Linear Bit Vector Search.

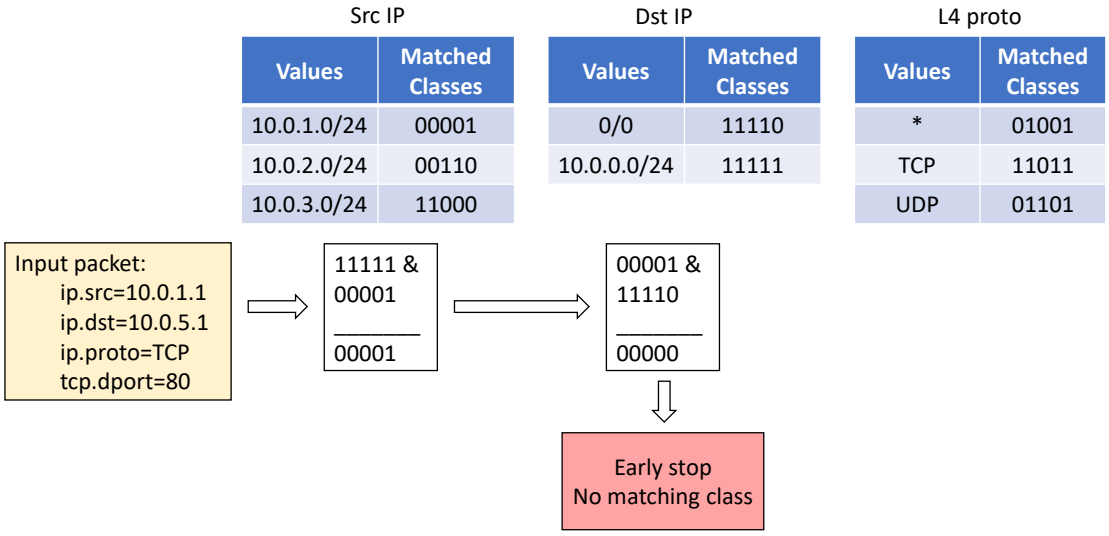

Figure 3.8: Example of early stop in Linear Bit Vector Search.

The number of operations performed in the parsing phase and in the final class identification phase is not related to the number n of configured classes. The complexity of the algorithm is therefore related to the tasks carried out in the matching steps. In every step 2 main operations are performed:

- The lookup of the header field in the corresponding table, that has a worst case complexity at most equal to $O(log(n))$ in case binary search is used (though,

as shown in the implementation chapter, the lookup is usually faster thanks to hash maps).

- The bit-wise AND among bit vectors, that, with a word size equal to w, requires exactly n/w iterations.

The bit-wise AND operation therefore dominates the complexity of the algorithm, leading to a final cost, considering k header fields, of $k * n/w$ operations.

3.4.2 Dynamic generation of the classification pipeline

Not all configurable header fields could be needed at a certain time and this could cause the execution of useless operations. The possibility provided by eBPF to update and re-inject the code at run-time can be useful to overcome this problem.

The eBPF program implementing the fast path of the classifier is generated every time a class is added or removed from the service and is crafted to perform only operations needed to match on fields currently configured. This allows to perform two main optimizations, shown in section 3.4.2:

- Avoid the parsing of layer 3 and layer 4 header, in case no fields of these levels are needed.
- Remove the code responsible of matching a field that is not specified by any class.

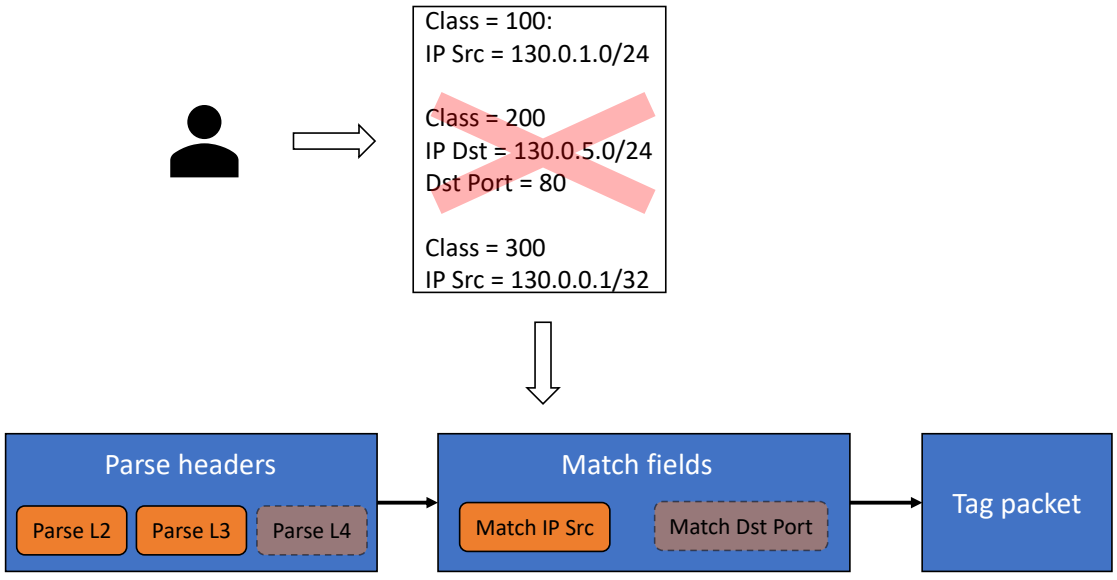

Figure 3.9: Dynamic generation of the classification pipeline.

3.5 Router

The router module is responsible of performing routing and forwarding of packets between the Access Network and the Packet Data Network. To implement this block the *pcn-router* service already available in the Polycube framework has been chosen. This implements a very simple router that only supports IPv4 and static routing.

The router connects to other elements with a set of ports, each one identified by a name and configured with a primary IP address, a list of secondary IP addresses and a MAC address. If the port is connected to a network interface of the host these parameters are aligned between the two entities. If no MAC address is configured (neither from port configuration nor from an existing interface) then a random one is generated.

The static routing table can be configured with a set of routes, each one characterized by the destination network (address and prefix length), the next hop and the path cost. When an IP address is configured on a port an additional *local* route toward the network the IP belongs to is added.

Besides routing IPv4 packets the service is also able to answer to ICMP Echo Request messages and to handle the Address Resolution Protocol. Entries of the ARP table can also be statically configured, setting the IP address of the host, its MAC address and the port that allows to reach it.

In the Mobile Gateway pipeline the router is configured with a set of ports directed to the Access Network, each one with an instance of the pipeline of the other three modules attached, and a set of ports facing the Packet Data Network. These could be either connected to physical interfaces of the host, leading to external networks, or to virtual interfaces, allowing the communication with VMs or containers running on the same host. Besides routes toward remote network on the PDN the routing table is configured with one /32 route for every user device, with the next hop set to the IP address of the Base Station the device is connected to or to another address if additional L3 entities are used to connect the Gateway to the base station, such as a Data Center Gateway.

Chapter 4

Prototype Implementation

This chapter explains how concepts expressed in the Architecture section has been implemented from a coding point of view, diving deeper in classes definition, their interconnection and the workarounds used to solve problems encountered in the process.

While the Router service already available in the Polycube framework defines a *standard cube*, connecting to other network entities (interfaces or other services) through a set of ports, the GTP Handler, the Policer and the Classifier have been defined as *transparent cubes*, to be attached to ports of the Router facing the Access Network.

Programming languages used in this phase are *C++* for the implementation of the control plane and *C* for the implementation of the in-kernel fast path of the data plane (no slow path is needed by the solution). Additionally, the *YANG* data modelling language has been used to describe the resources handled by each service.

4.1 Automatic Code Generation

The Polycube framework provides an automatic code generation tool, *polycube-codegen*, that can be used to generate a stub of the source files needed to implement the service starting from its YANG data-model.

Generated files have two main aims:

- They implement all the boilerplate code needed by the service to interact with the Polycube framework.
- They provide a starting point for the implementation of the internal logic of the service.

polycube-codegen operates by first using the *pyang* module to map the YANG data-model into an intermediate JSON representation, compliant with the OpenAPI specifications, and then, starting from this representation, creating a stub of

the source files, through the *swagger-codegen* module.

Generated files include:

- `src/base/{resource-name}Base.[h,cpp]`: one base class is generated for every resource defined in the data-model (including the service itself), this classes define the interface that must be implemented to be compliant with the management API.
- `src/api/{service-name}Api.[h,cpp]` and `src/api/{service-name}ApiImpl.[h,cpp]`: these classes implement the entry point to the service, providing methods to map every operation exposed by the REST API to operations on the resources. These methods are called by the service proxy in the Polycube daemon.
- `src/serializer/`: contains classes that perform marshalling and unmarshalling of JSON data exchanged by the service.
- `src/{resource-name}.[h,cpp]`: These classes implement the corresponding interface of the base directory, they provide a standard implementation for some of the methods, while others must be written by the programmer to define the actual behaviour of the service.
- `src/service-name_dp.c`: contains the fast path code for the service.

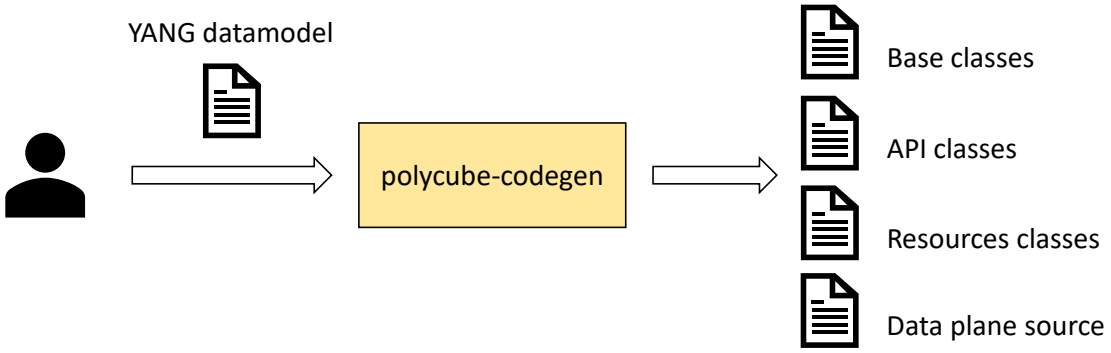

Figure 4.1: Polycube code generation.

4.2 GTP Handler

4.2.1 Data model and control plane

The data model of the service is very simple and defines a `list` of user equipment, each one identified by its IP address and characterized by the IP address of the tunnel endpoint (base station) used to reach the device.

The control plane of the service is composed by two classes:

- `UserEquipment`: stores information about a single user device. Its lifecycle is bound to a corresponding entry in the `user_equipment` eBPF map. The entry is added to the map in the constructor of the class, removed in the destructor and updated in case the tunnel endpoint is modified.

- `GTPHandler`: stores an `unordered_map` of *shared pointers* to user devices and provides getters and setters to manage the list. Upon creation of the service two callbacks are registered using the `subscribe_parent_parameter()` provided by the `TransparentCube` class of Polycube library, to be informed on changes of the MAC and IP addresses of the port the cube is attached to. Every time a notification is received the code of the data plane is re-injected with the updated parameters, needed to recognize packets directed to the gateway and to set the IP source address of the external IP header in GTP encapsulation.

4.2.2 Data plane

Two eBPF programs implement the data plane, one for the ingress and one for the egress direction, carrying out tasks explained in the *Architecture* chapter. Depending on the program type, two helpers are used to expand and shrink the packet buffer in order to add or remove encapsulation headers:

- `bpf_xdp_adjust_head()`: is used in programs of type *XDP_DRV* and *XDP_SKB*, the space is added/removed at the beginning of the packet buffer, therefore requiring to move the MAC header with a `memcpy()` before calling the helper.

- `bpf_skb_adjust_room()`: is used in programs of type *TC* and allows to allocate/remove the space between the MAC and the IP headers using the flag `BPF_ADJ_ROOM_MAC`.

When reading or writing the GTP header the TEID field is retrieved/stored in the `traffic_class` field of the `pkt_metadata` structure shared by cubes in a Polycube chain. Listings below show how additional headers are set in the encapsulation program.

```
ip->version = 4;
ip->ihl = 5;              // No options
ip->tos = 0;
ip->tot_len = htons(ntohs(inner_ip->tot_len) + GTP_ENCAP_SIZE);
ip->id = 0;              // No fragmentation
ip->frag_off = 0x0040;  // Don't fragment; Fragment offset = 0
ip->ttl = 64;
ip->protocol = IPPROTO_UDP;
ip->check = 0;
ip->saddr = LOCAL_IP;
ip->daddr = ue->tunnel_endpoint;
...
__wsum l3sum = pcn_csum_diff(0, 0, (__be32 *)ip, sizeof(*ip), 0);
pcn_l3_csum_replace(ctx, IP_CSUM_OFFSET, 0, l3sum, 0);
```

Listing 4.1: External IP header

```
udp->source = htons(GTP_PORT);
udp->dest = htons(GTP_PORT);
udp->len = htons(ntohs(inner_ip->tot_len) + sizeof(*udp) +
                 sizeof(struct gtp1_header));
udp->check = 0;
```

Listing 4.2: UDP header

```
#define GTP_TYPE_GPDU 255  // User data packet (T-PDU)
                           // plus GTP-U header
#define GTP_FLAGS 0x30     // Version: GTPv1, Protocol Type: GTP,
                           // Others: 0
...
gtp->flags = GTP_FLAGS;
gtp->type = GTP_TYPE_GPDU;
gtp->length = inner_ip->tot_len;
gtp->tid = md->traffic_class;
```

Listing 4.3: GTP header

4.3 Traffic Policer

Three versions of the Traffic Policer service have been implemented, each one leveraging one of the rate limiting algorithms explained in the *Architecture* chapter. These versions have then been compared (details in the *Evaluation* chapter) to show advantages and drawbacks of each one. Following sections explain the common parts of the data model, the control and the data planes and provide details on the implementation of the different algorithms.

4.3.1 Data Model

The data model of the Traffic Policer defines a set of contracts that that allows to instruct the service on how to handle different classes of traffic. Possible actions are defined by the *enumeration* `action-type` and are:

- `PASS`: let all the packets pass.
- `LIMIT`: apply rate and burst limits to the traffic.
- `DROP`: drop all the packets.

Every contract is described by three items:

- `action`: one of the three actions defined above.
- `rate-limit`: the maximum average traffic rate (in Kbps).
- `burst-limit`: the maximum size of a burst of packets (in Kbits), not available in the Fixed Window Counter implementation of the service.

`rate-limit` and `burst-limit` are optional values and have to be set only in case the action is set to `LIMIT`. To avoid updates that could bring to an inconsistent state, such as changing the action from `PASS` to `LIMIT` without setting burst and rate limits, data of the contract can only be updated with a dedicated *action*, therefore allowing to control the correctness of the operation in a single method of the service. This is made possible defining two versions of the contract-data, a fixed one and an updatable one, as shown below:

```
1 grouping contract-data {
2     leaf action { type action-type;
3                   polycube-base:init-only-config; }
4     leaf rate-limit { type uint64;
5                       polycube-base:init-only-config; }
6     leaf burst-limit { type uint64;
7                        polycube-base:init-only-config; }
8 }
9
```

```
grouping updatable-contract-data {
    uses contract-data;
    action update-data {
      input { uses contract-data; }
    }
}
```

Listing 4.4: Contract data description

Two main resources are defined in the data model:

- The list of contracts, each one associating contract data to a traffic class, expressed as a `uint32` value.
- The default contract, applied to all traffic whose class has not been explicitly configured (by default performing the `PASS` action).

4.3.2 Data and Control Planes

Data Plane

Information of every contract is stored in the `struct contract` data structure, that holds the action to perform on the traffic along with additional fields needed to handle the rate limiting functionalities, which depend on the implemented algorithm. These structures are stored in two eBPF maps: an `ARRAY_MAP` with a single element to store the data of the default contract and a `HASH_MAP` to map every class id to its contract. The code of the data plane is injected both for the ingress and the egress path, and maps are shared among the two direction declaring them using BCC helpers shown in listing 4.5.

```
#if POLYCUBE_PROGRAM_TYPE == 1  // EGRESS
BPF_TABLE("extern", int, struct contract, default_contract, 1);
BPF_TABLE("extern", u32, struct contract, contracts,MAX_CONTRACTS);
#else
BPF_TABLE_SHARED("array", int, struct contract,default_contract,1);
BPF_TABLE_SHARED("hash", u32, struct contract, contracts,
                 MAX_CONTRACTS);
#endif
```

Listing 4.5: Maps declaration in the dataplane

Upon reception of a packet the traffic class is retrieved from the corresponding field of the metadata (`struct pkt_metadata`) provided by Polycube to the function and is used to perform a lookup on the `contracts` map. In case the lookup fails the value of the default contract is retrieved from the corresponding map. In the end the proper action is applied to the packet, either dropping it, letting it pass or applying one of the rate limit functions described in following sections.

Control Plane

The control plane is implemented by three main classes: `Policer`, `Contract` and `DefaultContract`.

`Policer` is the access point of the service, it holds a *shared pointer* to an instance of `DefaultContract` that is instantiated upon creation of the service and can't be deleted, and an hash map (`unordered_map`) to store all the contracts. It provides getters and setters methods to read, add and delete specific contracts and to read the default one.

`Contract` and `DefaultContract` store the action and the optional rate and burst limits of the contract, plus the traffic class for the `Contract` class, all accessible through getters methods. The `updateData()` method is used to update these data with a single operation and allows to check that a consistent configuration is provided as explained in section 4.3.1. This method calls the `updateDataplane()` method (also called on object initialization) that sets the information of the contract in the corresponding eBPF map.

4.3.3 Fixed Window Counter implementation

In this solution a `s64` counter is stored in the `struct contract`, holding the number of bits that can still be forwarded in the current time window.

Every time a packet is processed it is first checked if the counter value is greater or equal to its size (in bits), in case it is the counter is atomically decreased by the size of the packet using the LLVM builtin `__sync_fetch_and_add()`, and the code `RX_OK` is returned to let the packet pass, otherwise the packet is dropped with code `RX_DROP` and the counter is not updated.

In the user space the `resetCounters()` function of the `Policer` class is run in a separate thread and is responsible of resetting counters to their original value every time a new window begins.

The size of the window has been set to one second.

```
1  void Policer::resetCounters() {
2    auto t = std::chrono::system_clock::now();
3    while (!quit_thread_) {
4      {
5        std::lock_guard<std::mutex> guard(contracts_mutex_);
6        if (default_contract_->getAction() == ActionTypeEnum::LIMIT)
7        {
8          default_contract_->updateDataplane();
9        }
10       for (auto &entry : contracts_) {
11         if (entry.second->getAction() == ActionTypeEnum::LIMIT) {
12           entry.second->updateDataplane();
13         }
14       }
15     }
```

```
16      std::this_thread::sleep_until(t + std::chrono::seconds(1));
17      t = std::chrono::system_clock::now();
18    }
19  }
```

Listing 4.6: resetCounters() function

The function has to scan the `contracts_` map to handle all the contracts with a `LIMIT` action and the `contracts_mutex_` is used to prevent other methods of the class from updating the structure while this operation is ongoing.

The `updateDataplane()` function of `Contract` and `DefaultContract` classes writes the action and the initial value of the counter in the corresponding entry of the eBPF map and a mutex specific of every contract guarantees that this operation isn't performed concurrently by the counter reset thread and another thread updating the data of the contract.

While this solution has the disadvantages explained in the *Architecture* chapter it has the advantage of allowing a very fast data plane since counters are reset in the control plane and no additional synchronization is needed in the data plane.

4.3.4 Token Bucket implementation

The main challenge of this solution is guaranteeing the atomicity of update operations on the data of the token bucket, made difficult by the little amount of synchronization primitives available in the eBPF virtual machine.

The first considered implementation encompassed the periodic addition of tokens to the bucket in user space while leaving only the task of consuming tokens for every packet in the data plane, similarly to what has been done in the Fixed Window Counter. Unlike the reset of the counter however, the addition operation requires different steps that include two accesses to maps, and eBPF does not provide a synchronization method to make these user space operations atomic with respect to the data plane. This produces the following situation:

1. The number of tokens currently available in the bucket is read from the eBPF map.

2. The value is increased of the defined amount: while this operation is performed packets in the data plane continue to consume tokens and be forwarded.

3. The map is updated with the increased value, therefore overwriting all consume operations happened after the map reading.

This can cause an output rate that is higher than the expected one and the problem is more evident with higher input rates or a higher frequency of bucket refill operations.

To overcome this problem a solution relying only on the data plane for the update of the bucket has been adopted, leaving to the control plane the only responsibility of its initialization. The solution is only available starting from Linux kernel version 5.1, thanks to the introduction of spinlocks to handle concurrent access to eBPF maps.

Spinlocks

The spinlock is one of the most simple synchronization primitives, in which the thread needing to acquire it continuously check if the locking variable is available with a loop, performing what is called a *busy waiting*. The update operation on the variable holding the lock requires to be atomic and is therefore usually implemented with special assembly instructions such as atomic exchange or atomic test-and-set. Spinlocks have the advantage of having a very little overhead, since waiting threads doesn't need to be put in a waiting queue, but require the program holding the lock to release it in a short time. This makes them ideal for the use in eBPF programs.

To use spinlocks in eBPF program a field of type `struct bpf_spin_lock` must be added to the value of the protected map. The lock can then be acquired with the `bpf_spin_lock()` helper and released with `bpf_spin_unlock()`.

eBPF implementation of spinlocks comes with some restrictions:

- Only `HASH` and `ARRAY` maps support spinlocks.
- Only one lock can be acquired at a time, to avoid the risk of deadlocks.
- No other helper functions can be called while holding a lock.
- Maps using a spinlock must be annotated with BTF (BPF Type Format) to allow safety checks in the verification phase, therefore requiring compilation with LLVM 9 or later.
- Only kernel space programs can manipulate spinlocks, for access to map items in user space the spinlock just assures that read and update operations are atomic but doesn't provide an instrument to atomically perform complex operations.

Final solution

Since the execution of eBPF programs can not be periodically triggered but is event-based the refill of the bucket in the data plane must be performed upon packet reception. The adopted solution therefore requires to associate to each bucket the timestamp of its last refill, allowing to compute, for every packet, the time passed since the operation and the resulting number of tokens to add.

eBPF provides the `bpf_ktime_get_ns()` helper to get the number of nanoseconds elapsed since the boot of the system. This helper however, as shown in the

Evaluation chapter, has proven to have a non negligible overhead if compared to other operations of the data plane. To overcome this problem a manually managed clock has been adopted, stored in a single cell `PERCPU_ARRAY_MAP` and updated in user space every millisecond. This solution brings to a slight boost in performance without a measurable loss in precision.

The final structure used by the eBPF program to hold information about the bucket is the following:

```
struct bucket {
  u64 tokens;       // Number of tokens currently in the bucket
  u64 refill_rate;  // Refill rate of the bucket in tokens/ms
  u64 capacity;     // Capacity of the bucket
  u64 last_update;  // Timestamp of the last time the bucket
                    // was refilled in ms
};
```

Listing 4.7: Bucket data structure

The code applying rate limit to each packet, shown in listing 4.8, operates as follows:

1. Retrieve current timestamp from the `clock` map.
2. Acquire the spinlock of the bucket.
3. If the current timestamp is greater than the one of last refill compute the number of new tokens and add them to the bucket, making sure that the new number doesn't exceed the bucket capacity.
4. Try to consume a number of tokens corresponding to the size of the packet, if there aren't enough tokens in the bucket store the `RX_DROP` return action, otherwise `RX_OK`.
5. Release the spinlock.

```
int limit_rate(struct CTXTYPE *ctx, struct contract *contract) {
  int zero = 0;
  struct bucket *bucket = &contract->bucket;
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;

  u64 *clock_p = clock.lookup(&zero);
  if (!clock_p) {
    return RX_DROP;
  }
  u64 now = *clock_p;

  bpf_spin_lock(&contract->lock);

  if (now > bucket->last_refill) {
```

```
16      u64 new_tokens =
17          (now - bucket->last_refill) * bucket->refill_rate;
18      bucket->tokens += new_tokens;
19      if (bucket->tokens > bucket->capacity) {
20        bucket->tokens = bucket->capacity;
21      }
22      bucket->last_refill = now;
23    }
24
25    u64 needed_tokens = (data_end - data) * 8;
26    u8 retval;
27    if (bucket->tokens >= needed_tokens) {
28      bucket->tokens -= needed_tokens;
29      retval = RX_OK;
30    } else {
31      retval = RX_DROP;
32    }
33
34    bpf_spin_unlock(&contract->lock);
35
36    return retval;
37  }
```

Listing 4.8: Token bucket rate limit function

4.3.5 Sliding Window implementation

The implementation of the Sliding Window algorithm is similar to the one of the Token Bucket, requiring timestamping and the use of spinlocks to update the information of the window. Unlike the token bucket however the granularity of the timestamp can not be reduced, due to the lack of floating point support in the eBPF virtual machine. The transmission time of packets is computed with the ratio between the size of the packet and the maximum rate and could result in a fractional number, whose decimal part is discarded. Having a high time granularity guarantees that the this decimal part is negligible and doesn't impact the behaviour of the algorithm. As a consequence the `bpf_ktime_get_ns()` helper is used despite its overhead, since a clock can not be updated in user space with nanoseconds precision.

listing 4.9 shows the implementation of the algorithm:

```
1  struct window {
2    u64 start;  // Timestamp of window start in ns
3    u64 size;   // ns
4    u64 rate;   // maximum rate in bits/s
5  };
6  ...
7  int limit_rate(struct CTXTYPE *ctx, struct contract *contract) {
8    u8 retval;
```

```
9     struct window *window = &contract->window;
10    void *data = (void *)(long)ctx->data;
11    void *data_end = (void *)(long)ctx->data_end;
12
13    u64 tx_time = (data_end - data) * 8 * 1000000000 / window->rate;
14
15    u64 now = bpf_ktime_get_ns();
16
17    bpf_spin_lock(&contract->lock);
18
19    if (window->start + tx_time > now) {
20      retval = RX_DROP;
21    } else if (window->start + window->size < now) {
22      window->start = now - window->size + tx_time;
23      retval = RX_OK;
24    } else {
25      window->start += tx_time;
26      retval = RX_OK;
27    }
28
29    bpf_spin_unlock(&contract->lock);
30
31    return retval;
32  }
```

Listing 4.9: Sliding Window algorithm implementation

4.4 Traffic Classifier

The data model of the service is very simple and defines a list of traffic classes, each one identified by a `uint32` id and characterized by a mandatory priority (`uint32`) and the optional fields to identify a packet: the direction (`INGRESS`, `EGRESS` or `BOTH`) that defaults to `BOTH` if not specified, source and destination MAC addresses, the ethertype (`ARP` or `IP`), source and destination IP addresses, layer 4 protocol (`TCP`, `UDP` or `ICMP`) and source and destination ports.

In the control plane, the `Classifier` class allows to read, add and delete traffic classes and holds a `shared_ptr` to every configured class into an `unordered_map`. Every time a change in the configuration is applied its `updateProgram()` method (that will be explained in details in following sections) is called to update the classification eBPF program.

The `TrafficClass` class stores information of a single class of traffic, upon its initialization and every time an update is performed the `isValid()` method is used to check that compatible values are set for classification fields.

One of the main features of the service is the dynamic generation of the program implementing the Linear Bit Vector Search algorithm in the data plane, that allows

to support a wide range of header fields for classification while only injecting the code needed by the actual configuration.

To achieve this behaviour two components are used:

- A series of data plane templates used to compose the final data plane code.
- A data plane generation logic in the control plane, responsible of compiling and combining the templates.

4.4.1 Data plane templates

Three template files are used to implement the the classification steps shown in figure 4.2:

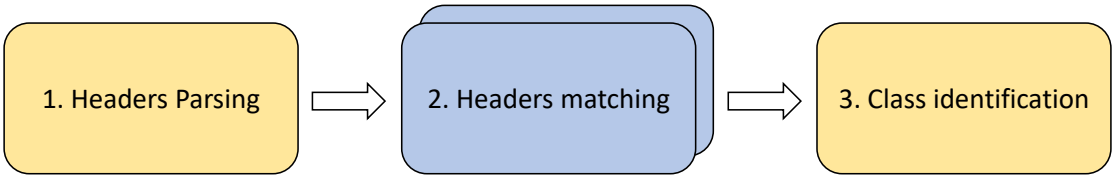

Figure 4.2: Classification steps.

Classifier_dp.c

This file contains the skeleton of the code of the data plane and implements point 1 and 3 of figure 4.2.

The bit-vector responsible of storing partial results across matching steps of the algorithm is declared as an array of 64 bit values and stored in a `PERCPU_ARRAY` map, as shown below:

```
1 struct bitvector {
2   u64 bits[_SUBVECTS_COUNT];
3 };
4 BPF_PERCPU_ARRAY(pkt_bitvector, struct bitvector, 1);
```

Listing 4.10: Packet bit-vector declaration

The parameter `_SUBVECTS_COUNT` defines the number of 64-bit subvectors needed build the complete bit-vector. The data structure is not stored as a local variable because of the limited stack size of 512 bytes in eBPF programs, that would allow to handle a maximum of less then 4096 classes, also taking into account the need to allocate other variables on the stack. The use of a `PERCPU_ARRAY` is an acceptable compromise since its access is very fast.

The initial part of the main function is responsible of parsing the headers of the packet and storing them in the local structure shown below.

```
struct pkt_headers {
  __be64 smac;
  __be64 dmac;
  __be16 ethtype;
  __be32 srcip;
  __be32 dstip;
      u8 l4proto;
  __be16 sport;
  __be16 dport;
};
```

Listing 4.11: Packet headers structure.

Two Boolean parameters, `_PARSE_L3` and `_PARSE_L4`, are used in compiler directives to dynamically enable or disable the compilation of the code responsible of parsing the layer 3 and layer 4 headers.

The packet bitvector is retrieved from the corresponding map and all bits are set to 1.

At this point the `_MATCHING_CODE` parameter allows the insertion of the code responsible of perform matching on the various header fields, whose details will be explained later.

After this step the packet bit-vector contains a bit set to one for every matching class. To identify the highest priority class the least significant bit must be found. Many CPUs have a dedicated instruction to quickly achieve this task that can be called with compiler built ins, such as `__builtin_ffs()` in GCC/Clang. Unfortunately the eBPF instruction set doesn't support this operation and to overcome this problem a *multiply and lookup* approach based on *de Bruijn sequences* [14] has been adopted.

```
int matching_class_index = -1;
u16 *matching_res;
for (int i = 0; i < _SUBVECTS_COUNT; i++) {
  64 bits = pkt_bv->bits[i];
  if (bits != 0) {
    int index = (int)(((bits ^ (bits - 1)) * 0x03f79d71b4cb0a89) >>
    58);
    matching_res = index64.lookup(&index);
    if (!matching_res) {
      return RX_DROP;
    }
    matching_class_index = *matching_res + i * 64;
    break;
  }
}
```

Listing 4.12: De Bruijn method.

A de Bruijn sequence of length n is cyclic sequence of n 0s and 1s where every substring of length $log_2(n)$ appears exactly once. For example with $n = 2$ a valid

sequence is 0110, that contains substrings 01, 11, 10, and 00 (wrapping). Assuming that the bit-vector x contains at least one bit set, the method operates as follows:

1. The least significant bit set to 1 is isolated performing a bitwise XOR between x and $x-1$ (an alternative is performing a bitwise AND between x and its two's complement).

2. A de Bruijn sequence of size n staring with $log_2(n)$ zeroes is chosen and used to apply the following hash function to y:

$$h(y) = (y * deBruijn) >> (n - log_2(n))$$

   Since y only contains one bit set to 1 in position i, the multiplication corresponds to left-shifting the de Bruijn sequence by i positions, that followed by the right shift allows to isolate the i^{th} de Bruijn substring. Since this substring is unique the hash function is free of conflicts.

   The sequence chosen for this use case, with $n = 64$, corresponds to value 0x03f79d71b4cb0a89.

3. The hash value is used to retrieve the corresponding position in the word, performing a lookup on an array of $log_2(n)$ elements. In this implementation an ARRAY_MAP of 64 elements is used to store the array and is filled in the control plane with the following values corresponding to the chosen de Bruijn sequence:

```
const uint16_t index64[64] = {
0,  47, 1,  56, 48, 27, 2,  60, 57, 49, 41, 37, 28, 16, 3,  61,
54, 58, 35, 52, 50, 42, 21, 44, 38, 32, 29, 23, 17, 11, 4,  62,
46, 55, 26, 59, 40, 36, 15, 53, 34, 51, 20, 43, 31, 22, 10, 45,
25, 39, 14, 33, 19, 30, 9,  24, 13, 18, 8,  12, 7,  6,  5,  63
};
```

Listing 4.13: index64 map content.

At this point, if an index has been found, the value is used to perform a lookup on the class_ids ARRAY_MAP to retrieve the ID of the corresponding class, that is then saved in the traffic_class field of packet metadata to be used by downstream cubes.

MatchingTable_dp.c

This template contains the code to declare maps used in matching steps. One map for every header field is declared, containing for every value specified in configuration the corresponding bit-vector of compatible classes.

An LPM_TRIE_MAP is used if the match is performed only on the prefix of the value (in the case of IP addresses), while a HASH_MAP is used if the match is exact. Following parameters are used:

- `_PREFIX_MATCHER`: whether matching is based on lpm trie or hash map.
- `_FIELD`: the name of the packet field to perform the match on, used to give unique names to the maps.
- `_TYPE`: the C type of the matching field, used to define the keys of the maps.
- `_CLASSES_COUNT`: the number of traffic classes, used to set the maximum size of maps.
- `_WILDCARD`: whether there is a wildcard to match on.

```
#if _PREFIX_MATCHER
struct _FIELD_lpm_key {
  u32 prefix_len;
  _TYPE key;
};
BPF_LPM_TRIE(_FIELD_rules, struct _FIELD_lpm_key, struct bitvector,
             _CLASSES_COUNT);
#else
BPF_HASH(_FIELD_rules, _TYPE, struct bitvector,
         _CLASSES_COUNT + 1);
#if _WILDCARD
BPF_ARRAY(_FIELD_wildcard_bv, struct bitvector, 1);
#endif
#endif
```

Listing 4.14: MatchingTable_dp.c template.

Matcher_dp.c

This template contains the logic to perform a match on a single header field. The same parameters of the *MatchingTable_dp.c* template are used, plus the `_SUBVECTS_COUNT` parameter to define bounded loops (the only ones supported in eBPF) on the bit-vector and `_DIRECTION` for debug purposes.

The code of the template performs the lookup of the configured field in the corresponding map to retrieve the bit-vector. In case the lookup fails and a wildcard is present, its bit-vector is read from the corresponding `ARRAY_MAP`. If no wildcard is available, code `RX_OK` is returned and the packet proceeds without being classified. In case of success a bitwise `AND` is performed between the retrieved bit-vector and the one associated to the packet. If the result has all bits set to zero the classification is stopped, otherwise the packet proceeds to the next matching field or to the final phase.

4.4.2 Data plane generation logic

The hierarchy of C++ template classes shown in figure 4.3 has been implemented to support the generation of matching code. A specialization of these templates is instantiated on service initialization (in the constructor of the `Classifier` class) to handle matching on every header field. The use of templates (both in classes and in data plane code) allows to to generate a logic that is not bound to specific fields an therefore provides an easy way for future introduction of additional fields, such as the ones of application layer headers.

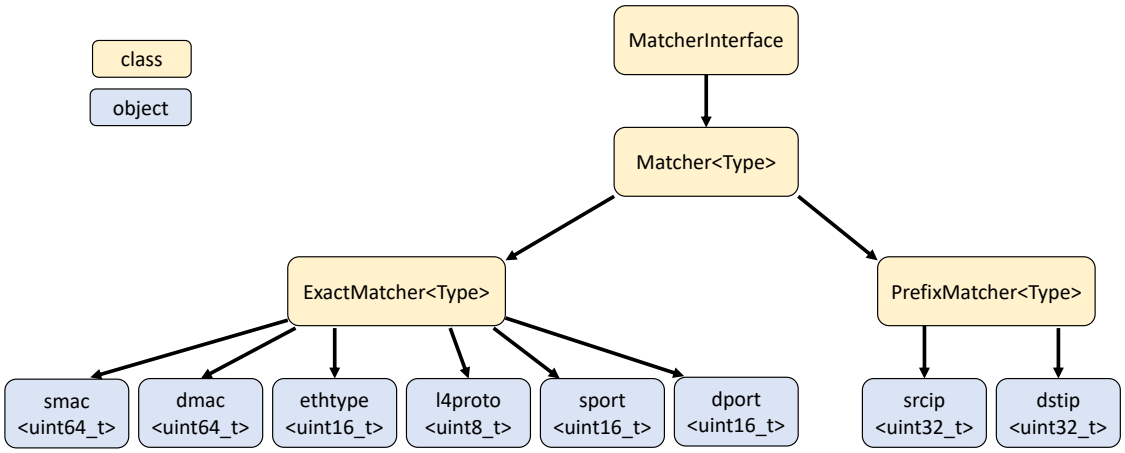

Figure 4.3: *Matchers* hierarchy.

Two types of *matchers* have been defined:

- `ExactMatcher`: allows to configure code to perform an exact match on a field, relying on a `HASH_MAP`.
- `PrefixMatcher`: allows to perform matching only on the initial part of a field, usign a `LPM_TRIE_MAP`.

`MatcherInterface` and `Matcher` classes define the methods common to all *matchers*:

- `initBitvector(uint32_t size)`: re-initializes the *matcher* with a clean bitvector (all bits set to 0), discarding all traffic classes formerly configured.
- `getTableCode()`: returns the table declaration code obtained compiling the *MatchingTable_dp.c* template with the current configuration of the matcher.
- `getMatchingCode()`: returns the matching code obtained compiling the *MatchingCode_dp.c* template.

- `isActive()`: tells whether a matching step on the represented field is needed (i.e. if at least one specific value is set).

- `appendWildcardBit()`: appends a 1 bit to the wildcard bit-vector and, as a consequence, also to bit-vectors of all other values currently configured.

- `loadTable(...)`: loads configured bit-vectors in the corresponding eBPF maps. For the exact matching type the bit-vector of the wildcard is loaded in a dedicated single-item `ARRAY_MAP`, while for prefix matching a \0 entry in the *lpm trie* is used.

Following methods are specific to the `ExactMatcher` class:

- `appendValueBit(T value)`: appends a 1 bit to the bit-vector of the given value while a 0 bit is left in all other bit-vectors (including the wildcard one). If the value is configured for the first time its bit-vector its initialized from the wildcard.

- `appendValuesBit(std::vector<T> values)`: allows to append a 1 bit to bit-vectors of multiple values (used for example when a class can match on bot TCP and UDP protocols).

Following method is specific to the `PrefixMatcher` class:

- `appendValueBit(T value, uint32_t prefix_len)`: appends a 1 bit to the bit-vector of the given prefix. Since the lpm trie performs a longest prefix match but any prefix match is valid in this use case, the bit must also be added to longer prefixes that include the configured one.

The process of updating the data plane requires, beside reloading the eBPF program, to clear and fill the maps, since insertion or removal of a single class implies the update of all bit-vectors. While program reload is an atomic operation, filling the maps can bring to a transitory time in which the service behaves into an undefined way. To overcome the problem two classification programs for every direction are used, and while the active one is processing packets the new one is configured. The entry point of the pipeline is moved to a *selector* program (*Selector_dp.c*) that allows either to switch between these two programs with a tail call or to disable the classification.

The `updateProgram(ProgramType direction)` method of `Classifier` allows to inject the updated classification program for the given direction and operates by the following steps:

1. Classes for the given direction are selected.

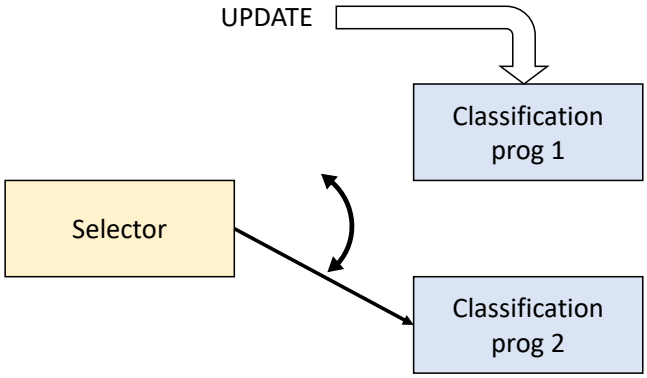

Figure 4.4: Classification program selection.

2. If no classes are found the *selector* program is updated to disable the classification, if a classification program is present it is deleted and the function ends.

3. Bit-vectors of all *matchers* are reset.

4. Classes are sorted by decreasing priority.

5. Classes are scanned in the given order and for every matching field specified by the class a 1 bit is appended to the bit-vector of the desired value in the corresponding *matcher*.

6. Classification code is generated retrieving map declaration and matching code from active *matchers* and inserting them in the *Classifier_dp.c* template. The code is injected as a new eBPF program.

7. Maps of the new program are populated.

8. The *selector* program is updated to point to the new program and the old one is deleted.

Chapter 5

Evaluation

This chapter presents a set of tests performed to evaluate the performance and feasibility of the proposed solution.

First, an overview of the benchmarking tools used and the testing methodologies is provided. Then a comparison of the different rate limit algorithms is performed, in order to chose the best solution taking into account both precision and cost. The performance of the pipeline is then compared with alternative solution based on different technologies, and its scalabilty is studied varying the complexity of the configuration and the number of CPU cores assigned to computation. In the end a micro-benchmark is presented in order to highlight the cost introduced by single modules composing the pipeline and to spot possible weak points and space for improvement.

The testbed is composed by two physical machines connected with two direct links using dual-port Intel XL710 40Gbps NICs. One machine operates as DUT (Device Under Test), executing the pipeline under test and forwarding packets between its interfaces, the other acts as tester, generating test packets on one interface and measuring processed traffic on the other one.

Machines are configured as follows:

- *DUT*: Intel Xeon Gold 5120 @2.60GHz processor with 14 cores (hyper-threading disabled) and 19,25 MB of L3 cache, 64 GB of DRAM and Ubuntu 18.04.4 LTS. Different versions of the kernel have been used according to the pipeline under test and will be specified in following sections.

- *Tester*: Intel Xeon E3-1245 v5 @3.50GHz processor with 4 cores (plus hyper-threading) and 8 MB of L3 cache, 32 GB of DRAM and Ubuntu 18.04.4 LTS with kernel 5.0.

5.1 Benchmarking tools

5.1.1 MoonGen

MoonGen [15] is a flexible high-speed packet generator that can saturate a 10 GbE link with minimum-sized packets while using only a single CPU core on commodity hardware. It relies on the Intel DPDK framework to achieve high speed packet processing and uses the Lua scripting language compiled with the LuaJIT compiler to provide the possibility to customize the packet generation logic.

MoonGen's architecture is shown in figure 5.1. Its core is represented by a Lua wrapper for DPDK that provides utility functions for packet generation and an API to configure hardware-related features like timestamping and rate control.

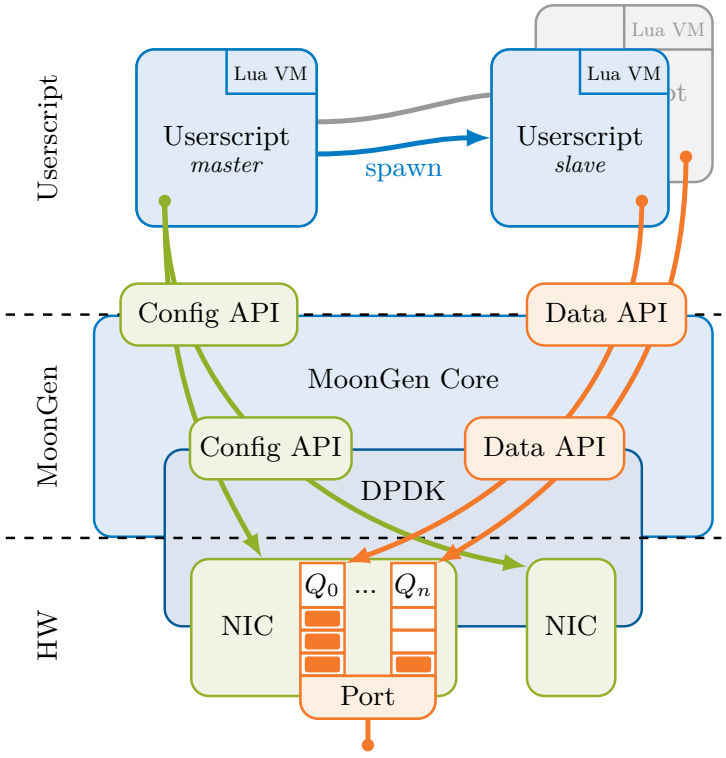

Figure 5.1: MoonGen architecture.

A user-provided script, the userscript, is used to describe the packet generation logic. A master function initializes the used NICs, i.e. the number of hardware queues, buffer sizes and filters for received traffic, and then spawns a set of slave tasks, each one executed in an independent LuaJIT VM that is pinned to a CPU core. These tasks usually receive a hardware queue as an argument and contain the loop responsible of transmitting or receiving packets via this queue.

MoonGen allows to obtain precise rate control, either by exploiting the rate

liming function embedded into supported NICs or via software. In this case invalid packets with a wrong CRC are used to evenly distribute valid packets and avoid micro-bursts. Invalid packets are then discarded by the NIC of the receiving host without any impact on the CPU.

The tool can also exploit timestamping capabilities of supported NICs, originally intended for clock synchronization across networks, to measure latencies with sub-microsecond precision.

5.1.2 TIPSY

TIPSY (Telco pIPeline benchmarking SYstem) [16] is a benchmark suite to evaluate and compare the performance of programmable data plane technologies and network-function virtualization frameworks over a set of standard scenarios rooted in telecommunications practice.

The tool provides a set of pipelines of increasing complexity, starting from a simple L2 forwarder (a switch) and going up to the 5G Mobile Gateway. These pipelines are implemented with different data plane technologies, such as OvS, BESS, t4p4, Lagopus, etc. The system has been extended adding the support for Polycube-based eBPF/XDP pipelines. Currently only the Mobile Gateway and the Port Forward (simple forwarding of packets between two ports) are available.

The setup needed to run a benchmark is show in figure 5.2 and reflects the one described at the beginning of the chapter. In relation to how the traffic flows, TIPSY distinguishes between the uplink direction (user-to-network direction) and the downlink direction (network-to-user direction). An additional management connection relying on SSH is used by the tester to configure the DUT.

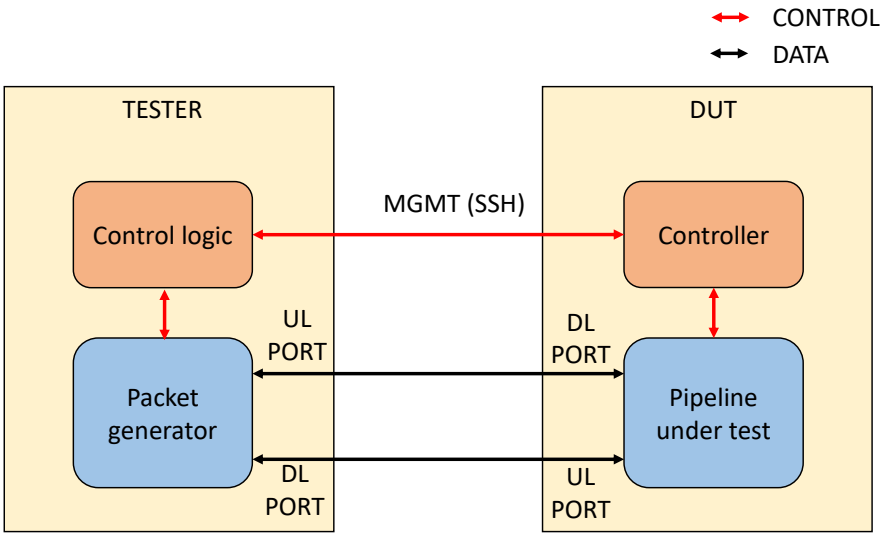

Figure 5.2: TIPSY setup.

TIPSY allows to run a completely automated benchmark starting from a JSON configuration file. The control logic on the tester drives all the benchmarking process shown in the control flow of figure 5.3: it parses the high level benchmark description and generate detailed pipeline configuration (e.g. IP address and base station of every user equipment) and a PCAP traffic trace, it connects to the DUT and instructs the controller (either a simple script or an SDN controller for OpenFlow based solution) on how to configure the pipeline, it uses the packet generator to replay the trace and then collects the results.

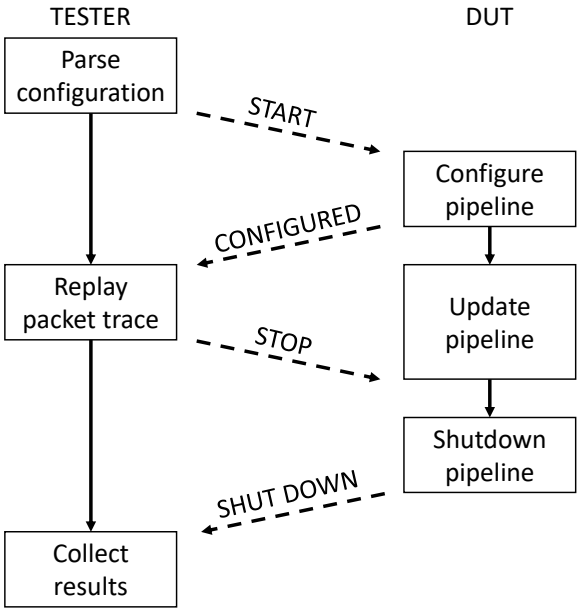

Figure 5.3: TIPSY control flow.

Every benchmark is described by four sections:

- `pipeline`: this section specifies which is the pipeline under test (`name` property) and how it is configured. Fields available for the configuration vary based on the selected pipeline. For the Mobile Gateway (`mgw`) they include:
  - IP and MAC addresses of the gateway.
  - The number of user devices (`user`), base stations and routes toward remotes networks on the PDN (`server`).
  - The number of cores dedicated to the pipeline.
  - The rate limit applied to the traffic of the user equipment (in bits/s).
  - A set of dynamic updates to run during test execution, expressed in updates per second: addition and removal of user devices and remote routes, handover operations (user equipment moving from one base station to another).

- `traffic`: defines parameters of the packet trace sent to the DUT, including the number o packets, their size, the direction of the traffic (uplink or downlink) and the encapsulation type (VXLAN and GTP are available) in case of pipelines handling tunneled traffic.

  The packet trace is generated using the *Scapy* library according to the provided configuration, randomly choosing source and destination parameters for every packet among the available set of users and servers. The `random-seed` parameter can be used to produce the same trace across different tests.

- `sut`: defines the parameters of the system under test (alias of DUT), like its hostname, needed by the tester to connect to it through SSH, the id of uplink and downlink ports and the backend used to implement the pipeline.

- `tester`: Defines the parameters of the tester, including uplink and downlink ports, the duration of the test, the number of cores to be used by the packet generator and the type of test. Two types are currently available, both relying on the MoonGen packet generator:

  - `moongen`: the packet trace is repeatedly replayed to the DUT either at the given rate or at the maximum rate supported by the packet generator, in order to test the average bandwidth handled by the pipeline. Additional packets are periodically sent to measure latencies. Unfortunately this last feature doesn't work with the Mobile Gateway pipeline, since MoonGen requires the packet to be unmodified to compute its processing time, but tunneling breaks this mechanism.
  - `moongen-rfc2544`: this kind of test allows to measure the maximum throughput that can be obtained by the pipeline according to criteria expressed in RFC2544. A binary search is performed, gradually adjusting the rate at which the trace is replayed in order to find the maximum throughput that produces a packet loss lower than the configured one (`loss-tolerance` parameter).

Multiple values can be set for every parameter (using a JSON array). In this case an additional option (`scale`) allows to specify how these values should be scaled in order to produce multiple tests, either performing the outer product, obtaining all possible combination, or scaling them jointly.

An example of complete benchmark configuration is shown in listing 5.1.

```
1 "pipeline": {
2     "bst": 1,
3     "core": 1,
4     "fakedrop": false,
5     "fluct-server": 0,
6     "fluct-user": 0,
```

```
7       "gw-ip": "200.0.0.1",
8       "gw-mac": "aa:22:bb:44:cc:66",
9       "handover": 0,
10      "name": "mgw",
11      "nhop": 2,
12      "rate-limit": 40000000000,
13      "server": 10,
14      "user": 100
15  },
16  "scale": "joint",
17  "sut": {
18      "coremask": "0xff",
19      "downlink-port": "0000:65:00.0",
20      "hostname": "dut-hostname",
21      "tipsy-dir": "/home/test/tipsy",
22      "type": "ovs",
23      "uplink-port": "0000:65:00.1"
24  },
25  "tester": {
26      "core": 1,
27      "downlink-port": "1",
28      "moongen-cmd": "/home/test/MoonGen/build/MoonGen",
29      "rate-limit": 3000,
30      "test-time": 60,
31      "type": "moongen",
32      "uplink-port": "0"
33  },
34  "traffic": {
35      "conf": "pipeline.json",
36      "dir": "downlink",
37      "pkt-num": 1000,
38      "pkt-size": 60,
39      "random-seed": 1,
40      "tunneling-method": "vxlan"
41  }
```

Listing 5.1: Example of Mobile Gateway benchmark configuration

5.2 Rate limit algorithms comparison

This set of tests aims to compare the proposed rate limit algorithms in order to find a good compromise between precision and processing overhead. The *Helloworld* service available in Polycube has been used to forward packets between the two interfaces of the DUT and the Policer has been attached to one of the ports to provide rate limit, assigning a single core of the DUT to packet processing.

5.2.1 Precision

In the precision test the packet generator has been configured to send packets at the maximum rate achievable with a single core, both using 64 bytes frames, producing around 22 Mpps, and 1518 bytes frames, producing about 3 Mpps. The Policer service has been configured with an increasing rate limit, starting from 100 Kbps up to 1 Gbps, and for algorithms allowing it a burst limit of $1/100^{\text{th}}$ of the desired rate has been set, in order to prevent the initial burst from having an impact on the measured average rate.

figure 5.4 and figure 5.5 show the percentage of error of the output rate with respect to the desired one, for both 64 and 1518 frame size.

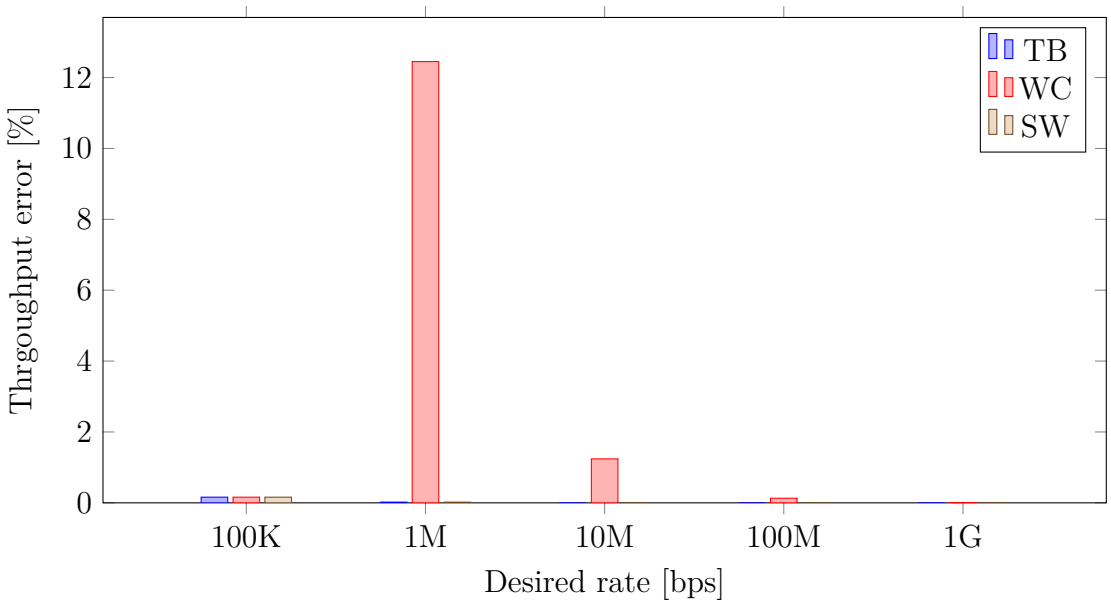

Figure 5.4: Rate limit algorithms precision with 64 bytes frames.

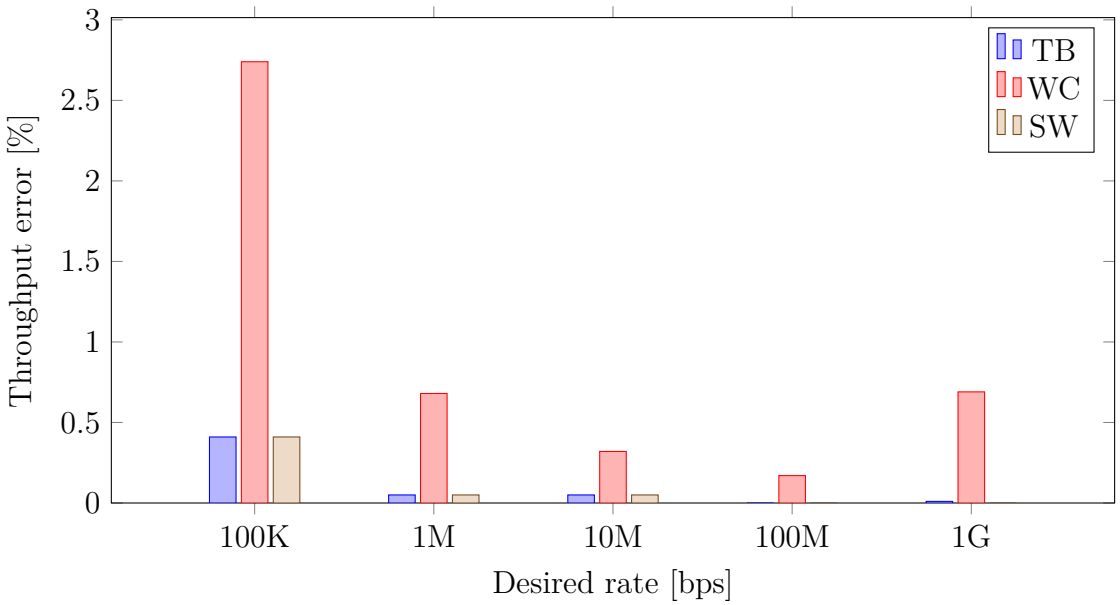

Figure 5.5: Rate limit algorithms precision with 1518 bytes frames.

Results show that the *Token Bucket* (TB) and *Sliding Window* (SW) present a similar behaviour, with a high precision rate limiting and an error that is almost constant in absolute terms, therefore becoming negligible when the desired rate grows. The *Fixed Window Counter* (WC) on the other hand has an average lower precision than other solutions with an irregular behaviour and an error that can exceed 10% in some cases. The reason of these results is not fully clear at the moment, but it is worth noticing that the error of the *Fixed Window Counter* decreases as the input rate gets closer to the desired one, while other solutions are not affected by this parameter.

5.2.2 Overhead

This test aims to evaluate the cost introduced by proposed rate limit solutions. To do this the *Policer* has been configured with a high rate limit (40 Gbps) in order not to influence the output rate, and an RFC2544-like measurement has been performed, obtaining the maximum rate that can be handled by the function with at most 1% packet drop rate. A baseline is also provided, configuring the `PASS` action on the *Policer*.

figure 5.6 show the results. The *Window Counter* (WC) solution has the minimum cost, allowing to achieve a throughput almost equal to the baseline, and this is due to its very simple data plane implementation, that just requires to decrease the counter stored in a map. The *Token Bucket* (TB) on the other hand has to perform a greater number of operation in the data plane, requiring both to add and consume token from the bucket. Most of the overhead however is due to the use of spinlocks

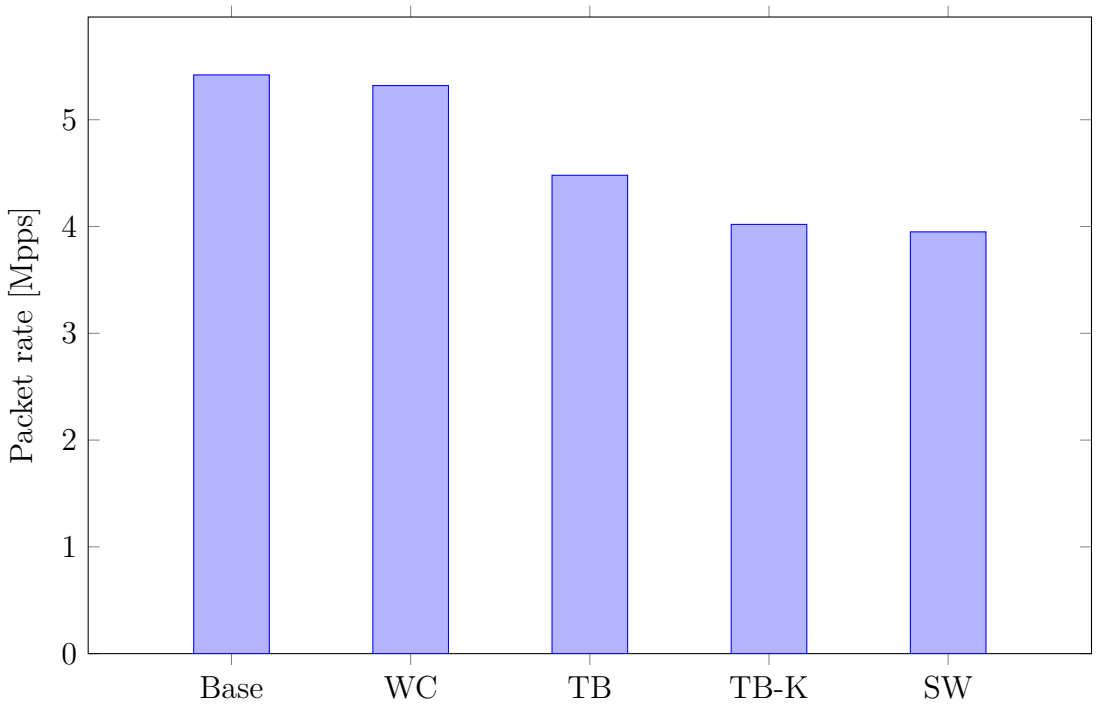

Figure 5.6: Rate limit algorithms overhead.

to manage concurrent access to data structures. The TB-K column shows an alternative implementation of the TB algorithm that uses the `bpf_ktime_get_ns()` helper to retrieve the timestamp, showing the advantage of a user space updated clock. The solution with poorest performance is the *Sliding Window* (SW), having to rely both on spinlocks and on the `ktime` helper.

The *Token Bucket* solution has been chosen for following tests on the whole Gateway pipeline, representing a good compromise between precision and performance, and allowing to configure both rate and burst limits, like other solutions (such as OvS meters) do.

5.3 Mobile Gateway performance

This set of tests evaluates the performance of the Mobile Gateway prototype and studies its scalability with an increasingly complex configuration of the service and with the number of cores dedicated to packet processing.

To provide some context the same tests have been run on equivalent pipelines available in the TIPSY repository and implemented with different technologies: BESS (Berkeley Extensible Software Switch) and OvS (v2.13.0), both in its user space, DPDK based, implementation and in its in-kernel mode. The eBPF version of the Gateway has been executed with kernel version 5.6, in order to take

advantage of the last eBPF features and performance improvements, while other implementation has been run on kernel version 5.0 using DPDK version 19.10 when needed. A preliminary test shows that packet size does not impact the resulting throughput. As a consequence following benchmarks has been performed using minimum sized packets. In the downlink direction this corresponds to 64 bytes. In uplink the encapsulation has to be taken into account: OvS and BESS pipelines use VxLAN due to the lack of GTP support in the framework, resulting in minimum sized encapsulated packets of 112 bytes, while the GTP encapsulation in the eBPF implementation produces 98 bytes packets.

5.3.1 Multiple users scalability

In this test the number of user devices has been scaled form 1 to 3000, contemporary configuring one base station every 100 users and 1 remote route on the PDN every 10 users. Both the uplink and the downlink directions have been tested taking RFC2544-like measurements, and using a packet trace composed of an average of 10 UDP flows per user.

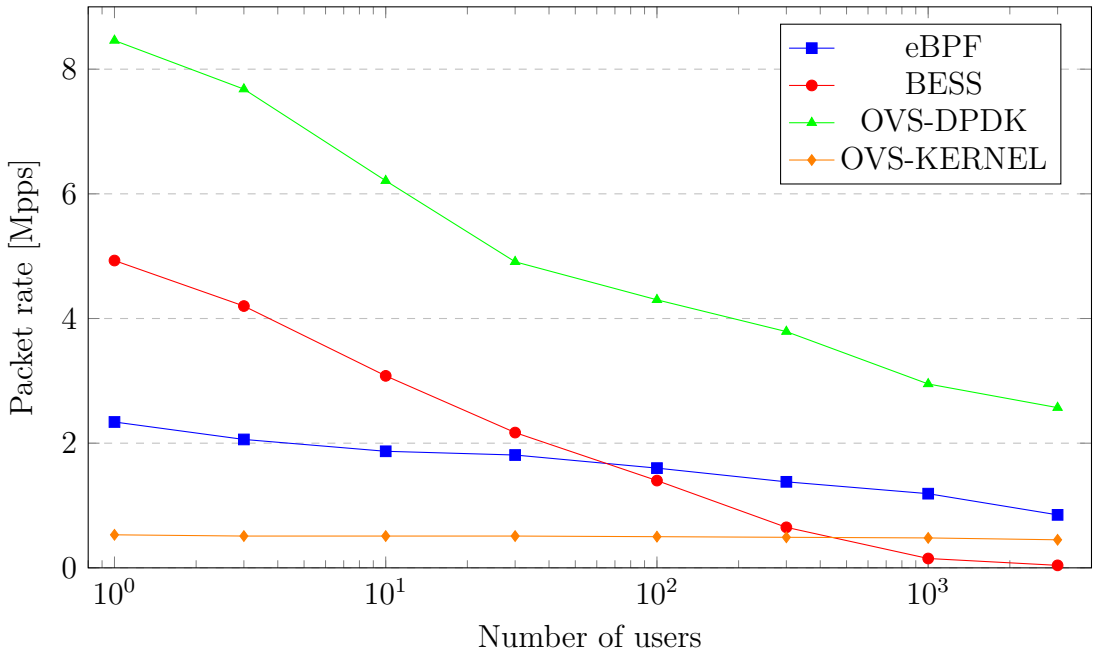

Figure 5.7: Multiple users scalability in the downlink direction.

Results of figure 5.7 and figure 5.8 show that the eBPF pipeline greatly outperforms its in-kernel counterpart. In relation to user space solutions eBPF can't reach the high packet processing speed of DPDK when the number of users is low. The situation changes when this number grows above 100. BESS shows a very poor scalability and its throughput quickly drops below that of in-kernel solutions.

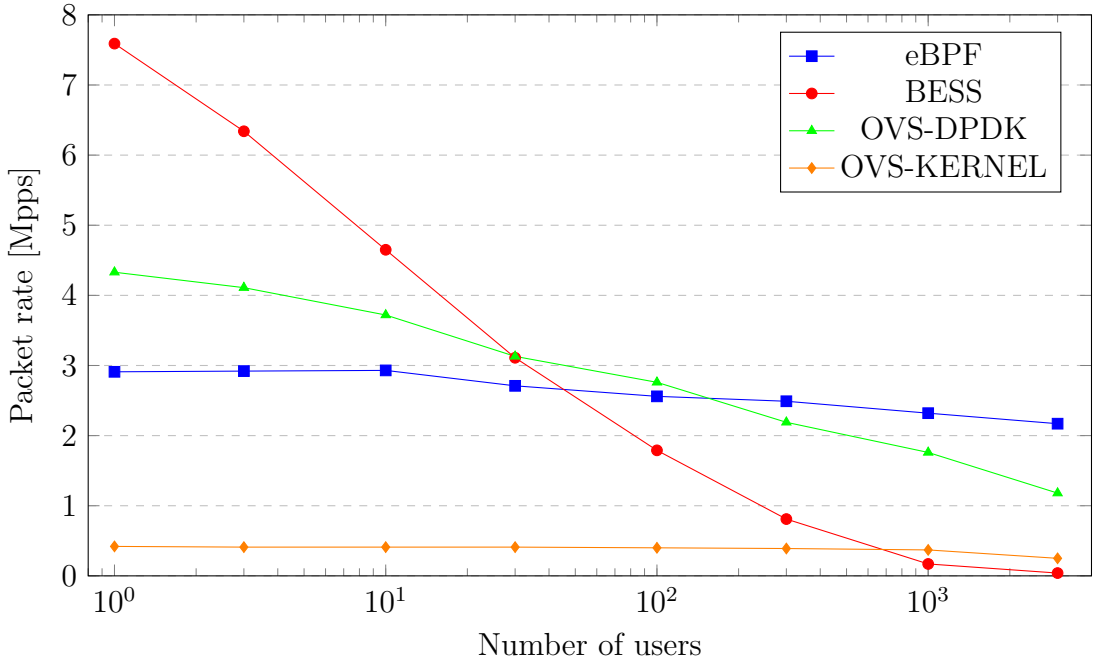

Figure 5.8: Multiple users scalability in the uplink direction.

OvS is greatly influenced by the direction of the traffic: while in downlink it maintains a high number of processed packets per second, in the uplink direction it is outperformed by the eBPF solution.

5.3.2 Multiple cores scalability

This test shows the ability of the solution to take advantage of parallel execution. The number of cores assigned to packet processing has been scaled from 1 to 8 adding 100 user devices, 1 base station and 10 remote routes for every new core, in order to have constant per-core load. As in the preceding test measurements have been taken in RFC2544 mode with a packet trace containing an average of 10 flows per user. In this specific scenario having a high number of flows is fundamental to allow the NIC to evenly distribute the traffic among different cores.

Results of figure 5.9 and figure 5.10 show that the multi-core scalability of the prototype is in line with the one of other solutions.

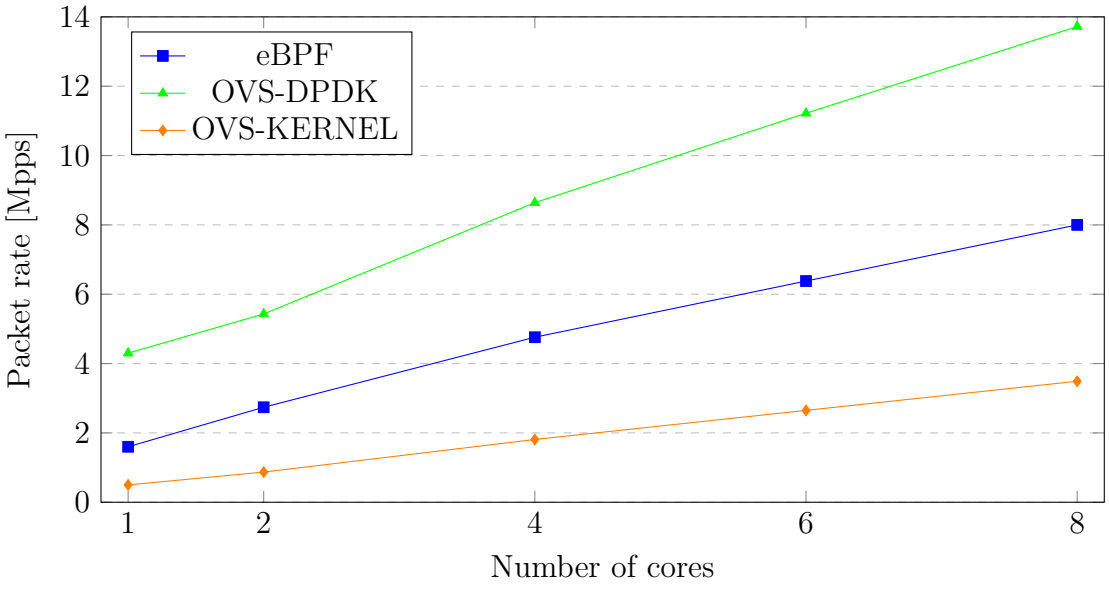

Figure 5.9: Multiple cores scalability in the downlink direction.

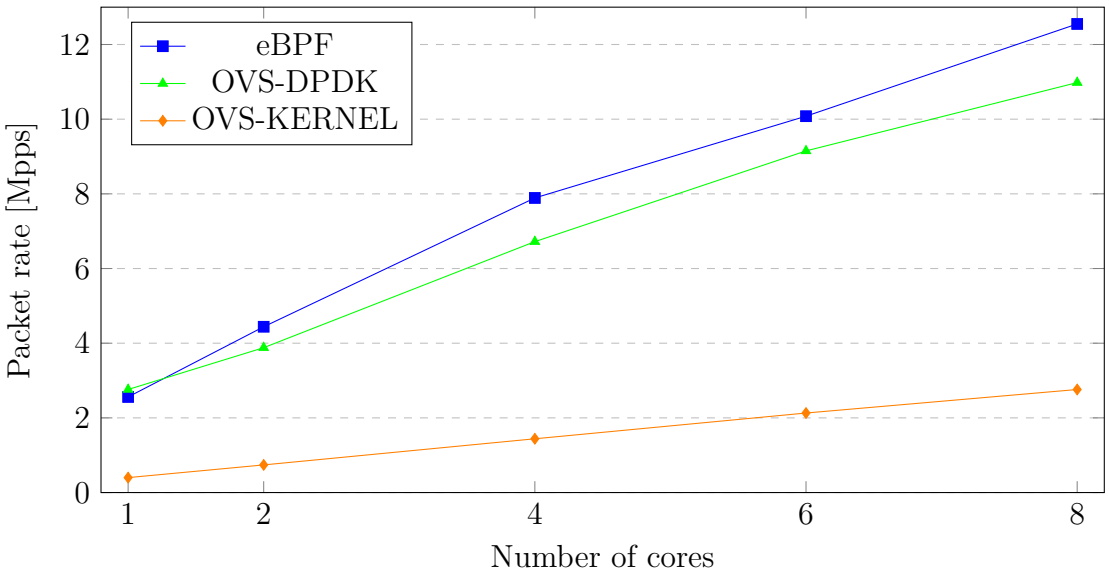

Figure 5.10: Multiple cores scalability in the uplink direction.

5.3.3 Modules overhead

This last test studies the impact that every module has on overall performance. The pipeline has been benchmarked in the downlink direction, starting only with the Router and then gradually adding the Classifier, the Policer and the GTP Handler. The results have been taken both with a simple configuration (single user, remote

route and base station) and with a complex one (1000 users, 100 remote routes and 10 base stations), in order to highlight the influence of the configuration on every module.

figure 5.11 shows the average time needed to process every packet in function of the modules inserted into the pipeline.

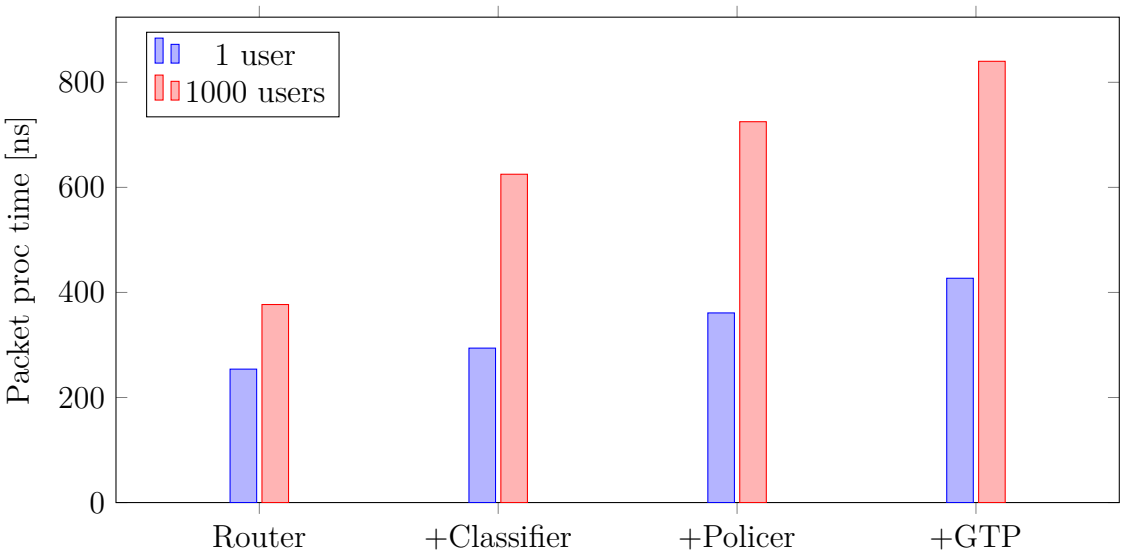

Figure 5.11: Overhead of modules on per-packet processing time

Results show that with a configuration of little complexity all modules present a similar overhead, with the Classifier having the lowest one. Things change when the number of users increases: in this scenario the Classifier becomes the more resource hungry component, introducing more than double the overhead of other modules. The reason of this behaviour is related to the algorithm chosen for classification, the Linear Bit Vector Search, that has linear cost on the number of classification rules. This is confirmed by the results presented in section 5.3.1, where the eBPF gateway shows a better scalability in the uplink direction, where classification is not required.

Chapter 6

Conclusions and future work

This thesis has presented a prototype of high-performance 5G Mobile Gateway running into the Linux kernel thanks to the use of eBPF/XDP, showing how this technology can be an interesting alternative for the implementation of telco data plane network functions and can be successfully leveraged in the deployment of the upcoming 5G mobile network.

A modular architecture has been proposed, composed by four basic components: a GTP Handler, a Traffic Policer, a Traffic Classifier and a Router. The chain implements a subset of the functionalities of a full Gateway but thanks to the modular approach can be easily be extended with new functions with low effort. Modules has been designed with the aim of being as general as possible and can therefore be reused to build different pipelines.

In the implementation phase different challenges have been faced due to the limitations imposed by the eBPF execution environment, including the event-based execution of programs, the limited number of synchronization mechanisms and the restricted access to memory. Solutions and workarounds to this problems have been found adopting different techniques for rate limiting and an efficient traffic classification algorithm.

The evaluation of the prototype shows that an eBPF-based Mobile Gateway is able to outperform other in-kernel packet processing solutions (OvS) and can compete with kernel bypass data plane technologies such as DPDK. While providing higher raw packet processing speeds, DPDK is less flexible than eBPF, since it requires a rigid partitioning of the resources of the machine it runs on. When enabled, DPDK acquires complete control over the NIC bypassing the kernel and, as a consequence, applications based on it can not leverage the consolidated Linux networking stack. Additionally this technology requires a set of cores to be exclusively dedicated to packet processing, while its polling of the NIC for new packets causes these cores to always run at 100%. This requirements may be too restrictive in a scenario like Edge Computing, composed of a great number of small, distributed

data centers, where servers have to run both data plane functions and more traditional cloud native applications, like control plane functions and user defined jobs. In these case eBPF could represent a more appealing solution, allowing more elastic sharing of the resources of the system and a higher level of integration with the Linux kernel.

The source code of the modules developed during this work is open source and available online in the Polycube repository[1]. A poster named "*A Proof-of-Concept 5G Mobile Gateway with eBPF*" has also been submitted to the 2020 SIGCOMM *Call for Posters, Demos, and Student Research Competition*.

One possible future work is the extension of the modules of the pipeline to implement additional features of the Mobile Gateway, like deep packet inspection and charging support. Another interesting aspect is the possibility to exploit the run-time injection capabilities of eBPF to achieve a dynamic optimization of the pipeline, based on the current global configuration, in order to reach better performance. A cross-modules optimization mechanism should also be studied, in order to improve the performance while maintaining an high level modular representation of the pipeline.

[1] https://github.com/polycube-network/polycube/tree/mobile-gateway

Bibliography

[1] Mansoor Shafi et al. "5G: A tutorial overview of standards, trials, challenges, deployment, and practice". In: *IEEE journal on selected areas in communications* 35.6 (2017), pp. 1201–1221.

[2] Faqir Zarrar Yousaf et al. "NFV and SDN—Key technology enablers for 5G networks". In: *IEEE Journal on Selected Areas in Communications* 35.11 (2017), pp. 2468–2478.

[3] *5G; System architecture for the 5G System (5GS)*. Tech. rep. 3GPP TS 23.501. Version 15.9.0 Release 15. ETSI, 2020.

[4] The Cilium Authors. *BPF and XDP Reference Guide*. URL: `https://docs.cilium.io/en/v1.8/bpf/`. (accessed: 06.2020).

[5] Toke Høiland-Jørgensen et al. "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.

[6] Sebastiano Miano et al. "A Service-Agnostic Software Framework for Fast and Efficient in-Kernel Network Services". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–9.

[7] Balázs Pinczel et al. "Towards high performance packet processing for 5G". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. 2015, pp. 67–73.

[8] Suneet Kumar Singh et al. "Offloading Virtual Evolved Packet Gateway User Plane Functions to a Programmable ASIC". In: *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*. 2019, pp. 9–14.

[9] Tamás Lévai et al. "The price for programmability in the software data plane: The vendor perspective". In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2621–2630.

[10] Nikrad Mahdi. *An alternative approach to rate limiting*. 2017. URL: https://www.figma.com/blog/an-alternative-approach-to-rate-limiting/. (accessed: 06.2020).

[11] Quentin Monnet. *Stateful packet processing: two-color token-bucket PoC in BPF*. URL: https://github.com/qmonnet/tbpoc-bpf. (accessed: 06.2020).

[12] Sebastiano Miano et al. "Securing Linux with a faster and scalable iptables". In: *ACM SIGCOMM Computer Communication Review* 49.3 (2019), pp. 2–17.

[13] TV Lakshman and Dimitrios Stiliadis. "High-speed policy-based packet forwarding using efficient multi-dimensional range matching". In: *ACM SIGCOMM Computer Communication Review* 28.4 (1998), pp. 203–214.

[14] Charles E Leiserson, Harald Prokop, and Keith H Randall. "Using de Bruijn sequences to index a 1 in a computer word". In: *Available on the Internet from http://supertech. csail. mit. edu/papers. html* 3.5 (1998).

[15] Paul Emmerich et al. "Moongen: A scriptable high-speed packet generator". In: *Proceedings of the 2015 Internet Measurement Conference*. 2015, pp. 275–287.

[16] The TIPSY Authors. *TIPSY: Telco pIPeline benchmarking SYstem*. URL: https://github.com/hsnlab/tipsy. (accessed: 06.2020).