UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Thiago Arruda Navarro do Amaral

Run-time Adaptive In-Kernel BPF/XDP Solution for 5G UPF: Design, Prototype and Performance Evaluation

Solução BPF/XDP Adaptável em Tempo de Execução no Kernel para 5G UPF: Projeto, Protótipo e Avaliação de Desempenho

Campinas

2022

Thiago Arruda Navarro do Amaral

Run-time Adaptive In-Kernel BPF/XDP Solution for 5G UPF: Design, Prototype and Performance Evaluation

Solução BPF/XDP Adaptável em Tempo de Execução no Kernel para 5G UPF: Projeto, Protótipo e Avaliação de Desempenho

Dissertation presented to the Faculty of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering.

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Campinas

2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

Am15r

Amaral, Thiago Arruda Navarro, 1988-
Run-time adaptive in-kernel BPF/XDP solution for 5G UPF: design, prototype and performance evaluation / Thiago Arruda Navarro do Amaral. – Campinas, SP : [s.n.], 2022.

Orientador: Christian Rodolfo Esteve Rothenberg.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Comutação por pacotes (Transmissão de dados). 2. Linux. 3. Gateways (Redes de computadores). 4. Redes definidas por software (Tecnologia de rede de computadores). I. Esteve Rothenberg, Christian Rodolfo, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Solução adaptável em tempo de execução no kernel para 5G UPF: projeto, protótipo e avaliação de desempenho
Palavras-chave em inglês:
Packet switching
5G system
Linux
Gateways (Computer network)
Software-defined networking (Computer network technology)
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Marcos Augusto Menezes Viera
Cristiano Bonato Both
Data de defesa: 16-03-2022
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-4010-2039
- Currículo Lattes do autor: http://lattes.cnpq.br/4663642039232130

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato(a): Thiago Arruda Navarro do Amaral RA: 159121
Data de defesa: 16 de Março de 2022
Título da Tese: "Run-time Adaptive In-Kernel BPF/XDP Solution for 5G UPF: Design, Prototype and Performance Evaluation"

To my wife, Giovana, who created a clean path until the end of this journey.

Acknowledgements

This work was supported by grants 2018/23101-0 and 2020/05182-3 from the São Paulo Research Foundation (FAPESP).

“I continuously go further and further learning about my own limitations, my body limitation, psychological limitations. It’s a way of life for me.”
(Ayrthon Senna)

Resumo

A infraestrutura de computação de borda pode ser dimensionada de datacenters a um único dispositivo. A tecnologia mais conhecida para processamento rápido de pacotes é DPDK, que possui excelente desempenho em relação ao throughput e latência. Porém, existem algumas desvantagens quando o uso é feito na borda: *(i)* o mecanismo de polling para processamento de pacotes mantém a CPU ocupada exclusivamente mesmo que não haja tráfego, levando ao desperdício de recursos; e *(ii)* a interface DPDK torna-se indisponível para os aplicativos dentro do host, então a integração entre um aplicativo não DPDK e um aplicativo DPDK torna-se uma tarefa difícil. Neste trabalho, propomos uma solução 5G UPF de código-fonte aberto baseada no 3GPP Release 16 para ser implantada em um ambiente restritivo como o MEC, onde o host MEC e o UPF são colocados com a Estação Base, compartilhando os mesmos recursos computacionais e de rede. A solução aproveita o BPF / XDP, uma nova tecnologia de kernel Linux para processamento rápido de pacotes. Mostramos que ele pode escalar e atingir aproximadamente 10/11 Mpps usando apenas 60% da CPU com 6 núcleos.

Palavras-chaves: 5G; UPF; XDP; BPF.

Abstract

The edge computing infrastructure can scale from datacenters to a single device. The well-known technology for fast packet processing is DPDK, which has outstanding performance regarding the throughput and latency. However, there are some drawbacks when the usage is done in the edge: *(i)* the polling mechanism for packet processing keeps the CPU exclusively occupied even if there is no traffic, leading to wasted resources; and *(ii)* DPDK interface becomes unavailable for the applications inside the host, so the integration between a non-DPDK application and a DPDK application becomes a hard task. In this paper, we propose an open-source in-kernel 5G UPF solution based on 3GPP Release 16 to be deployed in a restrictive environment like MEC, where MEC host and UPF are collocated with the Base Station, sharing the same computational and network resources. The solution leverages the BPF/XDP, a novel Linux kernel technology for fast packet processing. We show it can scale and achieve about 10/11 Mpps using only 60% of the CPU with 6 cores.

Keywords: 5G; UPF; XDP; BPF.

List of Figures

List of Tables

Contents

1 Introduction

Mobile communications networks have been undergoing constant changes since their emergence in the 1980s, where their main functionality was to carry out data transmission, predominantly voice, in an analog form. At that time, networks were composed exclusively of hardware equipment that performed specific functions. Over time, new products (e.g. computer games) and services (e.g. Netflix, Surveillance and Monitoring) were developed and integrated into these networks and increasingly began to demand more resources. In addition, with the development of Industry 4.0 or Fourth Industrial Revolution, for example, there is an increasing need to develop a technology that is able to meet, mainly, the demand of eMBB (enhanced Mobile Broadband), uRLLC (ultra Reliable Low Latency Communications) and mMTC (massive Machine Type Communications) (GSMA, 2014). These scenarios demand challenges regarding the creation of infrastructures capable of meeting the applications that demand strict requirements of latency, real-time operation and high traffic rate, typically in the order of 10 Gbps.

With the emergence of the concepts of NFV (Network Function Virtualization) and SDN (Software Defined Network), networks are going through a new era called "Network Softwarization". The first enables operators to use generic and commoditized equipment to perform virtualized network functions. The second enables the separation of the control plane from the data plane, which were previously centralized on the same network device, improving network management and programmability. This separation is called CUPS (Control User Plane Separation). Therefore, these concepts made LTE networks (Long Term Evolution), which were initially conceived based on equipment with specialized functions, more efficient and competitive. As for the so-called fifth generation of mobile communication networks (i.e., 5G networks), these new paradigms are already an integral part of their system architecture.

The new model presented makes mobile communication networks more flexible, scalable, open and programmable. One of the challenges is associated with the performance of these networks, given that their components, previously embedded in specific hardware devices, can now be implemented in software and run on generic and commoditized equipment. In addition, uRLLC and eMBB scenarios demand new operational requirements for 5G networks. For example, tactile application requires low latency requirements (on the order of 1 ms) that cannot be achieved if only technologies used in 5G networks are considered (BONATI et al., 2020). In this context, one of the alternatives is to enable edge computing in mobile communication networks. Because of the edge infrastructure where the applications and services are deployed is closer to the users, the end to end latency is reduced and traffic is offload from the core of the network (ETSI, 2018b). This solution is known as MEC (Multi-Access Edge Computing), which therefore uses the NFV, SDN and Edge Computing paradigms as enabling technologies.

There are two key components in MEC: MEC host and UPF (User Plane Function). The former one is a general purpose computing facility that provides computing and storage resources to applications (ETSI, 2018b), while the latter is mainly responsible for handling user traffic to the appropriate DN (Data Network). These components could be collocated within the Base Station (edge), sharing the same network and computational resources (Figure 1.1). So, one of the concerns is about efficient resource utilization, because the infrastructure is shared between applications. Besides, the edge computing infrastructure can scale from datacenters to a single device and can be based on x86 or ARM processors (OpenStack, 2020), which depends on the use case requirements. So, because of this diversity, portable solutions for fast packet processing might be considered to avoid creating multiple version of the same solution for each infrastructure.

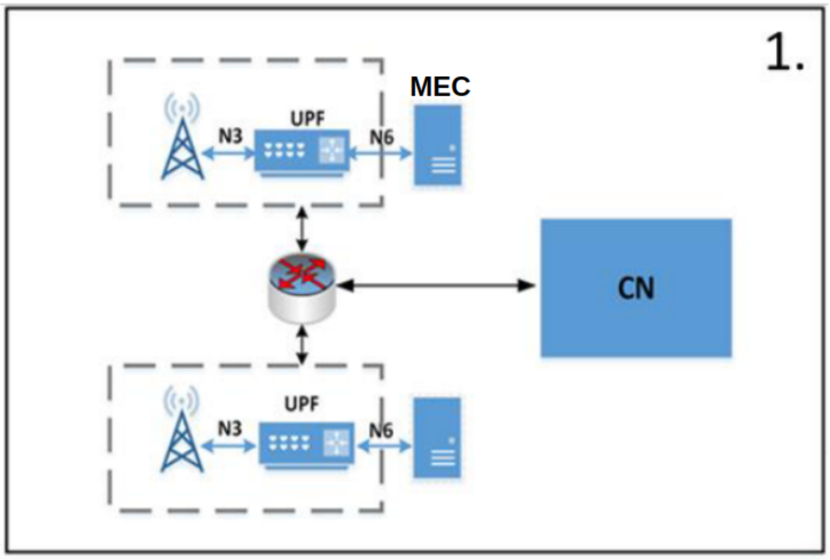


Figure 1.1 – MEC deployment option 1. MEC and UPF collocated with the Base Station. Source: adapted from (ETSI, 2018a).

DPDK is a well-known fast packet processing technology. Although it has an outstanding performance regarding the throughput and latency (INTEL, 2014), there are some drawbacks for a typical MEC deployment scenario: *(i)* DPDK support by the NICs; *(ii)* the polling mechanism for packet processing (DPDK Poll Mode Drivers) keeps the CPU exclusively occupied even if there is no traffic, leading to wasted resources; and *(iii)* DPDK interface becomes unavailable for the applications inside the host, so the integration between a non-DPDK application and a DPDK application becomes a hard task (HØILAND-JØRGENSEN et al., 2018).

In this thesis, we propose an open-source[1] run-time adaptive in-kernel solution

[1] <https://github.com/navarrothiago/upf-bpf>

based on 3GPP Release 16 as an alternative for 5G UPF (User Plane Function). The solution leverages the BPF/XDP (eXpress Data Path), a novel built-in Linux kernel technology for fast packet processing, which has advantages over DPDK (HØILAND-JØRGENSEN et al., 2018), i.e. *(i)* integrates cooperatively with the regular networking stack; *(ii)* does not require dedicating full CPU cores to packet processing. We present the proposed solution that can scale and achieve 10 Mpps using only 60% of the CPU with 6 cores. The main contributions of this thesis can be summarized as follows:

- Design of BPF/XDP based 5G UPF utilizing defacto library, which makes it easier to integrate with other software and potentially expand functionality in the future;
- Open-sourced the implementation so others can learn from it and advance the knowledge of the society;
- The method can reach 10 Mpps throughput and with CPU utilization due to event handling of the received packet;
- Reference points when comparing with the performance of other approaches and/or improving the performance of UPF.

The rest of the thesis is organized as follows. Chapter 2 describes relevant background on the in-kernel technology for fast packet processing (BPF/XDP) and the 5G System (5GS) architecture. Chapter 3 discusses related work and 5G open source projects. Chapter 4 describes the problem statement and introduces the main objectives and proposed methodology of the thesis. Chapter 5 details the design and explains the key components used in the implementation. Chapter 6 describes the setup and the test cases for the performance evaluation of the proposal design. The conclusions and future work are given in the last Chapter.

2 Background

This section provides relevant background on BPF/XDP and 5G user plane functions.

2.1 BPF

BPF (LWN.net, 2020), well known as eBPF, is a general purpose engine that allows you to execute instructions in the Linux kernel itself with two main goals: (*i*) to deliver negligible overhead when mapping these instructions to native instructions, and (*ii*) to guarantee that the program is safe at load time. This technology has been available since kernel version 3.14.

The engine is composed of 11 64-bit registers[1], a program counter and 512 bytes stack. Programs can be implemented using the "strict C" language and compiled using GNU (GNU Compiler Collection) or LLVM as backend[2] (Figure 2.1). After being compiled, the generated *bytecode* can be loaded via the *bpf* system call using the *BPF_PROG_LOAD* command. From there, two steps are performed: verification and, if enabled, JIT compilation (Just-In-Time).

The verification step is done by the Verifier. It ensures that the program runs safely and block it during the loading phase if the problem does not satisfy some checks. One of them is against infinite loops. These loops are prohibited in BPF programs to ensure that the program always terminates.

After verification, the bytecode can be JIT-compiled inside the kernel as a performance optimization. In this step, the transformation of the generic bytecode into machine instructions native to the hardware that the operating system is running takes place. For this reason, the BPF program can run just as efficiently as kernel modules or natively compiled code (Cilium Community, 2022).

After being loaded and verified, BPF programs can be hooked in specific areas (e.g. network stack) in the Linux kernel and its execution is triggered by an event (e.g. a received packet). One of them is XDP, which will be introduced in the next section. The BPF portability means that the BPF bytecode works correctly across different kernel versions without the need to recompile it for each specific kernel version. The BPF Compile Once, Run Everywhere (CO-RE) (NAKRYIKO, 2020) is the main technology behind this concept.

1 <https://github.com/torvalds/linux/blob/v4.20/include/uapi/linux/bpf.h#L45>
2 <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/filter.rst>

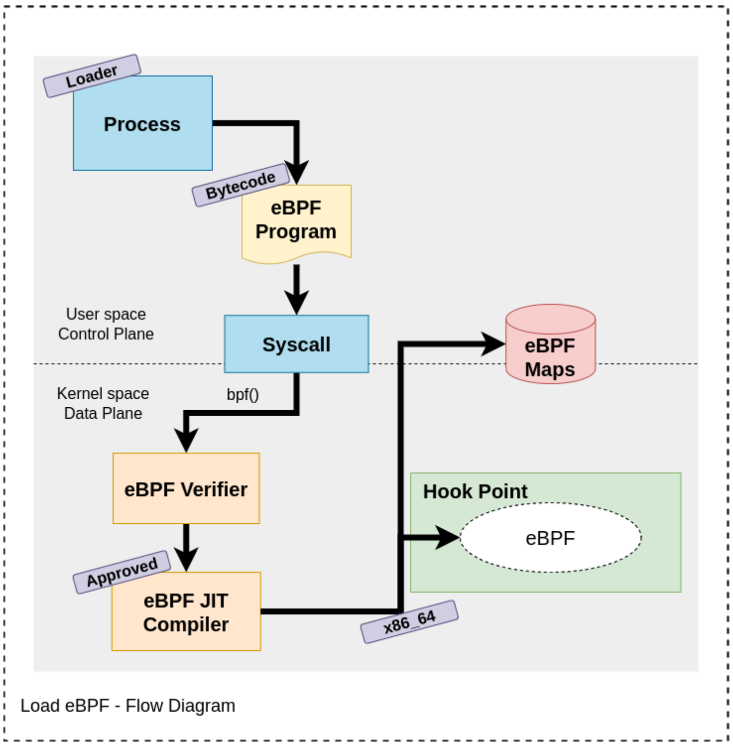


Figure 2.1 – Diagram representing the BPF program load flow in the kernel. Source: adapted from (Cilium Community, 2022)

BPF programs can store different types of data in generic structures called Maps. Each data is accessed by a key. Maps can be shared between userspace programs and BPF programs. More information about BPF is available at (Cilium Community, 2022), (VIEIRA et al., 2020).

One of the main advantages of using this type of technology is to enable programmability of the kernel without having to change its source code or install additional modules. Furthermore, BPF programs have a stable ABI (Application Binary Interface) that guarantees BPF programs keep running with newer kernel versions.

On the other hand, BPF programs have restrictions that make it challenging to develop these programs to perform more complex functions. Among them, the following can be highlighted:

- Limit of 5 arguments for a function;
- Limit of 32 nested *tail call* calls;
- Infinite loops are not allowed;
- Send the same packet to multiple ports;
- Only 30 data structure BPF maps types available (kernel v5.16.10)[3];

[3] <https://elixir.bootlin.com/linux/v5.16.10/source/include/uapi/linux/bpf.h#L878>

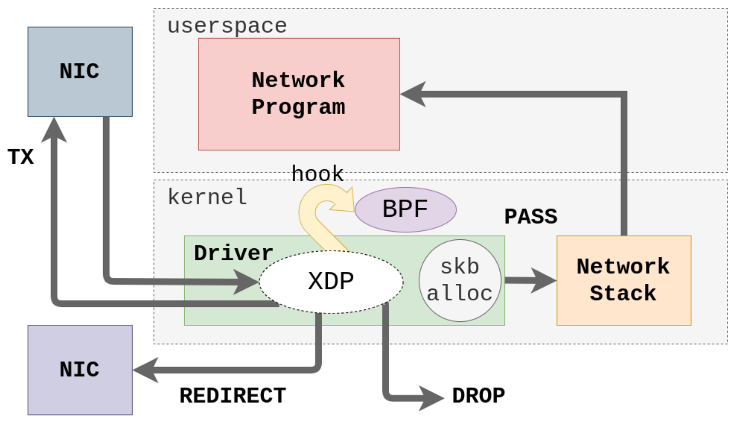


Figure 2.2 – Simplified XDP architecture diagram with the possible actions that can be applied to the received packet before the *socket buffer* allocation by the operating system. Source: adapted from (SUSE, 2022).

Some of these restrictions as well as workarounds for them are discussed in (Miano et al., 2018).

2.2 XDP

XDP (VIEIRA et al., 2020) enables fast packet processing within the Linux kernel through a hook point located in the reception chain of a network device driver before memory (i.e. socket buffer) allocation by the operating system. When the packet is received by the network device driver, the hook which contains the BPF program is executed. The BPF program can perform different actions, such as dropping the packet, redirecting it to another interface, or sending it to be processed in the Linux network stack (Figure 2.2). The network device driver must support XDP to take full advantage of its benefits. If not supported, the program runs in generic mode at lower performance. It is worth mentioning that the portability for this kind of program is almost transparent to the developer due to the extra layer abstraction between the kernel and BPF context that is represented by its structures. For instance, there is a mapping between the BPF structure (xdp_md) into internal kernel structure (xdp_buff) [4].

2.3 Libbpf

Libbpf[5] is a userspace library for loading and interacting with BPF programs. It is part of the Linux source tree and it is the reference library de facto. It is an alternative for the BCC (BPF Compiler Collections) toolkit. BCC has some disadvantages when comparing with libbpf, for instance, the userspace program has high footprint due to the clang compiler dependency during run-time, which uses the compiler to build the

[4] <https://elixir.bootlin.com/linux/v5.16.10/source/net/core/filter.c#L9116>
[5] <https://www.kernel.org/doc/html/latest/bpf/libbpf/index.html>

BPF program during the user program execution. This means that the development of the BPF code becomes harder, because the compilation errors are detected at run-time instead of the compilation time. Besides, the BCC depends on the kernel headers during the compilation, which might not be available inside the production environments. So, the adoption of libbpf creates a userspace program much smaller and improves the software life cycle of the BPF program. Another advantage is that libbpf is used by BPF CO-RE technology. The `upf-bpf` project is based on libbpf.

2.4 5G Network Architecture

The architecture of 5G networks is represented in Figure 2.3. The colored regions represent the 5G network core, called New Generation Core (NGC), where the green block represents the User Plane (UP), while the blue ones, the Control Plane (CP). One of the main changes regarding LTE networks is to provide a Service-Oriented Architecture (SOA), i.e., composed of virtual network functions with well-defined interfaces. These functions can be performed on commodity hardware equipment and accessed through a communication protocol (e.g. HTTP). Furthermore, the 5G architecture follows the Control and User Plane Separation (CUPS) model. This paradigm enables the deployment of UP closer to applications and services, thus reducing latency and traffic in the core of these networks.

It is important to note that 5G networks can work in two modes: Non-Standalone and Standalone. The former allows the deployment of the new generation of 5G radio (New Radio - NR) along with the core infrastructure of LTE networks, as known as EPC (Evolved Packet Core) to reduce the cost of deployment. The latter, instead of deploy the EPC, the NGC takes its place.

In the next sections, two relevant components of the NGC are detailed: Session Management Function (SMF) and UPF.

2.4.1 SMF

SMF is part of the 5G CP. One of the most relevant SMF functions is to manage the sessions used for data traffic between the UE and the Data Network (DN), namely Session Protocol Data Units (PDU). Each session is represented by a logical tunnel passing through the UE (User Equipment), Radio Access Network (RAN), and UPF. When a PDU session is established, a context is created in each of these components. A context contains a set of specific rules that will be applied in the data packet in order to guarantee Quality of Service (QoS) and to forward the packet to the next hop, for instance. In the case of UPF, this context is represented by the Packet Forwarding Control Protocol (PFCP) Session.

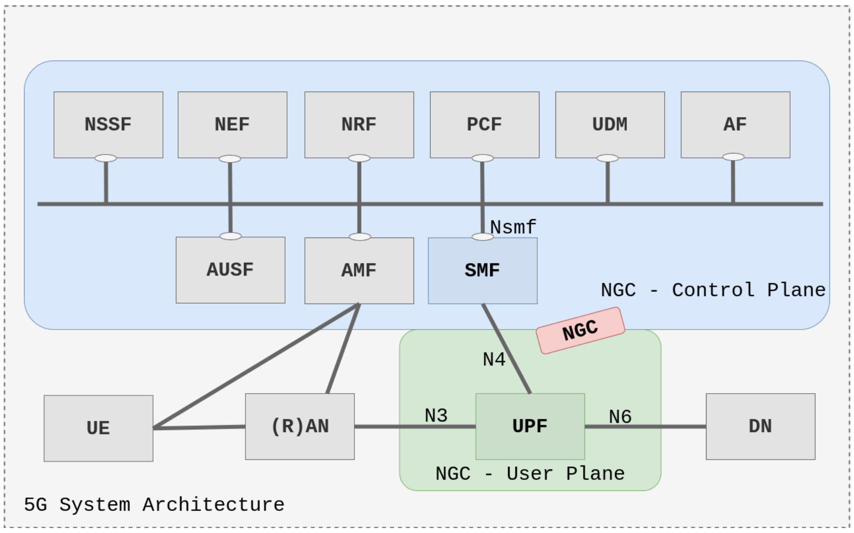


Figure 2.3 – Simplified 5G System architecture. Highlighted, the SMF and UPF components are addressed in the next sessions. The colored region represents the core of the 5G network.

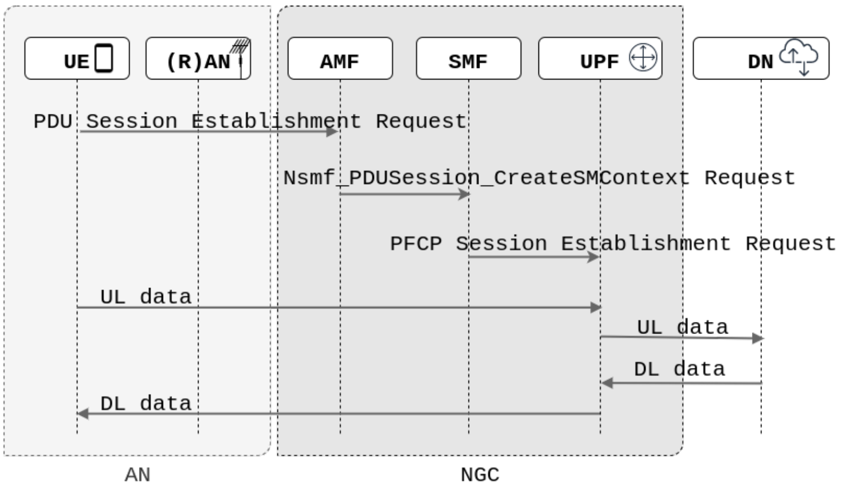

UE
(R)AN
AMF
SMF
UPF
DN
PDU Session Establishment Request
Nsmf_PDUSession_CreateSMContext Request
PFCP Session Establishment Request
UL data
UL data
DL data
DL data
AN
NGC

Figure 2.4 – Simplified procedure for requesting session establishment.

PFCP is the signaling protocol used for communication between SMF and UPF components. This communication involves PFCP session management procedures. The communication interface between the two components is represented by reference point N4 in Figure 2.3.

CP components communicate with SMF through the service provided by Nsmf_PDU-Session (ESTI, 2020a). This service involves PDU session creation, update, and removal procedures. One session establishment use case example is when the UE is registered at the core of the network and has data to send to the DN. If there is no session established, the procedure for requesting session establishment will be carried out as shown in Figure 2.4. After performing the session establishment, the UE will be able to send/receive data in uplink/downlink direction.

Compared with LTE networks, the SMF encompasses a set of functionalities from the Mobility Management Entity (MME) component within Evolved Packet Core (EPC). Additionally, the N4 reference point encompasses a set of functionalities defined for the Sxa/Sxb/Sxc interfaces, which are also used for signaling between the CP and the UP (ESTI, 2020c).

2.4.2 UPF

One of the main user plane components for NGC is the UPF. It is responsible for several functionalities related to user data traffic, such as forwarding and routing packets, applying rules to ensure quality of service, generating traffic usage reports, and inspecting packets (ESTI, 2020a). It works like a gateway between the RAN and the external DN (e.g. Internet, IP Multimedia System, local data network, etc). Regarding LTE networks, the UPF encompasses the functionalities of Servicing Gateway (SGW) and Packet Data Network Gateway (PGW)[6].

Data traffic takes place through a PDU session that is represented by contexts stored in the UE, RAN, and UPF components. In the case of the UPF, the context is represented by the PFCP Session, which contains the following rules:

- Packet Detection Rules (PDRs) - Rules for packet detection;
- Forwarding Action Rules (FARs) - Rules for forwarding packets;
- QoS Enforcement Rules (QERs) - Rules for applying QoS;
- Usage Reporting Rules (URRs) - Rules for generating reports;
- Buffer Action Rules (BARs) - Rules for "buffering" packets;
- Multi-Access Rules (MARs) - Rules for traffic steering functionality.

Table 2.1 shows the relationship between the signaling interfaces (5G and LTE) used between the CP and UP of 5G and LTE networks and the rules applied in the packet. We will see that the upf-bpf supports PDRs and FARs.

The flow for processing packets in the UP is represented in Figure 2.5. When the packet is received, its header is analyzed to find the PFCP Session to which the packet belongs. Once the session is found, the UP looks up the highest precedence PDR. The PDR contains the Packet Detection Information (PDI), which has Information Elements (IEs) that will be matched with the header of the received packet. For instance, the PDI contains the UE IP address that can be the source (uplink) or the destination (downlink) IP address

[6] In this work, the term SPGW will be used to refer to the combination of the SGW and PGW components as specified in (ESTI, 2020d). SPGWu represents the component responsible for implementing the user plan functionalities for LTE networks.

Rules	Interfaces			
	Sxa	Sxb	Sxc	N4
PDR	x	x	x	x
FAR	x	x	x	x
URR	x	x	x	x
QER	-	x	x	x
BAR	x	-	-	x
MAR	-	-	-	x

Table 2.1 – Relation between the signaling interfaces (5G and LTE) used between the CP and UP and the rules applied in the packet.

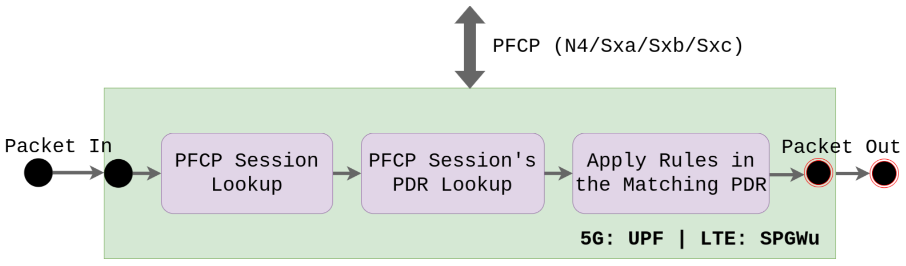


Figure 2.5 – Flow of packet processing in UP function defined by UPF and SPGWu, components of NGC and EPC respectively.

of the packet, depending on the traffic direction. After the PDI matches with the header of the packet, there might be more than one PDR that has the same PDI IEs. However, just the highest precedence is selected. In the next step, the selected PDR contains the rules (QERs, URRs, BARs or MARs) that are applied, if they are available. Only the FAR is mandatory according to specification (ESTI, 2020c). Finally, the packet can be forwarded to the network interface as defined in the FAR. It is important to highlight that these steps fit both 5G networks (UPF) and LTE networks (SPGWu) (ESTI, 2020c).

One of the protocols used to load user data packets is GPRS Tunneling Protocol User Plane (GTPu)(ESTI, 2020b). The original packet (IP datagram) is called a Transport Protocol Data Unit (T-PDU). When combined with the GTPu header, the packet is called a GTP encapsulated user Protocol Data Unit (G-PDU). The diagram representing the protocol stack used in the communication between the UP elements is represented in Figure 2.6. The Tunnel Endpoint ID (TEID) indicates which tunnel a particular T-PDU belongs to and it is part of GTP Standard Header. The GTP Extended Header is set whenever one of the flags Sequence Number, NPDU number or Extension Header is set.

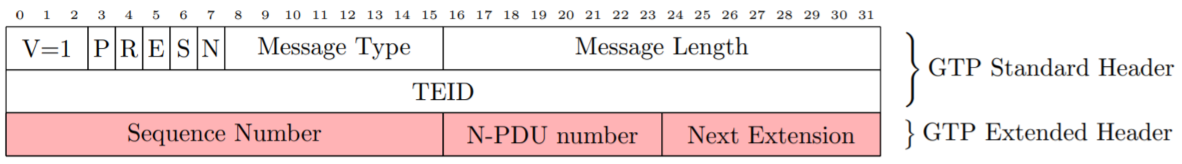


Figure 2.6 – GTP-U header with extensions. Source: (GIRONDI, 2020).

3 Related Work

Since the introduction of concepts such as NFV (PAPER, 2012), several works have been carried out addressing the issue of fast packet processing (FEI et al., 2020) in general and tailored to 5G (BONATI et al., 2020).

3.1 Academic Research

The authors in (Ricart-Sanchez et al., 2018a) present a prototype for packet processing based on hardware using the programmable platform NetFPGA with the programmable data plane in the P4 language. The work consists of developing a firewall located between the core and the edge of 5G networks. The firewall is responsible for analyzing the internal IP headers of each packet, unlike the traditional firewall, which only analyzes the external header. In parallel, (Ricart-Sanchez et al., 2018b) also proposes a similar solution, but with a focus on multi-tenancy scenarios to ensure quality of service for applications with strict latency requirements, with tests performed with OpenAirInterface (OAI) (OAI Community, 2022). Although FPGA based solutions perform well, they are considered expensive compared to general purpose CPU. In addition, they are difficult to be implemented, as they use low-level hardware description languages (low-level hardware description languages - HDLs), such as the VHDL language (FEI et al., 2020). It is important to point out that these presented solutions do not seem to be ideal to be implemented in datacenters infrastructures located at the edge, which have restrictions regarding the implementation cost (OpenStack, 2020).

Regarding software-based packet processing solutions, (Pinczel et al., 2015) presents a prototype using the modular router, Click (KOHLER et al., 2000), integrated with the framework Netmap (RIZZO, 2012) for transferring session context between base stations. In addition, (Chen; Liu, 2019) evaluates a DPDK-based (INTEL, 2022) prototype for media gateway located in the IP multimedia subsystem (Multimedia Subsystem - IMS). DPDK-based solution can increase performance 10 times for packet processing (INTEL, 2014), however it is surpassed by XDP technology in scenarios when packet forwarding happens through the same interface (HØILAND-JØRGENSEN et al., 2018).

Finally, to the best of our knowledge, the only work found in the literature that comes closest to this proposal are the works from Parola (Parola, F. et. al., 2020), (Parola, F., 2020) and (Parola, F. et. al., 2021) and which present a prototype of a mobile gateway based in BPF using TC and XDP technology for fast packet processing. It focuses on developing a component that can be deployed on the edge. The main features presented are packet forwarding and classification and ensuring QoS policies. It was developed inside the Polycube Framework (Polycube Community, 2022). The framework provides fast and lightweight network functions such as bridges, routers, firewalls, and others (the pull request related to the mobile gateway was integrated yet). It also supports traffic debugging. Regarding the performance evaluation, Parola showed that the solution scale using multiple cores. The main advantages of our work comparing with (Parola, F. et. al., 2020) are *(i)* it is decoupled to a specific frameworks, i.e. the solution is basically depends on the Linux kernel; *(ii)* it is aligned with 3GPP specifications and; *(iii)* it is based on *libbpf* instead of BCC (BCC Community, 2022). As already mentioned in Section 2.3, BCC is not maintaining by Linux community and depends on the clang compiler in runtime. So, we argue that our solution can be easily integrated with different software-based 5G used plane solutions.

3.2 Open Source Projects

The approach presented in this work may leverage open source projects for telecommunication networks core, such as srsLTE (SRS LTE Community, 2022), OAI (OAI Community, 2022), Open5GS (Open5GS Community, 2022), UPF-EPC (OMEC Community, 2022), Magma Facebook (Facebook, 2022), and Free5Gc (Free5Gc Community, 2022). Only srsLTE (SRS LTE Community, 2022) does not provide support for NCG functionalities. The OAI, srsLTE, and Open5GS solutions do not have specific technologies for fast processing in the UP, which has been developed in the operating system's user space. Already UPF-EPC, Magma Facebook, and Free5Gc present kernel-level UP implementations using the kernel module gtp5g (GTP5G Community, 2022), BESS (Berkeley Extensible Software Switch) (HAN et al., 2015) and OvS (Open vSwitch) (PFAFF et al., 2015), respectively. With regard to CUPS support, we can highlight OAI, UPF-EPC, Open5GS, Free5Gc, and Magma Facebook. Although the Magma Facebook is based on an older version of the OAI without CUPS support, the solution was built using the SDN Ryu controller (Ryu Community, 2022) and OvS. We believe all these projects could benefit from our proposed solution, especially those that do not support fast packet processing, like OAI, srsLTE, and Open5GS.

Academic Research							
	5GS		Fast Packet Processing		Experimental Evaluation		
Work	**Location**	**Component**	**Based on**	**Technology**	**Env**	**OSS CN**	**Application**
(Parola, F. et. al., 2020), (Parola, F., 2020), (Parola, F. et. al., 2021)	Edge	UPF	SW	BPF, TC, XDP	Polycube	No	QoS, Traffic forward
(Chen; Liu, 2019)	N/A	N/A	SW	DPDK	Standalone	No	5G Media Gateway
(Pinczel et al., 2015)	Core	UPF	SW	Netmap, Click	Docker Container	No	Context migration, service chain
(Ricart-Sanchez et al., 2018b)	Edge & Core	gNB & UPF	HW	NetFPGA, P4	OAI	Yes	Multi-Tenancy, QoS, Multimedia
(Ricart-Sanchez et al., 2018a)	Edge & Core	gNB & UPF	HW	NetFPGA, P4	Standalone	No	Firewall
This Work	Edge	UPF	SW	BPF, XDP	Standalone	Yes	Traffic forward
Open Source Software							
	5GS		Fast Packet Processing		Details		
OSS	**Location**	**Components**	**Based on**	**Technology**	**CUPS**	**Language**	**Application**
(OAI Community, 2022)	Core	U/D	U	U	Yes	C/C++	CN SA/NSA
(SRS LTE Community, 2022)	Core	U	U	U	No	C++	CN NSA
(Facebook, 2022)	Core	U/D	U	OvS, Kernel module	No	C	CN NSA
(OMEC Community, 2022)	Core	UPF	SW	BESS	Yes	C++	CN SA
(Open5GS Community, 2022)	Core	U/D	U	U	Yes	C	CN SA
(Free5Gc Community, 2022)	Core	All	SW	Kernel module	Yes	Go/C	CN SA/NSA
This Work (upf-bpf)	Core	UPF	SW	BPF, XDP	Yes	C++	CN SA/NSA

Table 3.1 – Related work.
Legend: U/D - Under Development; N/A - Not Apply; All - All NCG components; U - Unavailable;

4 Problem Statement and Approach

The current 5G user plane technologies presented in the previous chapter are based on sockets, programmable switches such as BESS, OvS or kernel module. Regarding the scenarios where the implementation of these solutions takes place in diversified infrastructures and with lower computational power (e.g. NPN local networks) when compared to datacenters, the following challenges can be highlighted:

- Kernel modules have a dependency on existing data structures in the kernel itself. These structures can vary depending on the version. Therefore, the cost of maintaining the solution's compatibility for different kernel versions is high. Furthermore, these solutions that run in kernel space can be compromised all the system if there is a bug (e.g. invalid memory access), leading the system to crash;
- OvS depends on kernel implementations. Therefore, it presents the same problems as the previous item;
- BESS is based on DPDK. This solution exclusively keeps the CPU busy, even if there is no traffic. In addition, a dedicated network interface is required. Then, computational resources are wasted even if there is no traffic to be processed;
- IPv4 sockets rely on Linux networking stack, which is a generic and inefficient implementation for processing high data rates (i.e. on the order of 10 Gbps);

4.1 Research Objectives

Based on the presented foundation in the evolution of 5G mobile telecommunication networks, in the challenges and opportunity for high-performance programmability in the Linux kernel, the objective of this research was to investigate the BPF and XDP technology to be applied in the 5G UPF. The main objectives are to answer the following research questions:

- **RQ1**: Is it possible to create a solution for 5G UPF using BPF/XDP technologies? How would be the design in order to create a proof of concept for the 5G UPF using BPF/XDP technologies following the specifications of *Release* 16 of 3GPP?
- **RQ2**: What levels of performance can be expected for the proposed solution? How is the performance compared with existing solutions based on BPF/XDP technologies?

4.2 Methodology

To address the research objectives, the following activities are proposed:

(i) define the minimum requirements for the UP of the 5G network core;

(ii) develop core features of the 5G UPF using BPF and XDP based on the requirements of the previous step;

(iii) evaluate the performance in terms of scalability and workload of the proposed solution.

The work differs from other works by presenting an investigation of the use of BPF and XDP technologies for fast packet processing. Furthermore, the solution *(i)* is not limited to specific *frameworks*, such as (Parola, F. et. al., 2020), *(ii))* is aligned with 3GPP specifications and also *(iii)*, as it is decoupled, i.e. it can be reusable in different open source 5G solutions, like the OAI.

Therefore, the approach of this proposal is innovative and promising. Regarding the knowledge of related works so far, it is the first work to investigate and propose the development of a prototype of the UP using BPF/XDP that is aligned with 3GPP specification.

Briefly, the main advantages of using BPF and XDP technology are:

- Action taken for a given packet can be performed with higher performance than existing technologies which implement the data plan in telecommunication networks, possibly decreasing the total latency for packet processing;
- The data plan can hold the flexibility to reprogram, add or remove functionality in real time without interruptions in data traffic;
- BPF/XDP technologies, currently state-of-the-art in networks, are widely developed and used by large cloud computing providers, being therefore robust, well-documented, and with a wide spectrum of production use cases;
- The benefits of the SDN and NFV paradigms are directly applicable to the solutions proposed with BPF/XDP, such as the portability of network functions, the programmability of data plans, and the flexibility of functionalities at runtime;
- As they are technologies integrated into the kernel, there is no need for specific hardware or dedicated solutions, i.e., BPF/XDP are interesting technologies to be implemented in diversified infrastructure, consisting of generic equipment and with more limited computational resources, but not restricted to them.

4.2.1 Design and Implementation of 5G UPF

The methodology adopted in this step consisted of defining the minimum functionalities of the solution. After that, a high level design was done to create a runtime adaptive solution based on BPF/XDP following the functionalities. The implementation was leveraged by library libbpf. The idea was to create a software layer to abstract the lifecycle management of the BPF program, the data structures (maps) shared with UP and the PFCP messages received from other network core components, as explained in Section 4.2.1. The test cases were also developed to simulate user data traffic. The traffic was generated using the TRex - Realistic Traffic Generator (CISCO, 2022a) tool. All code developed in this step was stored in a public Git repository. In addition, tools can be used to support software compilation, test execution and packaging on different platforms, such as CMake (Cmake Community, 2022). This allows the developer to work in an already known development environment, being an important item of choice for the community. It is important to note that during this phase, the open source principles available in *OpenSource.Guide* (Open Source Guide, 2022) were used.

4.2.2 Testing and Performance Evaluation

The methodology adopted in this step consisted of defining indicators (e.g. data transfer rate, scalability, CPU usage) to be used to measure the system performance (*System Under Test - SUT*) of the 5G UP based on BPF/XDP. Automated tests were created and executed against the implementation solution from Section 4.2.1. Data was collected and analyzed. The tool that was used in this phase is TRex - Realistic Traffic Generator, already mentioned, to generate traffic and perform flow measurements in the *downlink* and *uplink* directions. All documentation and results were stored in the same repository as the developed code. At the end of this step, graphs were created and analyzed based on the results.

5 Design and Implementation of 5G UPF

This section describes our proposed architecture and implementation for fast packet processing using BPF/XDP for user plane on the mobile core network (5G/LTE). The key pillars of the 5G UPF prototype are:

- In-kernel fast packet processing;
- Flexible, extensible and programmable dataplane;
- Portable to different systems.

These points are achieved mainly by BPF/XDP and CO-RE (Compile Once - Run Everywhere) technologies. Besides, it is based on the following 3GPP Technical Specification (ESTI, 2020a) and (ESTI, 2020c). The main goal is to enable in-kernel fast packet processing in third-party UPF/5G or SPGWu/LTE components in order to:

- Boost them for those which does not have any fast packet processing enabled, or;
- Co-locate them with other fast packet processing solutions (e.g. DPDK).

Possible scenarios that take advantage of this type of technology: MEC, 5G NPN (Non Public Networks), on-premise, 5G enterprise.

5.1 Features

Routing and forwarding packets are amongst the most relevant dataplane features in 5GS (ESTI, 2020a). We have defined these actions as the core functionalities of the proposed solution, as part of the PFCP session context, which is created by sending PFCP Session Establishment Request message from the control plane (i.e. SMF for 5G network or SPGWc for LTE networks) to the dataplane (i.e. UPF for 5G and SPGWu for LTE). This request is sent when the UE has data to transmit to the network, for instance. The diagram presented in Figure 5.1 shows the PFCP session context data model of the proposed solution based on (ESTI, 2020c). It does not support all IEs, but only the PDRs and FARs to enable routing and forwarding packets in the core network user plane. The main functionalities supported are:

i PFCP session management: create, read, update, and remove PFCP sessions, PDRs and FARs;

ii Fast packet processing for uplink and downlink user data traffic: classify and forward UDP and GTP traffic based on PDR and FAR, respectively.

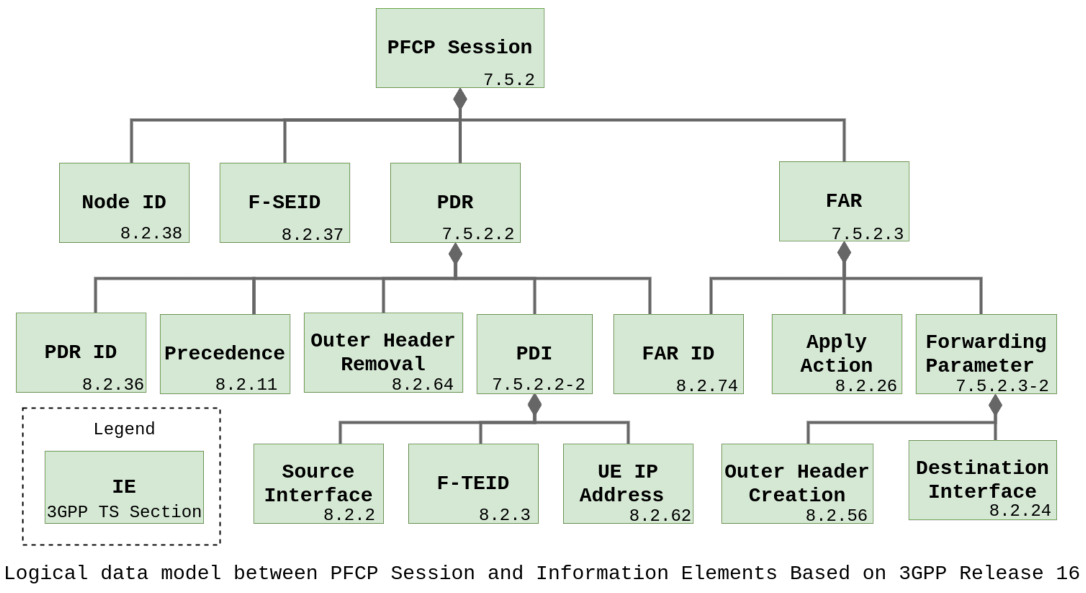


Figure 5.1 – The PFCP context data model with the IEs of the proposed solution.

The PFCP Context Data Model

The PDR contains the following IEs: the identifier (PDR ID), the precedence, the forwarding rule identifier (FAR ID), the packet detection information (PDI) and the identifier for removing headers (Outer Header Removal). The PDI contains information about the TEID (Tunnel Endpoint Identifier), the identification of the packet's source interface (Source Interface) and the UE IP address, which can be the source or destination IP, depending on the source interface. This information will be compared with the GTPu/UDP/IP header of the received packet. The Precedence defines the selection precedence of all PDRs of the same PFCP session. The lower the precedence, the higher the priority.

The FAR contains the following IEs: the rule identifier (FAR ID), the action to be applied to the packet (Apply Action) (e.g. forward) and the parameters that will be used when the packet is forwarded (Forwarding Parameter). These parameters are represented by two elements: the destination interface (Destination Interface) and the set of parameters used for header creation (Outer Header Creation). This is related to the arrival of UDP/IP packet by reference point N6 of DN and application of PDR and FAR to be forwarded to reference point N3. In this case, the GTP/UDP/IP header (underlayer), which is available in the parameter set, is added to the package.

In addition to the PDR and FAR, the PFCP session is composed of two more IEs: the F-SEID (Fully Qualified Session Endpoint Identifier) and the node identifier (Node

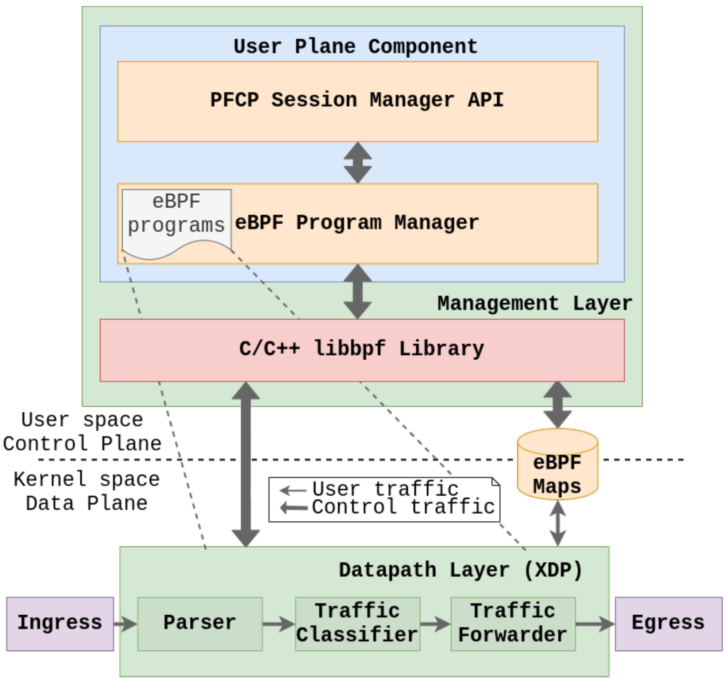


Figure 5.2 – High level design of the user plane library using BPF/XDP.

ID). This can be represented by the IP address or the FQDN (Fully Qualified Domain Name) of the component that belongs to either the CP or the UP. The Node ID is used in the PFCP Association Setup Procedure (PFCP Association Setup Procedure) to enable the CP to use UP resources (e.g. establish PFCP session) (ESTI, 2020c). During this procedure, participating entities exchange information about their Node ID. With this, each entity contains the identification of its peer. This information can be useful when an element is not responding. With this, your peer can consult, for example, in the DNS (Domain Name System) to find an altering element to carry out the communication.

Finally, the F-SEID is composed of the local IP address plus the PFCP session identifier (SEID). In this case, the IP address represents the interface that is used in the communication between the PFCP entities (e.g. SMF or UPF) that performed the PFCP Association Configuration Procedure.

Therefore, it is important to highlight that this data model presented enables the packet forwarding and routing functionalities in the UP for 5G and LTE networks. This is because such rules (PDR and FAR) are supported by reference points N4 (5GS) and Sxa/Sxb/Sxc (LTE), as shown in the table 2.1

5.2 Design

There were two versions of the design. Both follow the architecture shown in Figure 5.2. The library is divided in two main layers: Management Layer, and Datapath Layer.

5.2.1 Version 1

The version 1 was created in order to have a minimum viable prototype. The next section will present the Management Layer, the Datapath Layer and the limitations of this version.

Management Layer

It is an user space layer to manage PFCP sessions and BPF programs lifecycle. A client can create/read/update/delete PFCP sessions through API. When a PFCP session establishment request message is received by the user plane component, the message is parsed and a call is made to the library via PFCP Session Manager API. The PFCP Session Manager calls the BPF Program Manager to load dynamically a BPF bytecode, which represents the new PFCP session context, i.e., there is a BPF program running in kernel space for each PFCP session. The program contains the BPF maps used to store the PDRs and FARs. All the communication between the user space and the kernel space is through the libbpf library (Linux Community, 2022), which is maintained by the Linux kernel source tree. The PFCP Session Manager parses the structures received to BPF map entries and updates the maps with them. The PFCP session context is created in Datapath Layer, where the user traffic will be handled. Figure 5.3 shows the activity diagram of the create PFCP session use case, which is handled by this layer.

Datapath Layer

It is a kernel space layer to process the user traffic inside the XDP. A service chain function was created with three main components: the Parser, the Traffic Detector and the Traffic Forwarder. The Parser parses the ingress traffic to check if it is a uplink (GTPu) or a downlink (UDP) flow. If it is an uplink/downlink flow, the TEID/UE IP address key is used to get the PFCP session context. Then, the packet is passed to the PFCP session context represented by a BPF program via tail calls. Here, the Traffic Detector accesses the BPF hash maps in order to find a PDR associated with the packet. If there is a PDR stored, the packet passes to the Forwarder. Finally, the Forwarder uses the FAR ID from the PDR to find the FAR, which is stored in a BPF hash. The FAR contains the action (e.g. forward) that will be applied, the outer header creation and the destination interface. Besides, the Forwarder accesses other BPF maps to check the MAC address of the next hop and the index of the destination interface where the packet will be redirected. The datapath workflow is shown in Figure 5.4. The diagram presents the BPF sections and the relation between the NFs. The entry point section contains the Parser and PFCP session section contains the Detector and Forwarder.

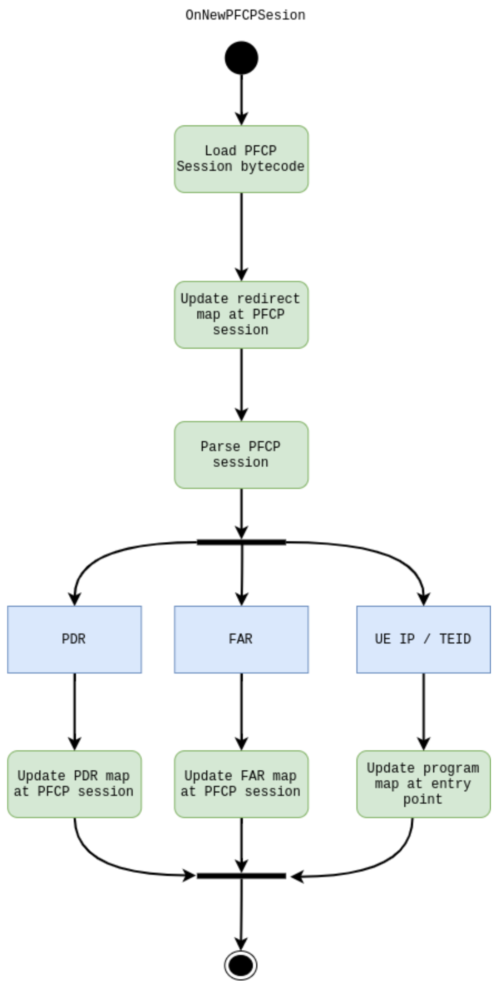

OnNewPFCPSesion
Load PFCP Session bytecode
Update redirect map at PFCP session
Parse PFCP session
PDR
FAR
UE IP / TEID
Update PDR map at PFCP session
Update FAR map at PFCP session
Update program map at entry point

Figure 5.3 – Create PFCP session activity diagram (version 1).

Constraints and Drawbacks

In the version presented, the main limitations can be divided into two categories: functionality and flexibility.

- **Functionality Constraints:** For each TEID/UE IP address, which is the key of the BPF hash maps for uplink/downlink flow, the control plane can map only one PDR. Based on (ESTI, 2020c), there might be more than one PDR mapped to the same TEID. The PDI also may take account of other IEs to lookup the PDR, for instance, the Source Interface (e.g. Access or Core, which denote an uplink and downlink traffic direction respectively). Therefore, there could be two PDR mapped to the same TEID, but with a different source interface. However, this does not solve our problem, because it is possible to have two PDRs with the same PDI. So, how does the Datapath Layer differentiate between them? Here the IE Precedence is used to define the highest precedence between them. This solves our problem: create an interaction between the PDRs which has the PDI that matches with the header of the packet and find the highest precedence between the PDRs that were found.

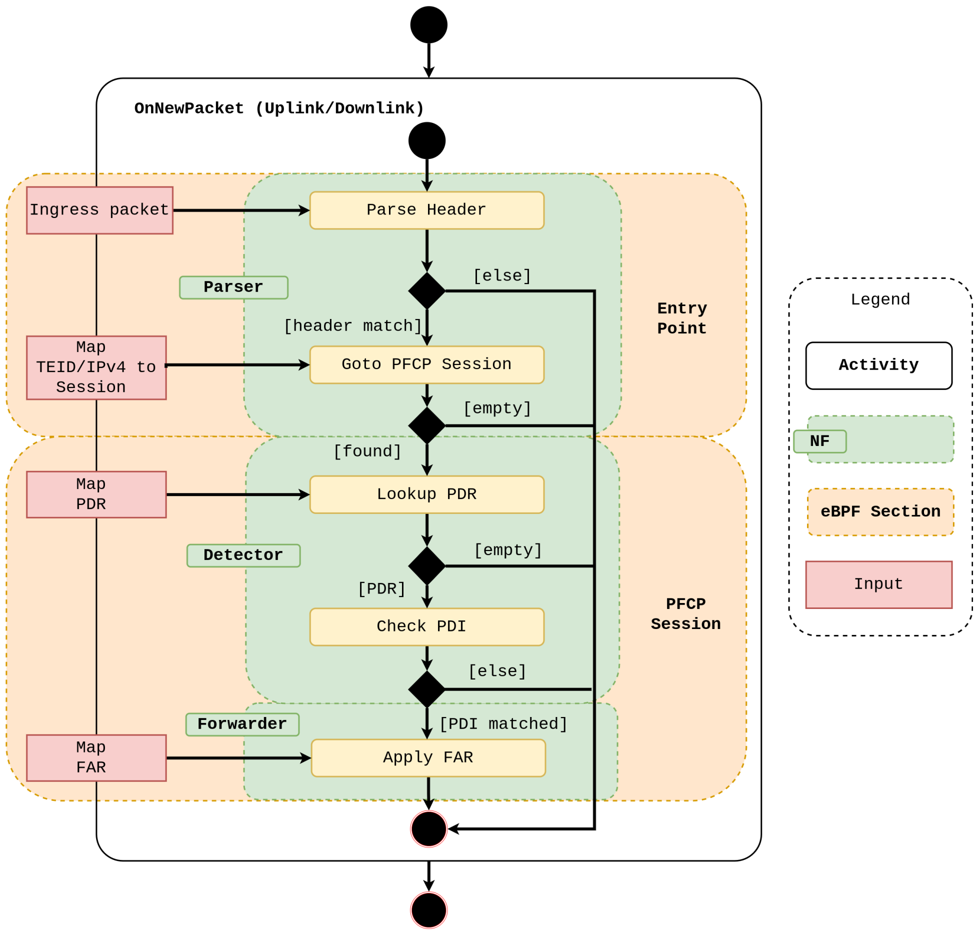


Figure 5.4 – On new packet activity diagram (version 1).

However, the way that was designed, all the PDRs are stored in the Datapath layer. In order to find the highest precedence, an iteration (loop) would be implemented to lookup the matched PDR inside the Datapath Layer (Figure 5.5). The solution does not scale, because the loop increases the size of the BPF program, leading the verifier to reject the BPF when loaded into the kernel. The verifier in Linux kernel has a limitation to verify 1M instruction maximum (The Linux Kernel documentation, 2022). Besides, the latency and the time to load the BPF program increase due to the PDR lookup loop and the size of the BPF program, respectively. Both may impact the performance for 5G ultra Reliable Low-Latency Communication (uRLLC) use cases.

- **Flexibility Constraints:** Adding new rules (e.g. QER, URR, etc) in the current version involve to add a new PFCP session context (BPF section), because all the rules are coupled in the same BPF section (Figure 5.4). Then, the BPF program (PFCP session) becomes bigger whenever we add support for new rules. Besides, the time to load it into the kernel also increases, leading the solution to have less time to respond to changes in the network (less flexible). Furthermore, the current design does not support modifying the pipeline inside the PFCP session BPF section.

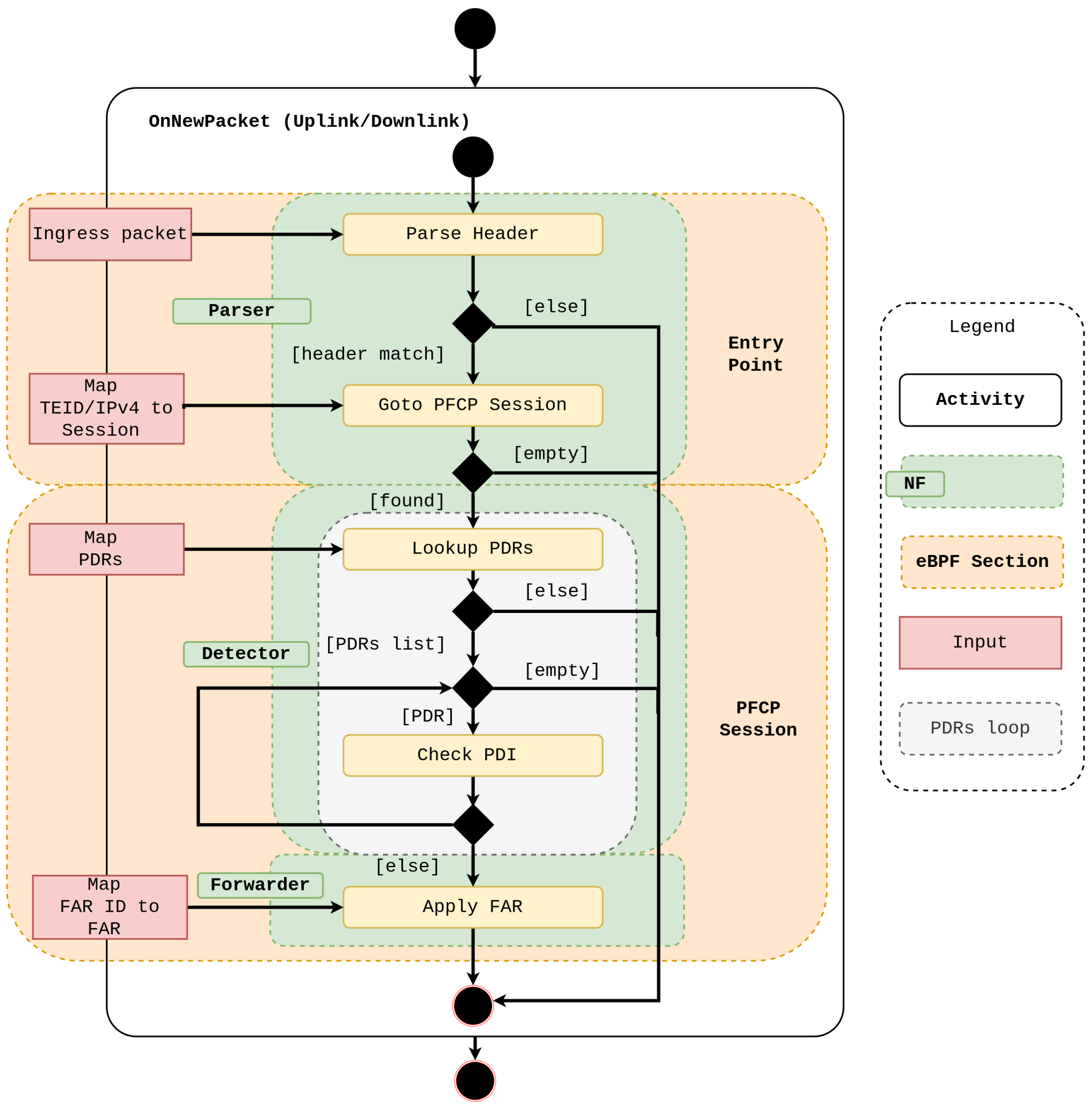


Figure 5.5 – Workflow in Datapath Layer with PDR lookup loop (version 1).

Figure 5.6 shows the current design adding the QER support. There might be some cases that a PFCP session is composed of only one FAR (just the FAR is mandatory (ESTI, 2020c)). In this case, whenever the packets arrive, the BPF program will lookup if there is a QER and it will skip the apply QER action. This operation leads to the increase of the packet processing latency. This check could be avoided if there is one BPF program for each rule and the Management Layer only deploys those programs (rules) that are available inside the highest priority PDR of the PFCP session.

5.2.2 Version 2

A new design was created to fix the limitations discussed in the previous section. Here, we will explain the main differences of the design in the Management Layer and the Datapath Layer.

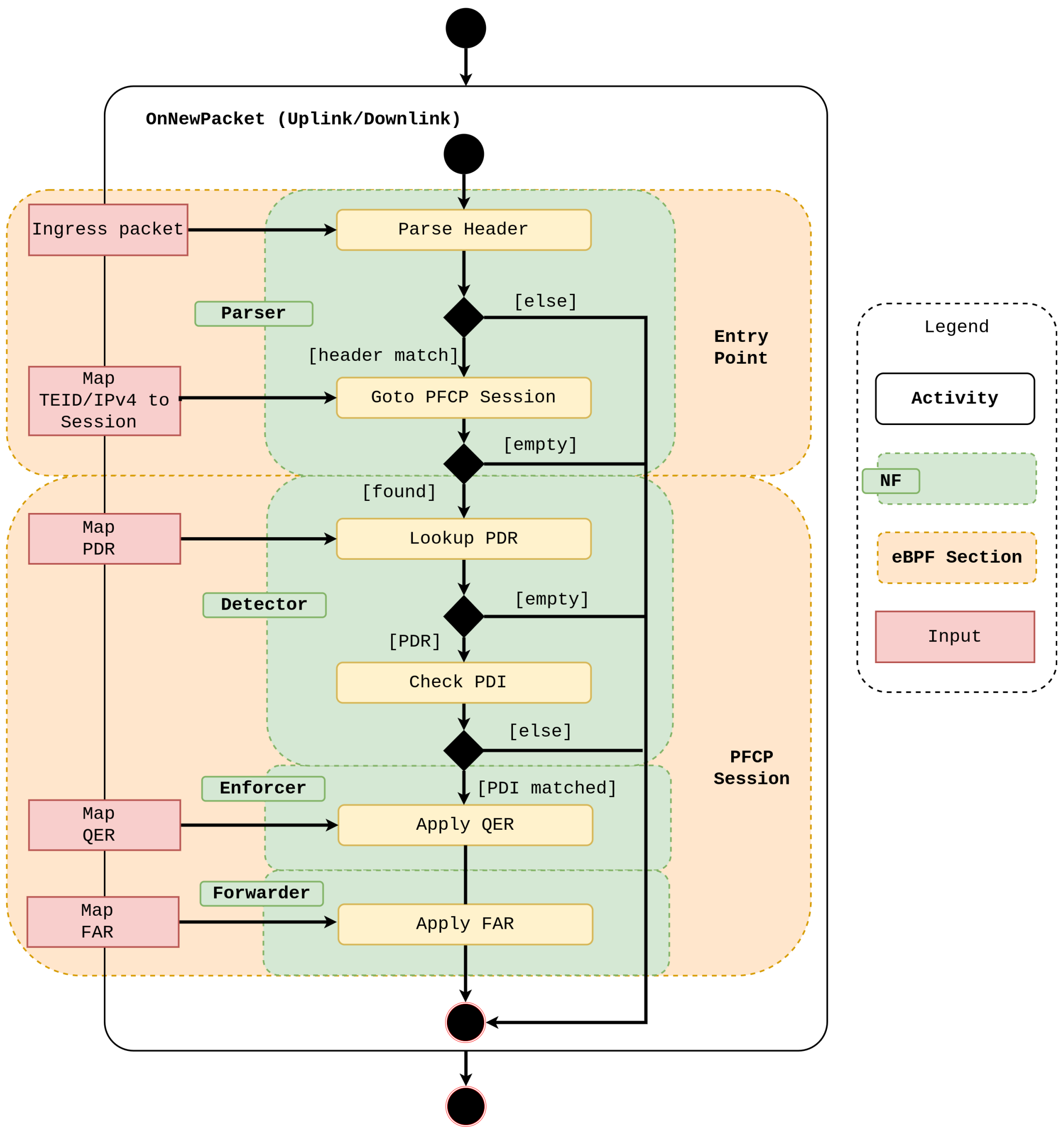


Figure 5.6 – Workflow in Datapath Layer with QER and FAR (version 1).

Management Layer

The main difference with the previous version is that instead of loading all the PFCP session context on the Datapath Layer, these information are stored in the Management Layer and only the highest precedence PDR along with the associated rules for each PFCP session will be deployed in the Datapath Layer. Figure 5.7 shows the activity diagram of the PFCP session creation use case.

Datapath Layer

The main difference between the previous version is that instead of mapping one BPF program to one PFCP session, this version maps one BPF program for each rule defined in highest precedence PDR (e.g. FAR) for each PFCP session created. For instance, if one PFCP session is composed of two PDRs and the high precedence PDR is

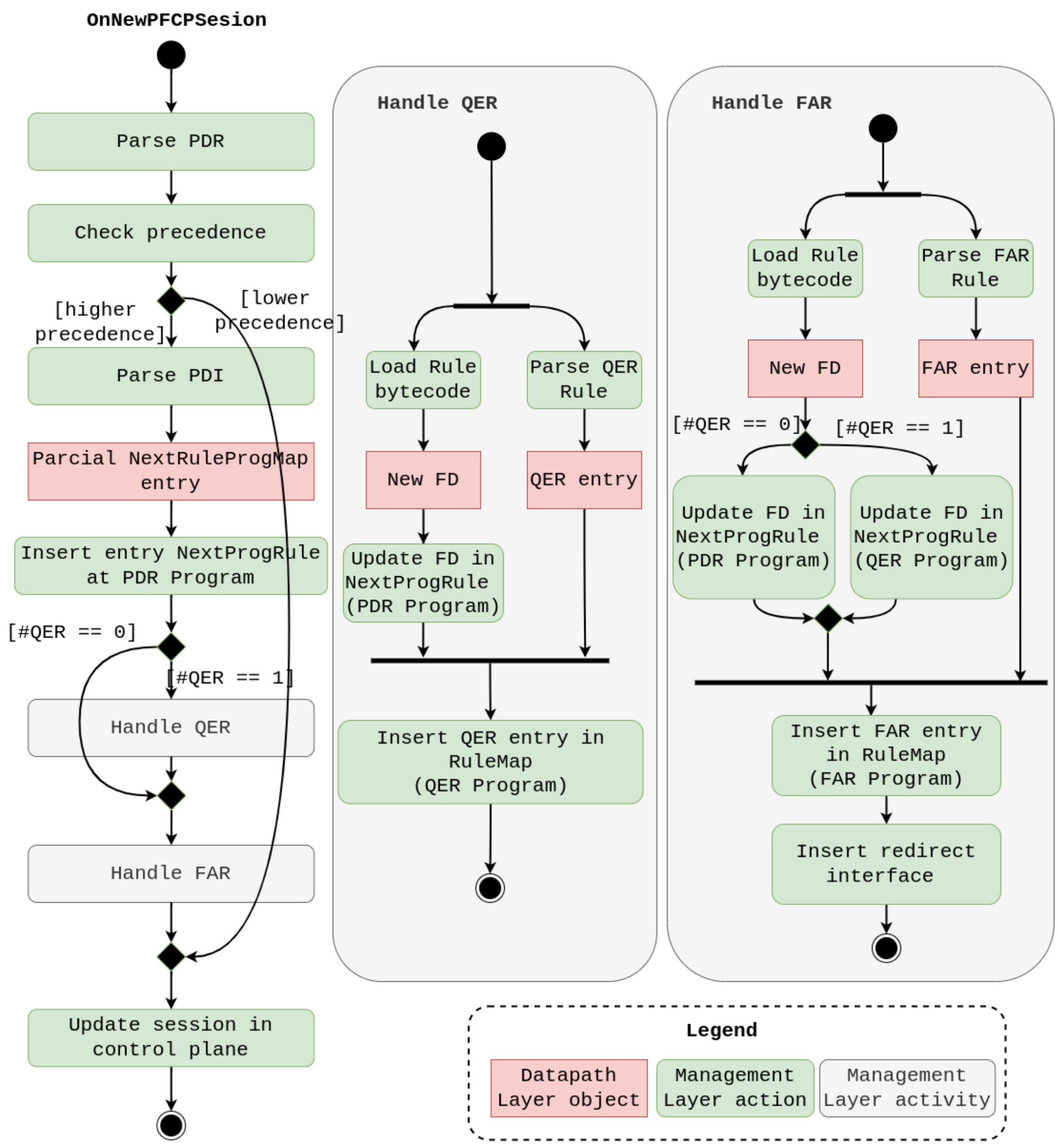

OnNewPFCPSesion
Parse PDR
Check precedence
[higher precedence]
[lower precedence]
Parse PDI
Parcial NextRuleProgMap entry
Insert entry NextProgRule at PDR Program
[#QER == 0]
[#QER == 1]
Handle QER
Handle FAR
Update session in control plane
Handle QER
Load Rule bytecode
Parse QER Rule
New FD
QER entry
Update FD in NextProgRule (PDR Program)
Insert QER entry in RuleMap (QER Program)
Handle FAR
Load Rule bytecode
Parse FAR Rule
New FD
FAR entry
[#QER == 0]
[#QER == 1]
Update FD in NextProgRule (PDR Program)
Update FD in NextProgRule (QER Program)
Insert FAR entry in RuleMap (FAR Program)
Insert redirect interface
Legend
Datapath Layer object
Management Layer action
Management Layer activity

Figure 5.7 – PFCP session creation activity diagram in Management Layer (version 2).

composed of one QER and one FAR and the other one is composed of one FAR, then two BPF program will be deployed on the Datapath Layer, one for the QER and another for the FAR related to the PDR with the highest precedence. The main advantage of following this approach is to avoid the PDR loop (Figure 5.5). Furthermore, the BPF programs are more decoupled which can be easily extensible for new rules (e.g. QER, URR). The pipeline is also more flexible, which can be changed based on the rules contained in the PFCP session. Figure 5.8 shows the activity diagram for the OnNewPacket use case. In this case, the Datapath layer supports only FAR and QER. The activity diagram presented on Figure 5.9 is more generic. The xRule can be one of the 5G rules described in Chapter 2. The xRule is an abstraction of the one rule associated with the highest precedence PDR. If there are more than one rule, the xRule will be chained.

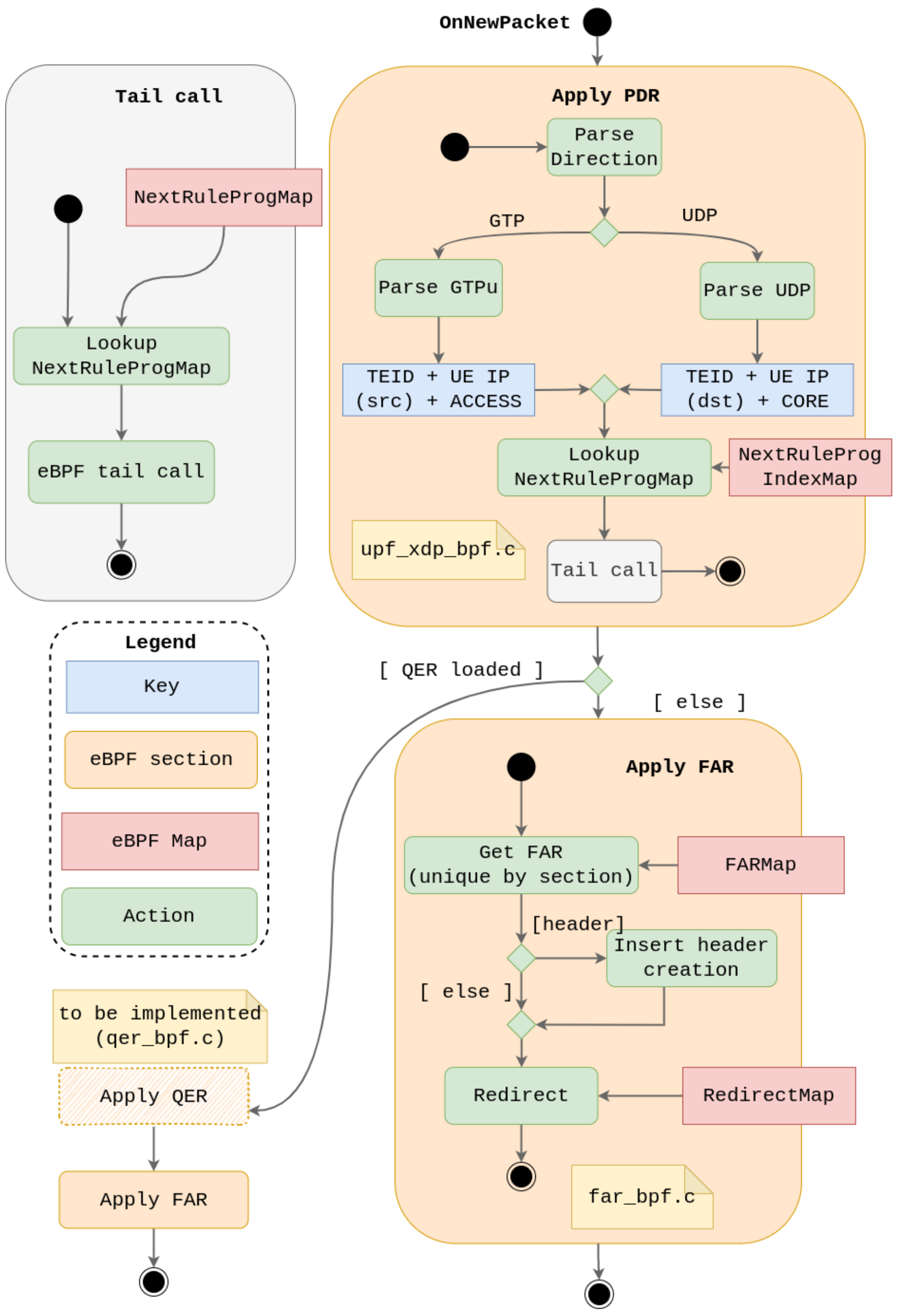


Figure 5.8 – On new packet received activity diagram in Datapath Layer (version 2) .

5.3 Implementation

The components were developed in C++ (Management Layer) and restrict C (Datapath Layer). The source code is available through Apache-2.0 License. The UTs of the PFCP Session Manager component were implemented using Google Testing framework. Besides, a HTTP API was developed for end-to-end tests that will be presented in the next chapter. This API simulates the PFCP control plane and wrappers the upf-bpf library. So, when a HTTP request is received, the JSON message is converted to the library structures

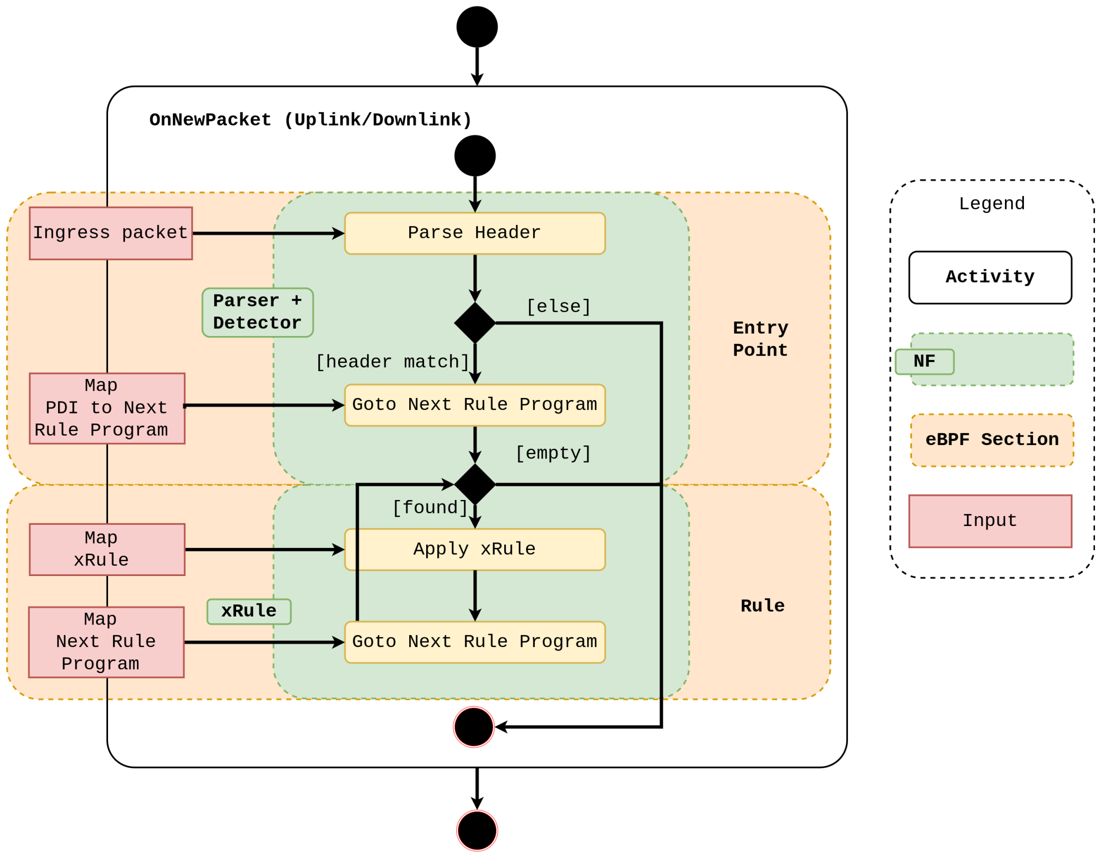


Figure 5.9 – On new packet activity diagram with the decoupled rules (version 2).

which are passed via function call to Session Manager API. It is important to highlight that the same structures used in the Management Layer are also used in the Datapath Layer and were based on the data model as described in Figure 5.1. The Datapath Layer supports both XDP mode: XDP driver and XDP generic. The end-to-end performance evaluation was developed in python using Trex Stateless API (CISCO, 2022b) and will be described in the next chapter. All the tests were automated and the report of the test was generated automatically. Some of the challenges and limitations faced during the development of the BPF/XDP program are well addressed in (Miano et al., 2018). The following sections will describe the code that was developed based on the use cases: set up UPF, create PFCP session and on new packet (uplink/downlink).

5.3.1 Setup UPF

The initial setup of the upf-bpf is provided by the *setup* method in UserPlaneComponent class. As shown, the method receives the name of the interface that represents the N3 and N6 reference points. The method deploys BPF entry point section (Figure 5.9) using the UPFProgram class (line 16).

```cpp
void UserPlaneComponent::setup(std::shared_ptr<RulesUtilities> pRulesUtilities, const
↪  std::string& gtpInterface, const std::string& udpInterface)
{
  LOG_FUNC();

  mpRulesUtilities = pRulesUtilities;
  mGTPInterface = gtpInterface;
  mUDPInterface = udpInterface;
  mpUPFProgram = std::make_shared<UPFProgram>(gtpInterface, udpInterface);

  if(!mpUPFProgram) {
    LOG_ERROR("Program not initialized");
    throw std::runtime_error("Program not initialized");
  }

  SignalHandler::getInstance().enable();
  mpUPFProgram->setup();
}
```

The entry point section implements the logic of the *Apply PDR* activity as shown in Figure 5.8. Both parser (GTPu and UDP) are presented on the snippet code below.

The downlink data processing starts when the datapath receives the UDP packet. If the destination port is different of 2152, then the packet is sent to another BPF program via tail call only if the TEID (0 for all UDP packets), source interface (INTERFACE_VALUE_CORE) and destination IP address (line 28 in *udp_handle* function) match with some entry in the BPF map (line 12 in *tail_call_next_prog* function). The next programs of the chain represents the rules that will be applied on the packet, for instance the FAR.

The uplink data processing follows the same logic. However, the GTPu packet is received instead of UDP. The destination port must match with 2152, which is defined in (ESTI, 2020c) (line 26 in *udp_handle* function). In this case, the *gtp_handle* function handles the GTP packet. This function check the type of GTP message (line 17) and the inner packet (line 24). Finally, *tail_call_next_prog* (line 30) is called passing the TEID presented in the GTP header, the source interface (INTERFACE_VALUE_ACCESS) and the destination IP address.

```c
1  /**
2   * @brief Handle UDP header.
3   *
4   * @param p_ctx The user accessible metadata for xdp packet hook.
```

```c
 * @param udph The UDP header.
 * @return u32 The XDP action.
 */
static u32 udp_handle(struct xdp_md *p_ctx, struct udphdr *udph, u32 dest_ip)
{
  void *p_data_end = (void *)(long)p_ctx->data_end;
  struct next_rule_prog_index_key map_key;
  u32 index_prog;
  u32 dport;

  /* Hint: +1 is sizeof(struct udphdr) */
  if((void *)udph + sizeof(*udph) > p_data_end) {
    bpf_debug("Invalid UDP packet");
    return XDP_ABORTED;
  }

  bpf_debug("UDP packet validated");
  dport = htons(udph->dest);

  switch(dport) {
  case GTP_UDP_PORT:
    return gtp_handle(p_ctx, (struct gtpuhdr *)(udph + 1), dest_ip);
  default:
    tail_call_next_prog(p_ctx, 0, INTERFACE_VALUE_CORE, dest_ip);

    return XDP_PASS;
  }
}
```

```c
/**
 * @brief Check if GTP packet is a GPDU. If so, process the next block chain.
 *
 * @param p_ctx The user accessible metadata for xdp packet hook.
 * @param p_gtpuh The GTP header.
 * @return u32 The XDP action.
 */
static u32 gtp_handle(struct xdp_md *p_ctx, struct gtpuhdr *p_gtpuh, u32 dest_ip)
{
  void *p_data_end = (void *)(long)p_ctx->data_end;

  if((void *)p_gtpuh + sizeof(*p_gtpuh) > p_data_end) {
    bpf_debug("Invalid GTPU packet");
    return XDP_DROP;
  }

  if(p_gtpuh->message_type != GTPU_G_PDU) {
    bpf_debug("Message type 0x%x is not GTPU GPDU(0x%x)", p_gtpuh->message_type,
    ↪  GTPU_G_PDU);
    return XDP_PASS;
  }

  bpf_debug("GTP GPDU received");

  if(!ip_inner_check_ipv4(p_ctx, (struct iphdr *)(p_gtpuh + 1))) {
    bpf_debug("Invalid IP inner");
    return XDP_DROP;
  }

  // Jump to session context.
  tail_call_next_prog(p_ctx, p_gtpuh->teid, INTERFACE_VALUE_ACCESS, dest_ip);
  bpf_debug("BPF tail call was not executed! teid %d\n", htonl(p_gtpuh->teid));

  return XDP_PASS;
}
```

```
static u32 tail_call_next_prog(struct xdp_md *p_ctx, teid_t_ teid, u8 source_value,
↪  u32 ipv4_address)
{
  struct next_rule_prog_index_key map_key;
  u32 *index_prog;

  __builtin_memset(&map_key, 0, sizeof(struct next_rule_prog_index_key));

  map_key.teid = teid;
  map_key.source_value = INTERFACE_VALUE_CORE;
  map_key.ipv4_address = ipv4_address;
  bpf_debug("map key teid: %d, source: %d, ip: %d \n", teid, INTERFACE_VALUE_CORE,
  ↪  ipv4_address);
  index_prog = bpf_map_lookup_elem(&m_next_rule_prog_index, &map_key);

  if(index_prog){
    bpf_debug("BPF tail call to %d key\n", *index_prog);
    bpf_tail_call(p_ctx, &m_next_rule_prog, *index_prog);
    bpf_debug("BPF tail call was not executed!\n");
  }
  return 0;
}
```

5.3.2 Create PFCP Session

When a PFCP session is created, the method *createBFPSession* is called from SessionManager class. This method finds the highest precedence PDR (line 10) that will be deployed on the Datapath Layer along with the rules. The *createPipeline* method (line 39) loads the BPF programs in kernel space.

```
 1  void SessionManager::createBPFSession(std::shared_ptr<pfcp::pfcp_session> pSession)
 2  {
 3    LOG_FUNC();
 4    LOG_DBG("Session {} received... Lets prepare the UP", pSession->get_up_seid());
 5
 6    LOG_DBG("Find the PDR with highest precedence");
 7    // The lower precedence values indicate higher precedence of the PDR, and the
 8    // higher precedence values indicate lower precedence of the PDR when matching
 9    // a packet.
10    std::sort(pSession->pdrs.begin(), pSession->pdrs.end(),
      ↪  SessionManager::comparePDR);
11
12    LOG_DBG("Extract the key (PDI) from the highest priority PDR");
13    auto pUPFProgram = UserPlaneComponent::getInstance().getUPFProgram();
14
15    pfcp::pdi pdi;
16    pfcp::fteid_t fteid;
17    pfcp::ue_ip_address_t ueIpAddress;
18    pfcp::source_interface_t sourceInterface;
19
20    if (pSession->pdrs.empty()){
21      LOG_ERROR("No PDR was found in session %d", pSession->seid);
22      throw std::runtime_error("No PDR was found in session");
23    }
24
25    auto pdrHighPriority = pSession->pdrs[0];
26    if(!(pdrHighPriority->get(pdi) && pdi.get(fteid) && pdi.get(sourceInterface) &&
      ↪  pdi.get(ueIpAddress))) {
27      throw std::runtime_error("No fields available");
28    }
29    LOG_DBG("PDI extracted from PDR {}", pdrHighPriority->pdr_id.rule_id);
30
31    LOG_DBG("Extract FAR from the highest priority PDR");
32    std::shared_ptr<pfcp::pfcp_far> pFar;
33    pfcp::far_id_t farId;
34
35    if (!(pdrHighPriority->get(farId) && pSession->get(farId.far_id, pFar))){
36      throw std::runtime_error("No fields available");
37    }
38
39    SessionProgramManager::getInstance().createPipeline(pSession->get_up_seid(),
      ↪  fteid.teid, sourceInterface.interface_value, ueIpAddress.ipv4_address.s_addr,
      ↪  pFar);
40
41    LOG_DBG("Add session");
42    mSeidToSession[pSession->get_up_seid()] = pSession;
43  }
```

The *createPipeline* parameters are composed of the PDI, i.e. TEID, source interface, UE IP address (Figure 5.1). As mentioned before, this information is retrieved to check if the hearder of the packet (UDP or GTP) matches with theses information. The PDI is stored in the BPF map of the UPFProgram (line 20). The file descriptor of the FAR BPF program is also stored in the BPF map (line 21). The FAR BPF program is loaded into the kernel (line 13) and the context of the FAR rule is stored inside of the BPF maps of this program (line 31). It is worth mentioning that the pipeline can be extended to support new rule (e.g. QER).

```
1   void SessionProgramManager::createPipeline(uint32_t seid, uint32_t teid, uint8_t
    ↪  sourceInterface, uint32_t ueIpAddress,
2                                              std::shared_ptr<pfcp::pfcp_far> pFar)
3   {
4     LOG_FUNC();
5     struct next_rule_prog_index_key key = {.teid = teid, .source_value =
      ↪  sourceInterface, .ipv4_address = ueIpAddress};
6     u32 id;
7     s32 fd;
8     LOG_DBG("teid: {}, source interface: {}, ue ip: {}", teid, sourceInterface,
      ↪  ueIpAddress);
9
10    LOG_DBG("Instantiate a new FARProgram");
11    // Instantiate a new FARProgram
12    std::shared_ptr<FARProgram> pFARProgram = std::make_shared<FARProgram>();
13    pFARProgram->setup();
14
15    LOG_DBG("Store FARProgram index in the UPFProgram");
16    auto pUPFProgram = UserPlaneComponent::getInstance().getUPFProgram();
17    id = pFARProgram->getId();
18    fd = pFARProgram->getFd();
19
20    pUPFProgram->getNextProgRuleIndexMap()->update(key, id, BPF_ANY);
21    pUPFProgram->getNextProgRuleMap()->update(id, fd, BPF_ANY);
22
23    LOG_DBG("Store FAR in the FAR program");
24    uint8_t index = 0;
```

```

  pfcp_far_t_ far = // FAR ID... HIDDEN, check the source code

  // Fwd - actions
  memcpy(&far.apply_action, &pFar->apply_action, sizeof(apply_action_t_));

  pFARProgram->getFARMap()->update(index, far, BPF_ANY);

  // Map the pipeline deployed to the seid. The seid will be used to detroyed it.
  mSessionProgramsMap[seid] = std::make_shared<SessionPrograms>(key, pFARProgram);
}
```

5.3.3 On New Packet

This section is a continuation of what was presented in 5.3.1. In this case, we assume that there will be a FAR program running in the Datapath Layer. When the header of the received packet matches with the information stored in the BPF map (entry point), then the tail call will be successful and the FAR BPF program is executed. The main logic of the FAR BPF program is shown below. The *pfcp_far_apply* function apply the forward action (line 25) in two different ways, depending on the destination interface defined in FAR. The destination interface can be to the core network (line 26) or to the access network (line 65). Before redirecting the packet to the destination interface, the outer header creation is checked. For uplink packets (GTP), if the outer header creation is set to be UDP, the GTP header is discarded (line 53) and the packet is redirected to the destination interface (line 55). On the other hand, for downlink packets (UDP), if the outer header creation is set to be GTP (line 69), the IP, UDP and GTP header is created (line 103-113, line 121-124 and line 147-149, respectively). For uplink and downlink, the MAC address of the next hop is also updated (line 48 and line 134). Finally, the packet is redirect to the destination interface (line 55 and line 156).

```
1  static u32 pfcp_far_apply(struct xdp_md *p_ctx, pfcp_far_t_ *p_far, enum
   ↪  FlowDirection direction)
2  {
3    void *p_data = (void *)(long)p_ctx->data;
4    void *p_data_end = (void *)(long)p_ctx->data_end;
5    struct ethhdr *p_eth = p_data;
6    void *p_mac_address;
7
8    u8 dest_interface;
9    u16 outer_header_creation;
10
11   if((void *)(p_eth + 1) > p_data_end) {
    bpf_debug("Invalid pointer");
    return XDP_DROP;
  }

  if(!p_far) {
    bpf_debug("Invalid FAR!");
    return XDP_DROP;
  }

  dest_interface =
  ↪  p_far->forwarding_parameters.destination_interface.interface_value;
  outer_header_creation = p_far->forwarding_parameters.outer_header_creation
  ↪  .outer_header_creation_description;

  // Check if it is a forward action.
  if(p_far->apply_action.forw) {
    if(dest_interface == INTERFACE_VALUE_CORE) {
      // Redirect to data network.
      bpf_debug("Destination is to INTERFACE_VALUE_CORE\n");
      // Check Outer header creation - IPv4 or IPv6
      switch(outer_header_creation) {
      case OUTER_HEADER_CREATION_UDP_IPV4:
        bpf_debug("OUTER_HEADER_CREATION_UDP_IPV4\n");
        struct ethhdr *p_new_eth = p_data + GTP_ENCAPSULATED_SIZE;

        // Move eth header forward.
        if((void *)(p_new_eth + 1) > p_data_end) {
          return 1;
        }
        __builtin_memcpy(p_new_eth, p_eth, sizeof(*p_eth));

        // Update destination mac address.
        struct iphdr *p_ip = (void *)(p_new_eth + 1);

        if((void *)(p_ip + 1) > p_data_end) {
          return XDP_DROP;
        }

        if(update_dst_mac_address(p_ip, p_new_eth)) {
          return XDP_DROP;
        }

        // Adjust head to the right.
        bpf_xdp_adjust_head(p_ctx, GTP_ENCAPSULATED_SIZE);

        return bpf_redirect_map(&m_redirect_interfaces, direction, 0);
        bpf_debug("OUTER_HEADER_CREATION_UDP_IPV4 REDIRECT FAILED\n");
        break;
      case OUTER_HEADER_CREATION_UDP_IPV6:
        bpf_debug("OUTER_HEADER_CREATION_UDP_IPV6\n");
        // TODO
        break;
      default:
        bpf_debug("In destination to CORE - Invalid option: %d",
        ↪  outer_header_creation);
      }
    } else if(dest_interface == INTERFACE_VALUE_ACCESS) {
      // Redirect to core network.
      bpf_debug("Destination is to INTERFACE_VALUE_ACCESS");
      switch(outer_header_creation) {
      case OUTER_HEADER_CREATION_GTPU_UDP_IPV4:
        bpf_debug("OUTER_HEADER_CREATION_GTPU_UDP_IPV4");
        // Resize the header in order to put the GTP/UPD/IP headers.
        // Adjust space to the left.
        bpf_xdp_adjust_head(p_ctx, (int32_t)-GTP_ENCAPSULATED_SIZE);

        // Packet buffer changed, all pointers need to be recomputed
        p_data = (void *)(long)p_ctx->data;
        p_data_end = (void *)(long)p_ctx->data_end;

        p_eth = p_data;
        if((void *)(p_eth + 1) > p_data_end) {
          bpf_debug("Invalid pointer");
          return XDP_DROP;
        }

        // Space allocated before packet buffer, move eth header
        struct ethhdr *p_orig_eth = p_data + GTP_ENCAPSULATED_SIZE;
        if((void *)(p_orig_eth + 1) > p_data_end) {
          return XDP_DROP;
        }
        memcpy(p_eth, p_orig_eth, sizeof(*p_eth));

        p_ip = (void *)(p_eth + 1);
        if((void *)(p_ip + 1) > p_data_end) {
          return XDP_DROP;
        }

        struct iphdr *p_inner_ip = (void *)p_ip + GTP_ENCAPSULATED_SIZE;
        if((void *)(p_inner_ip + 1) > p_data_end) {
          return XDP_DROP;
        }

        // Add the outer IP header
        p_ip->version = 4;
        p_ip->ihl = 5; // No options
        p_ip->tos = 0;
        p_ip->tot_len = htons(ntohs(p_inner_ip->tot_len) + GTP_ENCAPSULATED_SIZE);
        p_ip->id = 0;              // No fragmentation
        p_ip->frag_off = 0x0040; // Don't fragment; Fragment offset = 0
        p_ip->ttl = 64;
        p_ip->protocol = IPPROTO_UDP;
        p_ip->check = 0;
        p_ip->saddr = LOCAL_IP;
        p_ip->daddr =
        ↪  p_far->forwarding_parameters.outer_header_creation.ipv4_address.s_addr;

        // Add the UDP header
        struct udphdr *p_udp = (void *)(p_ip + 1);
        if((void *)(p_udp + 1) > p_data_end) {
          return XDP_DROP;
        }

        p_udp->source = htons(GTP_UDP_PORT);
        p_udp->dest =
        ↪  htons(p_far->forwarding_parameters.outer_header_creation.port_number);
        p_udp->len = htons(ntohs(p_inner_ip->tot_len) + sizeof(*p_udp) +
        ↪  sizeof(struct gtpuhdr));
        p_udp->check = 0;

        bpf_debug("Destination MAC:%x:%x:%x\n", p_eth->h_dest[0], p_eth->h_dest[1],
        ↪  p_eth->h_dest[2]);
        bpf_debug("Destination MAC:%x:%x:%x\n", p_eth->h_dest[3], p_eth->h_dest[4],
        ↪  p_eth->h_dest[5]);
        p_mac_address = bpf_map_lookup_elem(&m_arp_table, &p_ip->daddr);
        if(!p_mac_address) {
          bpf_debug("mac not found!!\n");
          return XDP_DROP;
        }
        // swap_src_dst_mac(p_data);
        memcpy(p_eth->h_dest, p_mac_address, sizeof(p_eth->h_dest));
        bpf_debug("Destination MAC:%x:%x:%x\n", p_eth->h_dest[0], p_eth->h_dest[1],
        ↪  p_eth->h_dest[2]);
        bpf_debug("Destination MAC:%x:%x:%x\n", p_eth->h_dest[3], p_eth->h_dest[4],
        ↪  p_eth->h_dest[5]);
        bpf_debug("Destination IP:%d Port:%d\n", p_ip->daddr,
        ↪  p_far->forwarding_parameters.outer_header_creation.port_number);

        // Add the GTP header
        struct gtpuhdr *p_gtpuh = (void *)(p_udp + 1);
        if((void *)(p_gtpuh + 1) > p_data_end) {
          return XDP_DROP;
        }

        u8 flags = GTP_FLAGS;
        memcpy(p_gtpuh, &flags, sizeof(u8));
        p_gtpuh->message_type = GTPU_G_PDU;
        p_gtpuh->message_length = p_inner_ip->tot_len;
        p_gtpuh->teid = p_far->forwarding_parameters.outer_header_creation.teid;

        // Compute l3 checksum
        __wsum l3sum = pcn_csum_diff(0, 0, (__be32 *)p_ip, sizeof(*p_ip), 0);
        pcn_l3_csum_replace(p_ctx, IP_CSUM_OFFSET, 0, l3sum, 0);

        bpf_debug("GTPU header were pushed!");
        return bpf_redirect_map(&m_redirect_interfaces, direction, 0);
        break;
      case OUTER_HEADER_CREATION_GTPU_UDP_IPV6:
        bpf_debug("OUTER_HEADER_CREATION_GTPU_UDP_IPV6");
        break;
      default:
        bpf_debug("In destination to ACCESS - Invalid option: %d",
        ↪  outer_header_creation);
      }
    }
  } else {
    bpf_debug("Forward action unset");
  }
  return XDP_PASS;
}
```

6 Testing and Performance Evaluation

This chapter presents the testing and performance evaluation of the proposed solution. In this scenario, a XDP native mode was chosen to achieve the maximum of the performance.

6.1 Setup

We have tested the proposed solution taking RFC2544-like measurements. The testbed is composed of two server Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 32GiB of the DRAM, 15M of L3 cache, 6 cores (hyper-threading disabled), dual-port 82599ES 10-Gigabit SFI/SFP+ NIC. Both machines have Ubuntu 20.04.02 LTS installed with Linux kernel v5.4.0-72-generic compiled with BTF flags enabled. One machine is used to generate user traffic with TRex Traffic Generator (CISCO, 2022a) and the other is the DUT (Device Under Test) where the proposed solution is deployed. The hyperthread was disabled[1] in the DUT machine. The setup is shown in Figure 6.1.

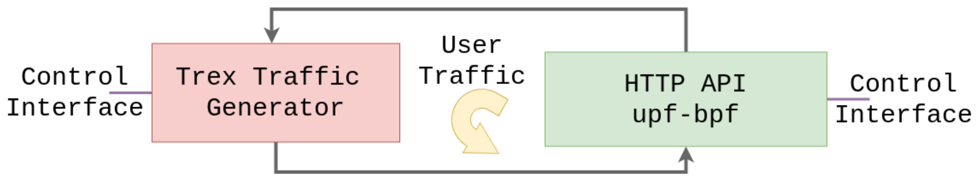
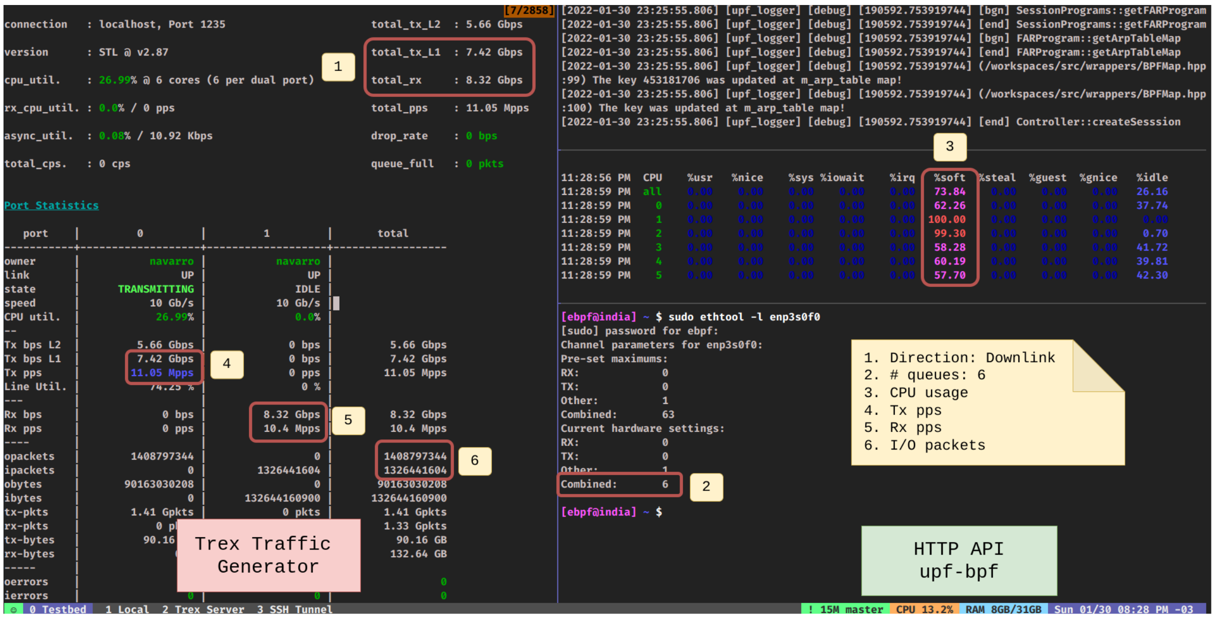


Figure 6.1 – Testbed setup.

The execution of the test can be done manually or automatically. For the manual execution, a tmux session script[2] was implemented to execute custom test case quicker. The main window is shown in Figure 6.2. The window is divided into four panes. The left pane is logged into the Trex Traffic Generator server and all the right panes are logged into DUT (HTTP API + upf-bpf). The first pane in the left presents the terminal user interface, which contains statistics about the transmission and the reception, e.g. throughput and number of packets. The first pane on the right is used to execute the http-api program and for output analysis. The middle one is used to run the mpstat program for CPU utilization analysis. Finally, the last pane in the right is used to check the number of queues that are set in the NIC. All these steps are executed automatically.

[1] echo off > /sys/devices/system/cpu/smt/control

[2] <https://github.com/navarrothiago/upf-bpf/blob/7e7c3e70c59f173e390eca1ce7a65a65d3a8f032/tests/scripts/start_session>



```
                                                                     [7/2858]
connection   : localhost, Port 1235                          total_tx_L2  : 5.66 Gbps

version      : STL @ v2.87                                   total_tx_L1  : 7.42 Gbps

cpu_util.    : 26.99% @ 6 cores (6 per dual port)            total_rx     : 8.32 Gbps

rx_cpu_util. : 0.0% / 0 pps                                  total_pps    : 11.05 Mpps

async_util.  : 0.08% / 10.92 Kbps                            drop_rate    : 0 bps

total_cps.   : 0 cps                                         queue_full   : 0 pkts


Port Statistics

   port    |         0         |         1         |       total
-----------+-------------------+-------------------+-----------------
owner      |           navarro |           navarro |
link       |                UP |                UP |
state      |      TRANSMITTING |              IDLE |
speed      |           10 Gb/s |           10 Gb/s |
CPU util.  |            26.99% |              0.0% |
--         |                   |                   |
Tx bps L2  |         5.66 Gbps |             0 bps |         5.66 Gbps
Tx bps L1  |         7.42 Gbps |             0 bps |         7.42 Gbps
Tx pps     |        11.05 Mpps |             0 pps |        11.05 Mpps
Line Util. |           74.25 % |               0 % |
---        |                   |                   |
Rx bps     |             0 bps |         8.32 Gbps |         8.32 Gbps
Rx pps     |             0 pps |         10.4 Mpps |         10.4 Mpps
----       |                   |                   |
opackets   |        1408797344 |                 0 |        1408797344
ipackets   |                 0 |        1326441604 |        1326441604
obytes     |       90163030208 |                 0 |       90163030208
ibytes     |                 0 |      132644160900 |      132644160900
tx-pkts    |        1.41 Gpkts |            0 pkts |        1.41 Gpkts
rx-pkts    |              0 p                               1.33 Gpkts
tx-bytes   |          90.16                                   90.16 GB
rx-bytes   |                                                 132.64 GB
-----      |
oerrors    |                                                          0
ierrors    |                 0 |                 0 |                0
```

```
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] [bgn] SessionPrograms::getFARProgram
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] [end] SessionPrograms::getFARProgram
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] [bgn] FARProgram::getArpTableMap
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] [end] FARProgram::getArpTableMap
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] (/workspaces/src/wrappers/BPFMap.hpp
:99) The key 453181706 was updated at m_arp_table map!
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] (/workspaces/src/wrappers/BPFMap.hpp
:100) The key was updated at m_arp_table map!
[2022-01-30 23:25:55.806] [upf_logger] [debug] [190592.753919744] [end] Controller::createSesssion
```

```
11:28:56 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
11:28:59 PM  all    0.00    0.00    0.00    0.00    0.00   73.84    0.00    0.00    0.00   26.16
11:28:59 PM    0    0.00    0.00    0.00    0.00    0.00   62.26    0.00    0.00    0.00   37.74
11:28:59 PM    1    0.00    0.00    0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00
11:28:59 PM    2    0.00    0.00    0.00    0.00    0.00   99.30    0.00    0.00    0.00    0.70
11:28:59 PM    3    0.00    0.00    0.00    0.00    0.00   58.28    0.00    0.00    0.00   41.72
11:28:59 PM    4    0.00    0.00    0.00    0.00    0.00   60.19    0.00    0.00    0.00   39.81
11:28:59 PM    5    0.00    0.00    0.00    0.00    0.00   57.70    0.00    0.00    0.00   42.30
```

```
[ebpf@india] ~ $ sudo ethtool -l enp3s0f0
[sudo] password for ebpf:
Channel parameters for enp3s0f0:
Pre-set maximums:
RX:             0
TX:             0
Other:          1
Combined:       63
Current hardware settings:
RX:             0
TX:             0
Other:          1
Combined:       6

[ebpf@india] ~ $
```

```
0 Testbed   1 Local   2 Trex Server   3 SSH Tunnel        ! 15M master  CPU 13.2%  RAM 8GB/31GB  Sun 01/30 08:28 PM -03
```

Figure 6.2 – Tmux session for manual test execution.

6.2 Test Case

We created a test case to evaluate the scalability and the workload when the solution achieves the maximum throughput for a specific traffic generation flow. Both uplink and downlink scenarios were tested. The tests consisted in:

1. Run Trex Traffic Generator server;
2. Run HTTP API (DUT);
3. Send a PFCP session establishment request via HTTP API to DUT;
4. Configure the number of Rx queue (LINUX, 2022) in DUT;
5. Generate traffic (GTP or UDP) using Trex Traffic Stateless API;
6. Collect workload per core (DUT) and throughput (Trex Traffic Generator) metrics:

For step 3, a control message was designed to define the PFCP session establishment request to be sent to DUT. This message contains the PDRs (uplink and downlink) and FARs IEs and the static ARP table of the next hop. The PFCP session context message (JSON) is shown in Listing 1.

For steps 4, 5 and 6, it was implemented a Python script[3] to automate the process. The script executes the test case varying the number of the rx queue. In the end,

[3] <https://github.com/navarrothiago/upf-bpf/blob/c1250469a101a10c4b7ac38503a6edda6c5ca1f1/tests/trex/test_cases/run.py>

a report based on JSON format with all the metrics (i.e throughput and CPU load) for each execution is generated. The code below shows its main functionalities.

```python
def run_ethtool_set_rx_queue(number_rx_queue, password):
    print("Setting NIC rx queue size...")
    ifaces = ["enp3s0f0", "enp3s0f1"]
    dicIface = {}
    for iface in ifaces:
        dicIface[iface] = int(number_rx_queue)
        cmd = 'echo {} | ssh india sudo -S ethtool -L {} combined {}'.format(
            password, iface, int(number_rx_queue))
        print("Running ethtool... {}".format(cmd))
        os.popen(cmd)
        cmd = 'echo {} | ssh india sudo -S ethtool -l {} '.format(
            password, iface)
        print("Running ethtool... {}".format(cmd))
        print(os.popen(cmd).read())
    item["ethtool"] = dicIface
    print("Setting NIC rx queue size...DONE")
    print()

def create_udp_pkt_flow(size, ip_min, ip_max, nflows, field):
    print("{} flow will be generated...".format(nflows))
    base_pkt = Ether()/IP(src="16.0.0.1", dst="10.1.3.27")/UDP(dport=1234)
    pad = max(0, size - len(base_pkt)) * 'x'
    return STLPktBuilder(pkt=base_pkt/pad, vm=create_vm(ip_min, ip_max, nflows,
    ↪  field))

def create_gtp_pkt_flow(size, ip_min, ip_max, nflows, field):
    print("{} flow will be generated...".format(nflows))
    base_pkt = Ether()/IP(src="172.20.16.99", dst="192.168.15.12")/UDP(dport=2152) /
    ↪  \
        GTP_U_Header(teid=100) / \
        IP(src="10.10.10.10", dst="10.1.3.27", version=4)/UDP(dport=1234)
    pad = max(0, size - len(base_pkt)) * 'x'
    return STLPktBuilder(pkt=base_pkt/pad, vm=create_vm(ip_min, ip_max, nflows,
    ↪  field))

def create_vm(ip_min, ip_max, nflows, field):
    vm = STLVM()

    # create a tuple var
    vm.tuple_var(name="tuple", ip_min=ip_min, ip_max=ip_max,
                 port_min=1234, port_max=1234, limit_flows=nflows)

    # write fields
    vm.write(fv_name="tuple.ip", pkt_offset="IP.{}".format(field))
    vm.fix_chksum()

    # vm.write(fv_name="tuple.port", pkt_offset="UDP.sport")
    return vm

def simple_burst(streams, m, duration):
    # create client
    # c = STLClient()
    # username/server can be changed those are the default
    # username = common.get_current_user(),
    # server = "localhost"
    c = STLClient(server="localhost", sync_port=1235, async_port=1236)
    passed = True

    try:
        # connect to server
        c.connect()

        # prepare our ports (my machine has 0 <--> 1 with static route)
        # Acquire port 0 for $USER
        c.reset(ports=[0, 1])

        # add both streams to ports
        c.add_streams(streams, ports=[0])

        # clear the stats before injecting
        c.clear_stats()

        # set port 1 as promiscuous mode
        c.set_port_attr(ports=[1], promiscuous=True)

        # choose rate and start traffic for 10 seconds on 5 mpps
        print("Running " + m + " on ports 0 for {} seconds...".format(duration))
        c.start(ports=[0], mult=m, duration=duration)
        run_mpstat(duration/2)

        # block until done
        c.wait_on_traffic(ports=[0])

        # read the stats after the test
        stats = c.get_stats()

        item["throughput"] = float(stats[1]["rx_pps"])/1000000
        item["loss"] = float(stats[0]["opackets"] -
        ↪  stats[1]["ipackets"])/stats[0]["opackets"]
        print("")
        print("Throughput: {} Mpps".format(item["throughput"]))
```

```
88
89          if (stats[0]["opackets"] > 100):
90              passed = True
91          else:
92              passed = False
93
94      except STLError as e:
95          passed = False
96          print(e)
97
98      finally:
99          c.disconnect()
100
101     if passed:
102         print("\nTest has passed :-)\n")
103     else:
104         print("\nTest has failed :-(\n")
105
106 def run_mpstat(duration):
107     global current_test
108     cmd = 'ssh india mpstat -P ALL {} 1 -o JSON'.format(int(duration))
109     print("Running mpstat... {}".format(cmd))
110     output = os.popen(cmd).read()
111     # print(json.loads(output))
112     item["mpstat"] = json.loads(output)
```

For step 4, the function *run_ethtool_set_rx_queue* was implemented. The command in line 7 is executed in the DUT machine (india) and the application *ethtool* is called to setup the rx queue. The function is called to setup the rx queue to 1 to 6, which the higher value represents the maximum number of cores in the DUT machine.

For step 5, the 1000 flows varying the source IP address randomly were generated using the Field Engine modules available in Trex Stateless API (lines 33 – 45). This technique is used to avoid the assignment to a specific receive queue, distributing the flows between the receive queues in the DUT. So, this improves the throughput due to the load balance between the cores. The GTP or UDP headers are generated depending on the test (uplink or downlink) based on the PFCP Session Establishment Request message. In both cases, the frame size is 64 bytes. The functions *create_udp_pkt_flow* (line 19) and *create_gtp_pkt_flow* were implemented to build the UDP and GTP flows, respectively. The flows are crafted in line 21 and 27. The traffic generation is started after line 75.

For step 6, the function *run_mpstat* was implemented. The command in line 108 is executed in DUT machine (india) and the application *mpstat* is called to collect a report about the statistics of the processors during the execution. Besides, the metrics related to the throughput and the packet loss rate were also collected (lines 84 and 85) from Trex Stateless API. All the metrics are saved in the *item* dictionary.

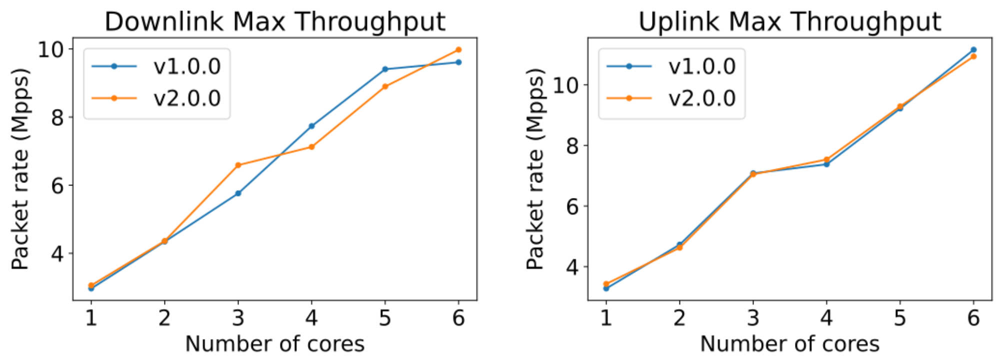
Regarding the throughput measurement, firstly the test case was executed flooding the media, which implied a significant level of loss. The saturation throughput value was found out and, after that, the same test case was executed, but using the saturation value of the throughput in order to achieve almost zero packet loss. The results will be shown in the next section.

```
{
 "seid": 1,
 "pdrs": [ {
     "pdrId": 20,
     "farId": 200,
     "outerHeaderRemoval": "UDP_IPV4",
     "pdi": {
       "teid": 0,
       "sourceInterface": "INTERFACE_VALUE_CORE",
       "ueIPAddress": "10.1.3.27"
     },
     "precedence": 2
   }, {
     "pdrId": 10,
     "farId": 100,
     "outerHeaderRemoval": "GTPU_UDP_IPV4",
     "pdi": {
       "teid": 100,
       "sourceInterface": "INTERFACE_VALUE_ACCESS",
       "ueIPAddress": "10.1.3.27"
     },
     "precedence": 1
   } ],
 "fars": [ {
     "farId": 200,
     "forward": true,
     "forwardingParameters": {
       "outerHeaderCreation": {
         "outerHeaderCreationDescription": "GTPU_UDP_IPV4",
         "ipv4Address": "10.1.3.27",
         "portNumber": 1234
       },
       "destinationInterface": "INTERFACE_VALUE_ACCESS"
     } }, {
     "farId": 100,
     "forward": true,
     "forwardingParameters": {
       "outerHeaderCreation": {
         "outerHeaderCreationDescription": "UDP_IPV4",
         "ipv4Address": "10.1.3.27",
         "portNumber": 1234
       },
       "destinationInterface": "INTERFACE_VALUE_CORE"
     } } ],
 "arpTable": [ {
     "ip": "10.1.2.27",
     "mac": "90:e2:ba:27:fd:3c"
   }, {
     "ip": "10.1.3.27",
     "mac": "90:e2:ba:27:fd:3d"
   } ] }
```

Listing 1 – The JSON message representing the PFCP Session Establishment Request used in the tests.

6.3 Results

Figure 6.3 shows the throughput as a function of the number of cores. The linear function behavior implies that the solution scales with the number of the cores, achieving almost 10 Mpps for downlink and greater than 11 Mpps for uplink. This difference is related to the actions performed in each direction. For the downlink, the UPF encapsulates the GTP header. The packet size is greater than 64 B after reception in the TRex Traffic Generator. On the other hand, for the uplink, the UPF decapsulates the GTP header and the packet size becomes smaller than 64 B. Also, Figure 6.4 shows a histogram regarding the CPU load when varying the numbers of cores. Note that the packet loss is less than 3%. Using 6 CPUs (Figures 6.4k and 6.4l), the average CPU load is about 60% (with almost 40% of the CPU idle, which could be used by other tasks).

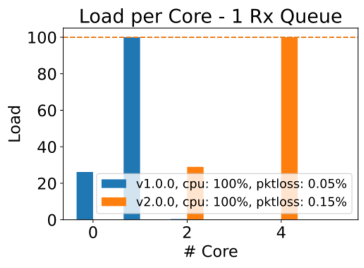


(a) Downlink (b) Uplink

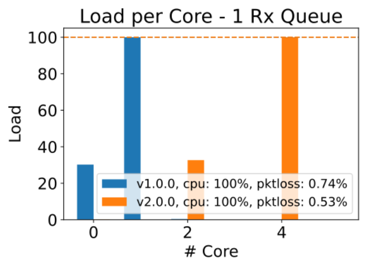
Figure 6.3 – Scalability of the proposed solution.

Figures 6.4g and 6.4i show a higher packet loss. The main reason could be the load distribution between the cores. As we can see, the core #1 load is 100% for both. So, the core did not handle all received packets and some of them may be overwritten from its receive queues. Basically, four factors can contribute to the load distribution: NIC hash algorithm, NIC hash key (tuple), NIC indirection table, and the flows. So, one solution to optimize the load balancing could be creating a custom flow-based indirection table. This analysis is not the scope of this paper. It is important to highlight that our results achieve almost the performance of what was presented in the 5G mobile gateway based on Polycube (Parola, F., 2020), although his work has not presented the CPU usage for the scalability test. Besides, the CPU usage is an important factor to understand the load distribution in each core and it can impact the results of the experiments.

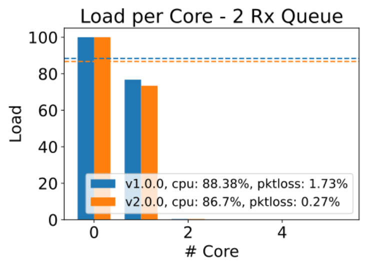
Figures 6.4a and 6.4b show the CPU load when only one core is used. Note that the core #1 is 100% and also there is some task consuming the cycles in core #0. The process could not be identified using *top* command. It has been an unknown issue for a while.



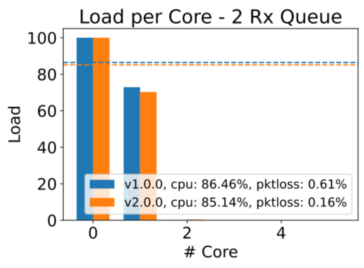
(a) Downlink 1 cores



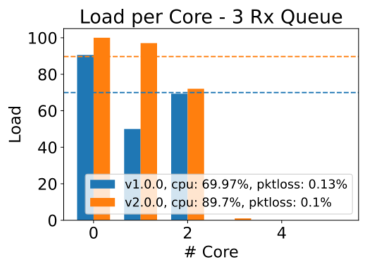
(b) Uplink 1 cores



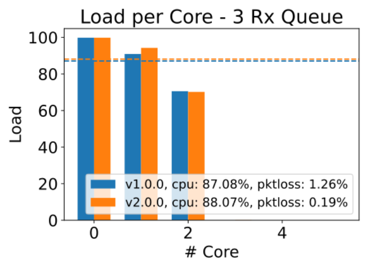
(c) Downlink 2 cores



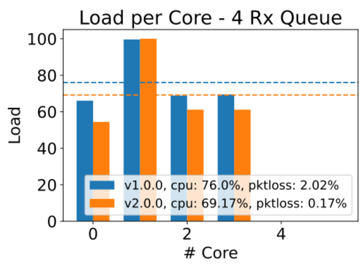
(d) Uplink 2 cores



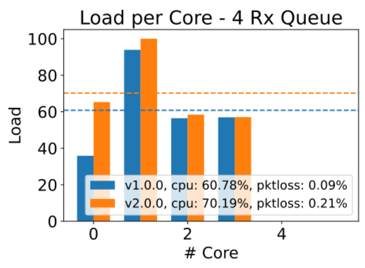
(e) Downlink 3 cores



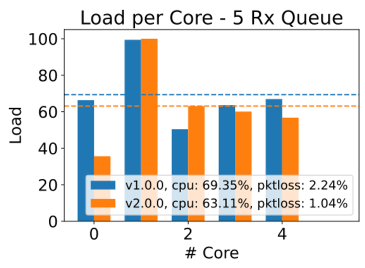
(f) Uplink 3 cores



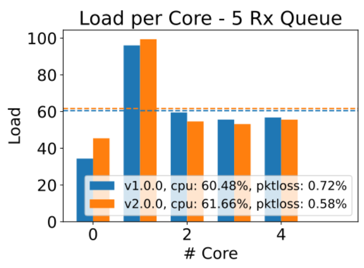
(g) Downlink 4 cores



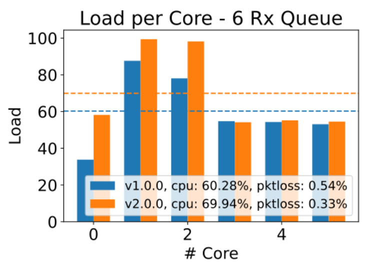
(h) Uplink 4 cores



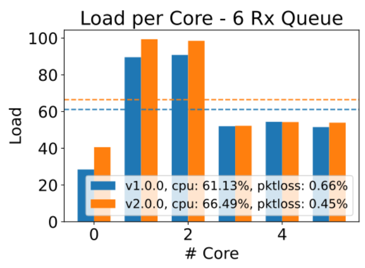
(i) Downlink 5 cores



(j) Uplink 5 cores



(k) Downlink 6 cores



(l) Uplink 6 cores

Figure 6.4 – Workload distribution when varying the number of cores.

The time spent to inject the BPF program for each version is shown in Table 6.1 after receiving a PFCP Establishment Request message composed of two PDRs and two FARs. All rules are deployed into one BPF section for the version 1 (1.0.0) and only the FAR associated with the highest precedence PDR is deployed into one BPF section for the version 2 (2.0.0) (see the explanation in Section 5.2.2). For version 1, it costs 27 ms to inject the BPF program. On the other hand, for version 2, only 1 ms (time reduction of 96%). Besides, the number of instructions decreased by 32%. The main reason is due the logic related to lookup the PDR is implemented in the control plane (Management Layer). The results follow the correlation between the number of instructions and the time to inject the BPF program into the kernel presented in (Miano et al., 2018).

Version	**BPF Section**	**BPF Insn**	**Injection (ms)**
v1.0.0	PFCP Session	402	27
v2.0.0	FAR	272	1

Table 6.1 – Time spent to inject BPF program into the Linux kernel (JIT compiler phase) after receiving a PFCP Establishment Request message.

7 Conclusions

This work addressed the run-time adaptive BPF/XDP solution for 5G UPF, a novel open source C++ library used to improve the performance of the user plane in third-party UPF components. The solution fits in a restrictive environment like NPN or MEC, when MEC host and UPF are collocated with the Base Station, sharing the same computational and network resources. We showed that the two versions of the `upf-bpf` design scale and achieve high throughput without consuming all the CPU resources. Besides, the last version showed to be more adaptive due to the lower time spent to inject the BPF program into the kernel. We are confident that the new design can be a reference point for those who want to create BPF-based network function implementation. In the next section, we revisit the research questions presented in Section 4.1 and the research direction for future work.

Research Questions Revisited

The **RQ1** asks if it is possible to create a solution of 5G UPF using BPF/XDP technologies and how would be the design in order to create a proof of concept following the 3GPP specification Release 16. Chapter 5 defined the minimum requirements in order to create a 5G UPF prototype. The routing and forwarding user plane features were selected to be the main features to make a proof of concept and two versions of the design were presented. In particular, the second design presented a fine-grained flexible solution to address the limitations of the BPF technology that was not considered in the first design. Each rule of the PFCP session represents a BPF program which is loaded in run-time when a new PFCP session establishes a request is sent to the UPF. The source code of each version is available in the GitHub upf-bpf repository.

The **RQ2** asks what levels of performance can be expected for the proposed solution. Chapter 6 presented the performance evaluation for the two versions. The results indicate that the solution scales for both versions and has similar results of other research based on BPF/XDP (Mobile Gateway using Polycube framework). Besides, the time to inject the BPF program in the last version indicates that the solution can respond faster when there is a change in the network requirements.

Comparison against the state-of-the-art

Comparing our work to the closest related work by (Parola, F., 2020), we can highlight the main differences from design and implementation perspectives as well as

	Parola	**This work**
	Design and Implementation	
Packet forward support	Yes (Implemented)	Yes (Implemented)
QoS support	Yes (implemented)	Yes (but not implemented)
BPF library	BCC	libbpf
Clang runtime dependency	Yes	No
Linux network processing layer	TC, XDP	XDP
Traffic debug support	Yes	No
Context based on PFCP (3GPP)	No	Yes
Open-source	polycube	upf-bpf
	Performance Evaluation	
DUT Machine	Intel Xeon Gold 5120 @2.60GHz processor	Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
DUT L3 Cache	19.25 MB	15 MB
DUT Memory DRAM	64 GB	32 GB
DUT NIC	Intel XL710 40Gbps	82599ES 10-Gigabit SFI/SFP+
DUT Hyper-threading	Disabled	Disabled
Packet Generator	MoonGen	Trex Traffic Generator
Multiple cores scalability	Yes	Yes
Packet size uplink	112 B	64 B
Packet size downlink	64 B	64 B
Performance uplink 6 cores	10 Mpps	10.9 Mpps
Performance downlink 6 cores	6.5 Mpps	9.9 Mpps
CPU load uplink 6 cores	Not available	69.9 %
CPU load downlink 6 cores	Not available	66.5 %

Table 7.1 – Comparison summary of (Parola, F., 2020) and this work.

the performance evaluation. Table 7.1 summarizes the comparison. The main differences are that Parola work is based on BCC while our work is based on libbpf. As shown in Chapter 2, the BCC depends on clang and kernel headers in run time. Besides, Parola's work supports QoS and it is still a work in progress. Our work was created from scratch, based on 3GPP specification, with the objective to implement the PFCP contexts described in (ESTI, 2020c) (see Section 4.2.1). On the other hand, Parola's work was developed inside polycube and has features like traffic debugging, for instance, that is not available in our work. Regarding performance evaluation, both solutions present similar results for uplink traffic, but our work used a smaller packet size (64 B against 112 B). Besides, there is a difference of around 3 Mpps for downlink compared to Parola's work for downlink traffic using the same packet size (64 B). Our work also measured the CPU usage for each throughput, which is an important indicator to know how optimized the packet processing is.

Future Work

As future work, we have identified a series of tasks regarding improving the functionality and portability. Also, integration with an 5G open source solution (e.g. OpenAirInterface) would be good a proof of concept to show the library working with a well-known 5G implementation. In the following, we describe briefly the foreseen activities.

QoS Enforcement Rule

In order to extend what has been proposed in the version v2.0 (Section 5.2.2), the QER functionality could be the next rule to be implemented, focusing in QoS control, i.e. maximum bit rate (MBR), guarantee bit rate (GBR) or Packet Rate enforcement according to (ESTI, 2020c). The task will be composed of changing the layers:

- **Management Layer:** parse the QERs when a new PFCP session is created. A new BPF program, which contains the QER functionality, will be loaded along with the FAR BPF program. The pipeline must be configured in order to include the new BPF program before FAR execution. A new class QERProgram that shall be implemented to abstract the BPF program lifecycle. It will also contain the BPF maps, which will contain information about the QER. Apart from the PFPC session creation, the update/delete session shall also be considered for the tasks;
- **Datapath Layer:** Figure 5.9 represents the logic of the datapath pipeline. The QER BPF program will be implemented to support the QoS control functionality described in (ESTI, 2020c). Fixed window counter, slide window or token bucket algorithms might be used. The program will be triggered through tail calls after the packet header is matched to a PDI of some PDR from a specific PFCP session. After execution, the QER BPF program will trigger the FAR BPF program through tail calls.

Regarding the test case, an important approach is to know how much latency is introduced when the QER BPF program is added into the pipeline. Besides, a test case to test the bit rate functionality might also be considered.

OpenAirInterface Integration

This activity consists in integrating the *upf-bpf* library into the OpenAirInterface SPGWu components. The integration has been started and is available in the GitHub repository[1]. The goal is to leverage the packet processing performance in a well-known open-source 5G core. Figure 7.1 shows a high-level design of the integration.

A basic test was performed as shown in the iteration diagram in Figure 7.2. The goal is to create a session in the SPGWu using the control plane available in the component, send GTP packets and check if the packet was processed.

The setup are composed of the following components:

[1] https://github.com/navarrothiago/openair-cn-cups

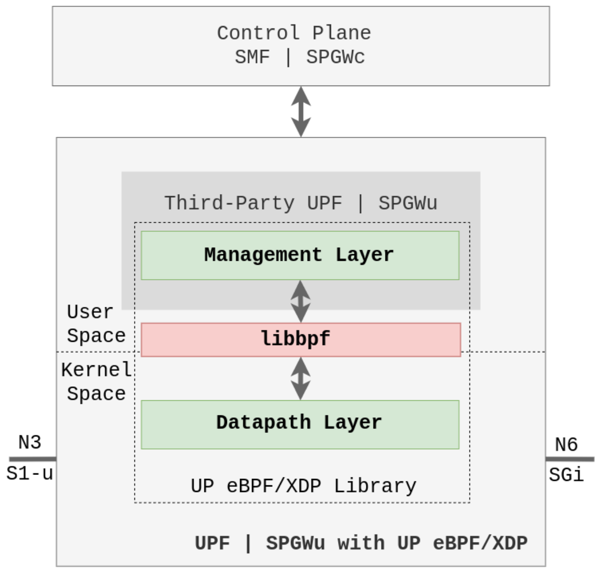


Figure 7.1 – High-leel architecture representing the integration between the *UPF BPF* library and the UPF/SGPWu component of the UP.

- **ITTI:** *InTerTask Interface* is the OAI *middleware* for exchanging messages between tasks (threads). All communication of the core components of the OAI network is done using this component;
- **Session Control Tester:** Component was developed for crafting and sending the PFCP session establishment request message to the SPGWu component via ITTI;
- **SPGWu:** *Gateway* of the OAI that implements the UP. Receives the PFCP session establishment request message and forwards it to the *upf-bpf* library;
- **UPF BPF:** The *upf-bpf* library proposed by this work. It handles the PFCP session establishment request and load the BPF program to handle the packets before the Linux network stack;
- **Packet Generator:** Python script that uses the *Scapy*[2] library to generate GTP packages of the G-PDU type. These packets are sent to the network.

There is a discussion available in the GitHub repository[3] that handles the next steps of this activity.

Technology Comparison

This activity consists in evaluating the performance between 5G UPF solutions based on high processing technologies like DPDK, SR-IOV and the `upf-bpf` (BPF/XDP).

2 <https://scapy.net/>
3 <https://github.com/navarrothiago/upf-bpf/discussions/52>

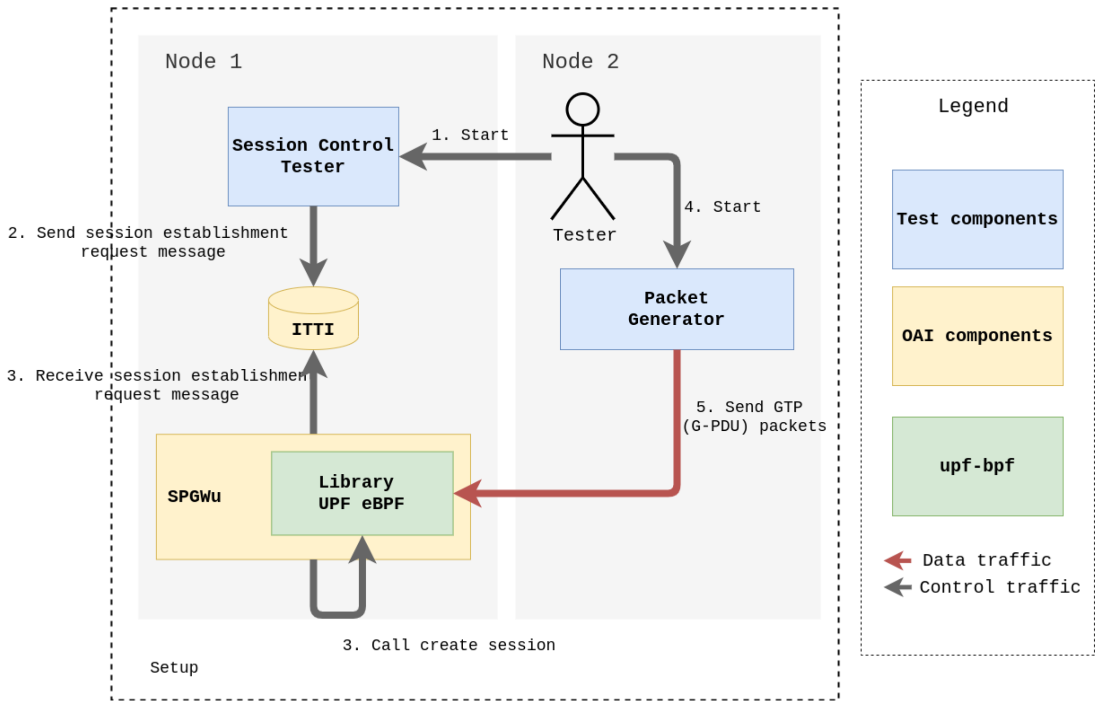


Figure 7.2 – Iteration diagram between OAI components and UPF BPF library. Caption: Yellow, OAI components; Blue, test components; Green, upf-bpf library inside SPGWu component.

Bibliography

BCC Community. *bpf compiler collection (bcc)*. 2022. Disponível em: <https://github.com/iovisor/bcc>. Citado na página 26.

BONATI, L. et al. Open, programmable, and virtualized 5g networks: State-of-the-art and the road ahead. *Computer Networks*, v. 182, p. 107516, 2020. ISSN 1389-1286. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128620311786>. Citado 2 vezes nas páginas 14 and 25.

Chen, W.; Liu, C. H. Performance enhancement of virtualized media gateway with dpdk for 5g multimedia communications. In: *2019 International Conference on Intelligent Computing and its Emerging Applications (ICEA)*. [S.l.: s.n.], 2019. p. 156–161. Citado 2 vezes nas páginas 25 and 27.

Cilium Community. *eBPF - Introduction, Tutorials Community Resources*. 2022. Disponível em: <https://ebpf.io>. Citado 3 vezes nas páginas 10, 17, and 18.

CISCO. *TRex - Realistic Traffic Generator*. 2022. Disponível em: <https://trex-tgn.cisco.com/>. Citado 2 vezes nas páginas 30 and 52.

CISCO. *TRex Traffic Generator Stateless API Documentation*. 2022. Disponível em: <https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/api/index.html>. Citado na página 41.

Cmake Community. *CMake*. 2022. Disponível em: <https://cmake.org/>. Citado na página 30.

ESTI. *5G; System architecture for the 5G System (5GS) (3GPP TS 23.501 version 16.6.0 Release 16)*. 2020. Disponível em: <https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf>. Citado 3 vezes nas páginas 21, 22, and 31.

ESTI. *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); General Packet Radio Service (GPRS); GPRS Tunnelling Protocol (GTP) across the Gn and Gp interface GPRS Tunnelling Protocol (GTP) across the Gn and Gp interface (3GPP TS 29.060 version 16.0.0 Release)*. 2020. Disponível em: <https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf>. Citado na página 23.

ESTI. *LTE; 5G; Interface between the Control Plane and the User Plane nodes (3GPP TS 29.244 version 16.5.0 Release 16)*. 2020. Disponível em: <https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf>. Citado 9 vezes nas páginas 22, 23, 31, 33, 35, 37, 42, 63, and 64.

ESTI. *Universal Mobile Telecommunications System (UMTS); LTE; Architecture enhancements for control and user plane separation of EPC nodes (3GPP TS 23.214 version 16.2.0 Release 16)*. 2020. Disponível em: <https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf>. Citado na página 22.

ETSI. *MEC Deployments in 4G and Evolution Towards 5G*. 2018. Disponível em: <https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp24_MEC_deployment_in_4G_5G_FINAL.pdf>. Citado 2 vezes nas páginas 10 and 15.

ETSI. *MEC in 5G networks*. 2018. Disponível em: <http://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp28_mec_in_5G_FINAL.pdf>. Citado 2 vezes nas páginas 14 and 15.

Facebook. *Magma - Facebook connectivity*. 2022. Disponível em: <https://connectivity.fb.com/magma/>. Citado 2 vezes nas páginas 26 and 27.

FEI, X. et al. Paving the way for nfv acceleration: A taxonomy, survey and future directions. In: . New York, NY, USA: Association for Computing Machinery, 2020. v. 53, n. 4. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3397022>. Citado na página 25.

Free5Gc Community. *free5Gc - Open-source project for 5th generation (5G) mobile core networks*. 2022. Disponível em: <https://www.free5gc.org/>. Citado 2 vezes nas páginas 26 and 27.

GIRONDI, M. *Efficient traffic monitoring in 5G Core Network*. Dissertação (Mestrado) — KTH Royal Institute of Technology, Stockholm, 2020. Citado 2 vezes nas páginas 10 and 24.

GSMA. *Understanding 5G: Perspectives on future technological advancements in mobile*. 2014. Disponível em: <https://www.gsma.com/futurenetworks/wp-content/uploads/2015/01/2014-12-08-c88a32b3c59a11944a9c4e544fee7770.pdf>. Citado na página 14.

GTP5G Community. *Kernel module for GTP protocol*. 2022. Disponível em: <https://github.com/PrinzOwO/gtp5g>. Citado na página 26.

HAN, S. et al. *SoftNIC: A Software NIC to Augment Hardware*. [S.l.], 2015. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>. Citado na página 26.

HØILAND-JØRGENSEN, T. 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*. New York, NY, USA: Association for Computing Machinery, 2018. (CoNEXT '18), p. 54–66. ISBN 9781450360807. Disponível em: <https://doi.org/10.1145/3281411.3281443>. Citado 3 vezes nas páginas 15, 16, and 25.

INTEL. *Building Enterprise-level Cloud Solutions with Outscale*. 2014. Disponível em: <https://www.intel.com/content/dam/www/public/us/en/documents/case-studies/xeon-e5-2660-family-ssd-s3700-series-dpdk-case-study.pdf>. Citado 2 vezes nas páginas 15 and 25.

INTEL. *Data Plane Development Kit (DPDK)*. 2022. Disponível em: <https://dpdk.org>. Citado na página 25.

KOHLER, E. et al. The click modular router. *ACM Trans. Comput. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 18, n. 3, p. 263–297, ago. 2000. ISSN 0734-2071. Disponível em: <https://doi.org/10.1145/354871.354874>. Citado na página 25.

LINUX. *Linux Kernel Network - Receive Side Scaling*. 2022. Disponível em: <https://github.com/torvalds/linux/blob/master/Documentation/networking/scaling.rst#rss-receive-side-scaling>. Citado na página 53.

Linux Community. *Libbpf Linux userspace library GitHub repository*. 2022. Disponível em: <https://github.com/libbpf/libbpf>. Citado na página 34.

LWN.net. *A thorough introduction to eBPF*. 2020. Disponível em: <https://lwn.net/Articles/740157>. Citado na página 17.

Miano, S. et al. Creating complex network services with ebpf: Experience and lessons learned. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. [S.l.: s.n.], 2018. p. 1–8. Citado 3 vezes nas páginas 19, 41, and 61.

NAKRYIKO, A. *BPF CO-RE (Compile Once - Run Everywhere*. 2020. Disponível em: <https://nakryiko.com/posts/bpf-portability-and-co-re/>. Citado na página 17.

OAI Community. *Openair-cn: Evolved Core Network Implementation of OpenAirInterface*. 2022. Disponível em: <https://github.com/OPENAIRINTERFACE/openair-cn>. Citado 3 vezes nas páginas 25, 26, and 27.

OMEC Community. *UPF EPC - 4G/5G Mobile Core User Plane*. 2022. Disponível em: <https://github.com/omec-project/upf-epc>. Citado 2 vezes nas páginas 26 and 27.

Open Source Guide. *Open Source Guides*. 2022. Disponível em: <http://opensource.guide/>. Citado na página 30.

Open5GS Community. *Open5Gs - Open source project of 5GC and EPC (Release-16)*. 2022. Disponível em: <https://github.com/open5gs/open5gs>. Citado 2 vezes nas páginas 26 and 27.

OpenStack. *Cloud Edge Computing: Beyond the Data Center - OpenStack Open Source Cloud Computing Software*. 2020. Disponível em: <https://www.openstack.org/use-cases/edge-computing/cloud-edge-computing-beyond-the-data-center>. Citado 2 vezes nas páginas 15 and 25.

PAPER, N. W. Network functions virtualisation: An introduction, benefits, enablers, challenges & call for action. issue 1. out. 2012. Citado na página 25.

Parola, F. *Prototyping an eBPF-based 5G Mobile Gateway*. Dissertação (Mestrado) — POLITECNICO DI TORINO, Italy, 2020. Citado 6 vezes nas páginas 11, 25, 27, 59, 62, and 63.

Parola, F. et. al. A proof-of-concept 5g mobile gateway with ebpf. In: *Proceedings of the ACM SIGCOMM 2020 Conference on Posters and Demos*. [S.l.]: Association for Computing Machinery, 2020. (SIGCOMM '20). Citado 4 vezes nas páginas 25, 26, 27, and 29.

Parola, F. et. al. Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway. In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. [S.l.: s.n.], 2021. p. 221–225. Citado 2 vezes nas páginas 25 and 27.

PFAFF, B. et al. The design and implementation of open vswitch. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. USA: USENIX Association, 2015. (NSDI'15), p. 117–130. ISBN 9781931971218. Citado na página 26.

Pinczel, B. et al. Towards high performance packet processing for 5g. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. [S.l.: s.n.], 2015. p. 67–73. Citado 2 vezes nas páginas 25 and 27.

Polycube Community. *About eBPF/XDP-based software framework for fast network services running in the Linux kernel*. 2022. Disponível em: <https://github.com/polycube-network/polycube>. Citado na página 26.

Ricart-Sanchez, R. et al. Hardware-accelerated firewall for 5g mobile networks. In: *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. [S.l.: s.n.], 2018. p. 446–447. Citado 2 vezes nas páginas 25 and 27.

Ricart-Sanchez, R. et al. Towards an fpga-accelerated programmable data path for edge-to-core communications in 5g networks. In: . [s.n.], 2018. v. 124, p. 80 – 93. ISSN 1084-8045. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1084804518302923>. Citado 2 vezes nas páginas 25 and 27.

RIZZO, L. Netmap: A novel framework for fast packet i/o. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USA: USENIX Association, 2012. (USENIX ATC'12), p. 9. Citado na página 25.

Ryu Community. *Ryu - SDN Framework*. 2022. Disponível em: <https://ryu-sdn.org/>. Citado na página 26.

SRS LTE Community. *srsLTE - Open-source 4G and 5G software radio suite developed by Software Radio Systems (SRS)*. 2022. Disponível em: <https://github.com/srsLTE/srsLTE>. Citado 2 vezes nas páginas 26 and 27.

SUSE. *SUSE - Introduction to eBPF and XDP*. 2022. Disponível em: <https://www2.slideshare.net/lcplcp1/introduction-to-ebpf-and-xdp>. Citado 2 vezes nas páginas 10 and 19.

The Linux Kernel documentation. *What are the verifier limits*. 2022. Available online: <https://www.kernel.org/doc/html/v5.2/bpf/bpf_design_QA.html#q-what-are-the-verifier-limits>. Accessed on 02 January 2022. Citado na página 36.

VIEIRA, M. et al. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, v. 53, p. 1–36, 02 2020. Citado 2 vezes nas páginas 18 and 19.

Appendix

APPENDIX A – Publication

T. A. N. do Amaral, R. V. Rosa, D. F. C. Moura, and C. E. Rothenberg, "An In-Kernel solution based on XDP for 5G UPF: design, prototype and performance evaluation." In *2021 1st International Workshop on Network Programmability (NetP 2021)*, Izmir, Turkey, Turkey, Oct. 2021.

T. A. N. do Amaral, R. V. Rosa, D. F. C. Moura, and C. E. Rothenberg, "Run-time Adaptive In-Kernel BPF/XDP Solution for 5G UPF." In *MDPI Electronics*, 2022.