Universidade Federal do Espírito Santo

Diego Rossi Mafioletti

A Place-as-you-go In-network Framework by Modular Decomposition for Flexible Embedding to Software/Hardware Co-design

Vitória-ES

2023

A Place-as-you-go In-network Framework by Modular Decomposition for Flexible Embedding to Software/Hardware Co-design

Diego Rossi Mafioletti

Tese de Doutorado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aprovada em 21 de Outubro de 2022.

Magnos Martinello

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
Vitória/ES, 21 de Outubro de 2022.

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

PROTOCOLO DE ASSINATURA

O documento acima foi assinado digitalmente com senha eletrônica através do Protocolo Web, conforme Portaria UFES nº 1.269 de 30/08/2018, por
MAGNOS MARTINELLO - SIAPE 1669875
Departamento de Informática - DI/CT
Em 21/10/2022 às 14:50

Para verificar as assinaturas e visualizar o documento original acesse o link:
https://api.lepisma.ufes.br/arquivos-assinados/588939?tipoArquivo=O

Ficha catalográfica disponibilizada pelo Sistema Integrado de
Bibliotecas - SIBI/UFES e elaborada pelo autor

Rossi Mafioletti, Diego, 1979-

R833p A place-as-you-go in-network framework by modular decomposition for flexible embedding to software/hardware co design / Diego Rossi Mafioletti. - 2022.

132 f. : il.

Orientador: Magnos Martinello.

Coorientador: Moises Renato Nunes Ribeiro.

Tese (Doutorado em Informática) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Redes de computadores. 2. Interconexão em rede (Telecomunicações). 3. Linguagem de programação de domínio específico (Computadores). 4. Sistemas embarcados (Computadores). 5. Sistemas de computação virtual. I. Martinello, Magnos. II. Nunes Ribeiro, Moises Renato. III. Universidade Federal do Espírito Santo. Centro Tecnológico. IV. Título.

CDU: 004

Dedico este trabalho à minha família e amigos.

Resumo

A computação em nuvem tornou-se muito popular como plataforma computacional, fazendo parte do cotidiano das pessoas. Para fornecer o poder de supercomputação exigido pela nuvem, a rede deve desempenhar um papel crucial ao conectar centenas de milhares de máquinas nos *data centers*. No entanto, com o surgimento de um novo conjunto maciço de aplicativos intensivos e nativos da nuvem (por exemplo, 5G, robótica em nuvem, aprendizado profundo, etc.) combinado com a virtualização de funções de rede (NFV), uma pressão significativa foi colocada sobre a capacidade de processamento das CPUs dos servidores, exigindo ainda mais alta performance e transformando a rede em um gargalo.

Normalmente, a interface de rede (NIC) é usada para conectar servidores à rede. No entanto, as interface de rede inteligentes (SmartNICs) estão se tornando um método cada vez mais popular de descarregar tarefas intensivas de processamento de pacotes dos servidores, liberando ciclos da CPU e impulsionando o desempenho de aplicativos.

O desafio chave para o uso de uma SmartNIC é como fazer uso eficiente desses recursos de computação heterogêneos na rede, pois há uma lacuna significativa entre o aplicativo e os recursos de computação dos dispositivos programáveis. Em primeiro lugar, este dispositivo carece de modelos genéricos de programação ou abstrações, sendo geralmente programado usando primitivas de baixo nível ou APIs proprietárias. Em segundo lugar, o desenvolvedor de rede precisa lidar com a complexidade interna dos recursos de hardware, bem como gerenciar o equilíbrio nas cargas de trabalho de descarregamento, tentando descobrir o balanço entre sobrecargas adicionais e benefícios do descarregamento. É necessário descobrir como co-projetar a lógica do aplicativo entre o hardware de rede programável e os servidores dentro do paradigma de computação de borda.

Esta tese apresenta um novo arcabouço para prototipagem e implantação de aplicativos em rede. Ele é estruturado em um conjunto de componentes que dependem de i) decomposição funcional da aplicação; ii) identificação dos blocos lógicos; iii) agregação de funções sobrepostas mescladas e mapeadas em funcionalidades de rede em linguagem P4; iv) interceptar, interagir e encaminhar com estruturas de dados para balancear o descarregamento de fluxos de rede.

A fim de demonstrar o princípio de co-design em várias aplicações, funções de rede virtual (VNFs) são criadas, e alguns de seus elementos funcionalmente decompostos são implantados como pequenas funções de rede incorporadas (eNFs) em processadores de rede em diversos casos de uso, revisando os componentes do arcabouço levantados anteriormente: decompondo funções de rede (i, ii, iii) e unificando componentes para se encaixarem em um conjunto de VNFs/eNFs, examinando latência, taxa de transferência e uso de vCPU em relação à sua contraparte implementada em software; interceptando e interagindo com robótica em nuvem (iv), abordando preocupações de segurança ao usar planos de dados programáveis; evoluindo para funções de rede seguras executadas em computação em rede (i, ii, iii, iv), verificando o overhead adicionado por elas; e modificando um mecanismo de escalonador *upstream* em PON (iv), fornecendo requisitos de baixa latência a aplicativos.

Palavras-chave: Computação em Rede, Co-design de hardware/software, Virtualização de Funções de Rede, Computação de Borda, Paradigma 5G.

Abstract

Cloud computing has become very popular as a computation platform, being part of people's daily life. In order to deliver the cloud-required super-computing power, the network must play a crucial role by connecting hundreds of thousands of machines within data centres. However, with the rise of a new massive set of cloud-native and intensive applications (e.g. 5G, cloud robotics, deep learning, etc) combined with network function virtualisation (NFV), a significant strain has been placed on the processing capacity of server CPUs, demanding even higher performance and turning the network into a bottleneck.

Typically, the network interface card (NIC) has been used to connect servers to the network. Although, smart network interface cards (SmartNICs) are becoming an increasingly popular method of offloading intensive packet processing tasks from servers, thus freeing CPU cycles to drive application performance.

The critical challenge towards using a SmartNIC is how to make efficient use of these in-network heterogeneous computing resources, as there is a significant gap between application software and the computing capabilities of programmable devices. First, this device lacks generic programming models or abstractions, being usually programmed using low-level primitives or proprietary APIs. Second, the network developer needs to deal with the internal complexity of hardware resources, as well as manage the balance on offloading workloads, trying to find out the trade-off between additional overheads and offloading benefits. It is needed to figure out how to co-design application logic between programmable network hardware and end-host servers within the edge computing paradigm.

This thesis presents a novel framework for prototyping and deploying in-network applications. The framework is structured into a set of components that rely on i) application functional decomposition; ii) identification of logic blocks; iii) aggregation of overlapped functions merged and mapped on network functionalities in P4 language; iv) intercepting, interacting and forwarding with data structures for balancing the offloading of network flows.

In order to demonstrate the principle of co-designing on diverse applications, Virtual Network Functions (VNFs) are created, and some of their functionally decomposed elements are deployed as small embedded Network Functions (eNFs) on in-network processors in sorted use cases, reviewing the framework components raised previously: decomposing network functions (i, ii, iii) and unifying components to fit into a set of VNFs/eNFs, examining latency, throughput and vCPU usage against their software implementation counterpart; intercepting and interacting with cloud robotics (iv), raising security concerns when using programmable data planes; evolving to security network functions running at in-network computing (i, ii, iii, iv), checking the overhead added by them; and modifying a PON upstream scheduler mechanism (iv), providing low latency requirements for applications.

Keywords: In-Network Computing, Hardware/Software Co-design, Network Functions Virtualisation, Edge computing, 5G paradigm.

List of Figures

1.1 Architectural options to In-Network Computing for packets processing. 11
1.2 Correlation between the main topics and publications of the thesis with the intersection between them. 15

2.1 Edge computing paradigm. Source [Shi and Dustdar 2016]. 19
2.2 A typical PON architecture. 20
2.3 PON DBA abstraction [ITU-T 2014]. 21
2.4 CU/DU-based C-RAN architecture [Ahmadi 2019]. 21
2.5 NFV implementation of network functions using virtualisation techniques over standard hardware. Source: [Han et al. 2015]. 24
2.6 SmartNIC architecture simplified block diagram. 26
2.7 Traditional switches vs. programmable switches [The P4 Language Consortium 2021]. 27
2.8 Portable Switch Pipeline [The P4.org Architecture Working Group 2020]. 27
2.9 Packet paths in PSA [The P4.org Architecture Working Group 2020]. . 28
2.10 Match-Action Unit Dataflow. [The P4 Language Consortium 2021]. . . 28
2.11 Programming a target with P4 [The P4 Language Consortium 2021]. . 29
2.12 The architecture of ClickNP [Kun et al. 2016]. 30
2.13 The SmartNIC GFT architecture, showing the flow of exception packets from the FPGA to software to establish a flow offloaded in hardware [Firestone et al. 2018]. 32
2.14 FLOEM system architecture. [Phothilimthana et al. 2018]. 33
2.15 λ-NIC's abstract machine model. [Choi et al. 2019] 34
2.16 NICA overview. [Eran et al. 2019]. 35
2.17 An overview of the iPipe runtime on the programmable NIC [Liu et al. 2019]. 36
2.18 FlowBlaze machine model. [Pontarelli et al. 2019]. 37
2.19 FairNIC sharing resources between two tenants. [Grant et al. 2020]. . 38
2.20 Programming a target with P4 [The P4.org API Working Group 2020]. . 39
2.21 Overview of the EP4 architecture [Karrakchou et al. 2021]. 40
2.22 The DPPx system architecture [Osinski et al. 2019] 41

3.1 PIaFFE framework architecture. 47
3.2 Illustrative example of functional decomposition using a set of network applications. 49
3.3 Multi-level forwarding using PIaFFE Intercepting and Forwarding (PIIF) element. 50
3.4 Architectural options for PIaFFE. 52

3.5 Netronome P4 SDK architecture. 53
3.6 PIaFFE framework building blocks. 53
3.7 Resumed steps for embedding applications into a programmable network device using the PIaFFE framework. 56

4.1 Network layers handled by PIaFFE in this Chapter. 60
4.2 Functional decomposition of monolithic VNFs, inspired by [Rahman Chowdhury et al. 2019]. 61
4.3 Modular eNFs and P4 logic mapping for network functionalities. 61
4.4 Multi-level Chaining of Network Functions. 62
4.5 PIaFFE outlined onto ETSI NFV reference architecture framework [Canada et al. 2017] 64
4.6 Prototype implementation and testbed. 66
4.7 Latency comparison for different scenario types. 67
4.8 Throughput in Millions of Packets per Second (Mpps). 68
4.9 Throughput in Megabits per second (Mbps). 68
4.10 VNF CPU usage for two extreme packet sizes. 69

5.1 Cloud Robotics and Programmable Data Plane. 72
5.2 Network layers handled by PIaFFE in this Chapter. 73
5.3 PIaFFE and multi-level chaining concept. 73
5.4 Network packet headers, ROS message header and ROS data. 74
5.5 ROS operation: the master registers and provides information for publishers and subscribers to establish connections and allow nodes to exchange information directly via topics. 76
5.6 Threat model: hijacking. 78
5.7 TurtleSim robot simulator: (1) Normal robot behaviour; (2) Setting a trigger via P4 table; (3) Abnormal robot behaviour (red track) 79
5.8 Threat model: replay attack. 80
5.9 Replay attack: a) robot position on plane varying with time, normal (blue) versus disturbed (red) operation 81
5.10 Replay attack: angular velocity of the robot under normal operation (blue zone) and under attack (red zone); the red dashed line marks the beginning of the attack. 82
5.11 Replay attack: linear velocity of the robot under normal operation (blue zone) and under attack (red zone); the red dashed line marks the beginning of the attack. 82

6.1 Network layers handled by PIaFFE in this Chapter. 86
6.2 Illustrative example of decomposition using a set of network applications. 87
6.3 Multi-level forwarding using Intercepting and Forwarding (PIIF) element. 88
6.4 Virtualisation techniques: Open vSwitch software switch 91
6.5 Virtualisation techniques: Intel DPDK L2 forwarder application 91
6.6 Virtualisation techniques: P4-enabled hardware 92
6.7 Testbed specification: Prototype implementation and testbed. 92
6.8 Using hardware timestamp to measure the time to process/deliver a packet. 93

6.9 Latency (Processing and forwarding) for software and hardware micro-applications running SHA-256 algorithm and network forwarding: Total latency using different types of forwarding/calculation virtualisation techniques. 95
6.10 Cumulative Distribution Function (CDF) Probability for SHA-256 calculation and forwarding. 95
6.11 Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: Open vSwitch (OVS) software switch. 96
6.12 Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: DPDK application. 96
6.13 Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: embedded network function with P4 in NIC. 97

7.1 Network layers handled by PIaFFE in this Chapter. 101
7.2 Fast Intercept concept. 103
7.3 Summary of steps involved in the traditional DBA process. 105
7.4 Proposed dual-DBA process with P4 in-NIC computing. 106
7.5 Multi-level DBA scheduling provided by In-Network Computing. 107
7.6 Fast Intercept rescheduling mechanism. 108
7.7 P4 Parser and Ingress pipelines 108
7.8 Comparison between multiple approaches to calculating the DBA algorithm. 109
7.9 Dissecting the algorithm: times of each step. 110

List of Tables

2.1 PIaFFE compared with other framework projects. 43

List of Codes

3.1 P4 source code of the PIIF 51
4.1 P4 source code example of an Offload Balancer 63
6.1 Pseudo code of the PIIF 88
6.2 A fragment of micro-C extern SHA-256 function. 89

List of abbreviations and acronyms

ASIC	Application-Specific Integrated Circuit
CDN	Content Delivery Network
DBA	Dynamic Bandwidth Allocation
DC	Data Centre
eNF	Embedded Network Function
IaaS	Infrastructure as a Service
INC	In-Network Computing
MEC	Multi-Access Edge Computing
National Institute of Standards and Technology	NIST
NF	Network Function
NFV	Network Function Virtualisation
NIC	Network Interface Card
PaaS	Platform as a Service
PIIF	PIaFFE Intercepting and Forwarding
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Network
SaaS	Software as a Service
SFC	Service Function Chain
SHA	Secure Hash Algorithm
SR-IOV	Single Root I/O Virtualisation
VNF	Virtual Network Functions

Contents

1 Introduction **10**
1.1 Research Question 12
1.2 Hypotheses 13
1.3 Objectives 14
1.4 Contributions 14
1.5 Text Structure 16

2 Foundations and State-of-the-Art **18**
2.1 Background 18
2.1.1 Cloud and edge computing 18
2.1.2 Network Function Virtualisation 23
2.1.3 Programmable network hardware 25
2.2 Related works 30
2.2.1 ClickNP 30
2.2.2 Azure Accelerated Networking (AccelNet) 31
2.2.3 FLOEM 33
2.2.4 Lambda-NIC 34
2.2.5 NICA 34
2.2.6 iPipe 35
2.2.7 FlowBlaze 36
2.2.8 FairNIC 37
2.2.9 P4Runtime 38
2.2.10EP4 39
2.2.11DPPx 40
2.3 PIaFFE framework 42
2.4 Chapter remarks 44

3 A framework for flexible software/hardware co-design of in-network computing **45**
3.1 Overview 45
3.2 Proposal 46
3.2.1 Design principles 46
3.3 Architecture design and enablers 51
3.3.1 Network programmability levels 54
3.4 PIaFFE automated deployment 55
3.5 Chapter remarks 57

4 A co-design use-case on Network Function Virtualisation (NFV) 58

4.1 Overview . . . 58

4.2 NFV use-case . . . 59

4.2.1 Experimentation design . . . 59

4.2.2 Usage of the framework . . . 63

4.3 Experimental evaluation . . . 64

4.3.1 Experimentation setup . . . 65

4.3.2 Testbed specification . . . 65

4.3.3 Latency results . . . 66

4.3.4 Throughput results . . . 67

4.3.5 vCPU usage . . . 69

4.4 Chapter remarks . . . 70

5 Hacking programmable data planes in cloud robotics: intercepting and modifying ROS messaging 71

5.1 Overview . . . 71

5.2 ROS use-case . . . 72

5.2.1 Experimentation design . . . 72

5.2.2 Usage of the framework . . . 74

5.2.3 Background review . . . 74

5.2.4 Vulnerabilities in programmable network devices . . . 75

5.3 Threat models exposed . . . 77

5.3.1 In-network hijacking: man-in-the-middle attack . . . 77

5.3.2 In-network replay attack . . . 80

5.4 Chapter remarks . . . 83

6 Providing data integrity by interacting and modifying raw packets: an SHA use-case 84

6.1 Overview . . . 84

6.2 SHA use-case . . . 85

6.2.1 Experimentation design . . . 85

6.2.2 Usage of the framework . . . 86

6.2.3 Functional decomposition for deployment . . . 86

6.2.4 Multi-level forwarding between hardware and software . . . 88

6.2.5 Hash algorithm implementation . . . 89

6.3 Experimental evaluation . . . 90

6.3.1 Benchmarking the virtualisation techniques . . . 90

6.3.2 Experimentation setup . . . 92

6.3.3 Results . . . 94

6.4 Chapter remarks . . . 99

7 Low-latency passive optical networks (PON) through intercepting and modifying DBA frames 100

7.1 Overview . . . 100

7.2 PON use-case . . . 101

7.2.1 Experimentation design . . . 101

7.2.2 Usage of the framework . . . 102

7.2.3 Implementation details . . . 107
7.3 Experimental evaluation . . . 109
7.3.1 Experimental results . . . 109
7.4 Chapter remarks . . . 111

8 Conclusion and Future Works **112**
8.1 Thesis Summary . . . 112
8.2 Future Works . . . 113
8.3 Final Remarks . . . 114

Bibliography **116**

Chapter 1

Introduction

Recent years have seen a rapid increase in network interface bandwidth for data centre (DC) servers [DC 2015, Cloud 2016], outpacing the CPU computing power. For example, since 2009, Azure [Firestone 2017] has seen a 50x improvement in the network bandwidth in contrast with modest increases in CPU performance. Consequently, data-centre operators have to burn more CPU cores to fully utilise the network bandwidth, leaving fewer computing resources for tenant workload execution, increasing the cost of running cloud services, and adding latency and variability to network performance.

Edge computing has the potential to alleviate these constraints [Panicucci et al. 2020], aiming to bring cloud resources and services at the border of the network, as an intermediate layer between the end users and cloud data centres. By approximating the user to the cloud, network-related issues (e.g. packet loss, security and high latency) can be fixed for new applications, such as cloud robotics [Mello et al. 2019, Xie et al. 2019] and passive optical networks (PON) [Das et al. 2020].

Nevertheless, edge computing is an expensive architecture solution when compared to the cloud and may not be able to tackle a large set of latency-sensitive and critical applications in production environments. Thus, there is room for exploiting this demand for computational resources on the edge, by using state-of-the-art data plane programmability for co-design applications and hardware, reducing host CPU demand and improving latency requirements.

In-Network Computing (INC) is a promising field that aims at using the capabilities of programmable network devices, such as programmable switch ASICs and programmable network interface cards (aka SmartNICs), offloading computing from commodity servers to the programmable data plane [Rüth et al. 2018, Nour et al. 2020].

Emerging in-networking processor-based smart network interface cards (SmartNICs) (e.g., Netronome Agilio [NETRONOME 2019], Cavium LiquidIO [CAVIUM 2019], Mellanox BlueField [MELLANOX 2019]) offer a solution to offload the network traffic from traditional commodity servers (x86-64), providing a software development kit (SDK) for customising the data plane by use of domain-specific programming languages (e.g., P4, eBPF, etc.), expressing how packets should be processed into these programmable devices.

Figure 1.1 shows the architecture options for INC, highlighting the three main architectural targets when prototyping (in-)network applications. In terms of forwarding performance and computational resources, the main characteristics are the following:

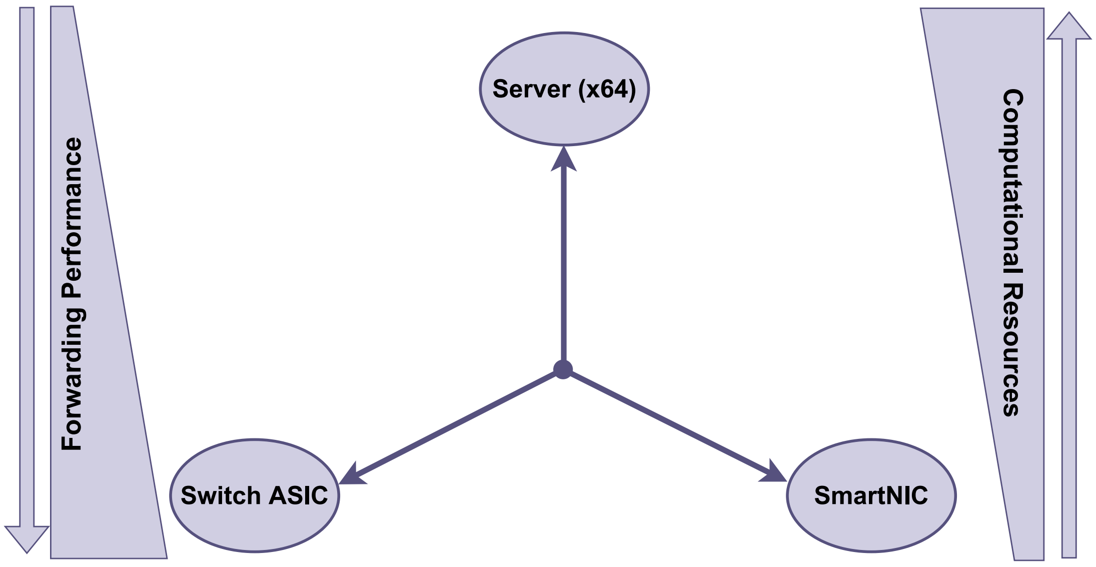


Figure 1.1: Architectural options to In-Network Computing for packets processing.

i. **Traditional commodity servers**, with plenty of CPU and memory resources for processing network packets, but with restricted packet forwarding performance;

ii. **Programmable switch ASIC**, with multiple network physical ports and high-performance for forwarding packets, but with limited system resources, i.e., memory, CPU;

iii. **SmartNIC (or FPGA-based) programmable device**, placed close to the commodity servers, which can boost up the offloading due to the shortest path to the *software layer*, however with the same limited computational resources as presented in programmable switch ASIC.

Traditionally, these three main architectures can be combined in order to achieve the aimed performance by offloading specific network function operations to meet the requirements of a large variety of applications. For example, in the Network Function Virtualisation (NFV) paradigm, a direct benefit of employing a SmartNIC-based solution for edge computing is its physical placement in relation to the commodity servers, shorting the pathway between virtual network functions (VNF) implemented in software and their potential pairs implemented into the hardware data plane.

On that account, these INC capabilities allow commodity servers to offload limited amounts of simple but general computations onto the programmable NIC while keeping the support of complex application logic on the hosts. By offloading lightweight and frequently invoked operations from the host to its own programmable network device, we can accelerate cloud applications on edge, while reducing the host CPU load without sacrificing the program generality.

However, this intuitive scenario is not trivial to be implemented considering that the following points should be addressed:

- a compact eNF (embedded Network Function) version from a traditional VNF (Virtual Network Function) or application using a domain-specific programming language that fits into a resource-limited programmable NIC;
- a proper mechanism for exchanging network traffic between software and hardware levels;
- the correct balance of network traffic that can be offloaded without degrading the system's performance.

Therefore, the first key issue to be addressed is how to fit a conventional VNF inside a resource-limited programmable network device. In this context, **functional decomposition** is the process of taking a complex process and breaking it down into its smaller, simpler parts [Rahman Chowdhury et al. 2019]. Consequently, a modular functional decomposition can be used to dissect a VNF, enabling the reconstruction of it as an eNF employing the P4 language.

The next issue to be handled should be the communication between eNF and its VNF counterpart, as each is running in a different environment separated by PCI-e bus and system layers. As the network traffic data in the edge mainly flows from the NIC to the host itself (in contrast with DC networks, where VM traffic flows inside the host should be predominant), the rational choice is to create an **interception mechanism** inside the SmartNIC, which will be able to steer traffic between hardware and software levels, while following the application requirements.

Furthermore, merging the capabilities of software and hardware is another stone in the design and implementation process, as they have different constraints and resources. Hence, the **co-design** technique aims to incorporate hardware/software and exploit the synergy between the two, satisfying system load constraints, i.e., the correct proportion of offloading without overwhelming the overall system, including host CPU and SmartNIC hardware.

1.1 Research Question

To tackle the aforementioned problems, this thesis aims to answer the following question:

- How to exploit co-design, i.e., between programmable network devices and end-host processing, opened by SmartNICs to accommodate diverse logic and their requirements considering different applications, including malicious ones, within the edge computing paradigm?

The co-design of applications presents an enormous challenge, as well as an opportunity, for network developers. By prototyping software-based network applications regarding the possibility of acceleration by offloading tasks to the data plane, developers can further explore the hardware capabilities in coexistence with their own applications, considering an offloading based on application/hardware requirements/constraints, leveraging edge computing by tackling concurrent resources within in-network computing.

Moreover, the co-design could be exploited (as we will reveal in cloud robotics security applications), creating a “tampered network device” which can compromise the network premises of a robot fleet by using malicious techniques for flooding or modifying the network traffic while flowing transparently through a customised data plane. Envisioning such a possibility, we can draw up a cryptographic network function embedded into the SmartNIC, ensuring data integrity by processing traffic data, and also saving CPU resources by offloading workload to the NIC.

1.2 Hypotheses

In this thesis, we foresee the following hypotheses to tackle the described problem:

- **Hypothesis H1**: The suitable modular functional decomposition (identification and aggregation) is a key principle in matching the processing capacity of in-network computing, provided by the P4 language, to the specialised logic of applications.

  - **Justification H1**: Since P4 language was not conceived with a modular functional decomposition method, porting oversized applications originally designed for the x86-64 architecture – which relies on vast computational resources (memory and CPU) – to a programmable network device can demand a lot of effort from the network developers. Functional decomposition urges simplifying this further migration or integration between applications and programmable data planes, easing the portability mainly by identifying the small functional parts of applications and giving a bird’s-eye view of the network functions.

- **Hypothesis H2**: The lack of an architectural component for intercepting and interacting with raw packets is preventing innovative applications for in-network computing, other than the regular offloading of network flows onto SmartNICs.

  - **Justification H2**: New complex and critical applications which can take advantage of edge computing, i.e. cloud robotics and passive optical networks, bring specific security and low latency requirements, as a minimum interference may disrupt the robot fleet operation or latency-sensitive applications. A programmable network device is a powerful tool, enabling the customisation of the packet processing/forwarding at a high-granularity level, but at the same time, raises concerns like: “what if that device has tampered”? A malicious attack could be delivered by a jeopardised device, compromising network security. By processing packets beyond the traditional network layers, or raw packet data, in-network computing can intercept/interact with the application layer, manoeuvring the end hosts according to its logic. At the same time, this characteristic also could be used for leveraging network applications and building embedded network functions that can be able to interact with the network traffic data without the modification of their software/protocol counterpart.

1.3 Objectives

To the best of our knowledge, we have not seen a state-of-art framework that covers this level of co-design and interaction between hardware and software applications using proper network programmable hardware and a domain-specific programming language. In order to tackle the defined research question and test the aforementioned hypothesis, this work has the following general objectives:

- To propose a framework that relies on modular functional decomposition (identification and aggregation) for flexible embedding network applications with their functionalities developed in P4 language and deployed into SmartNICs. The principle of co-designing will be demonstrated by an NFV use case.
- To introduce an architectural component in the framework for intercepting and/or interacting with raw packets, tackling security and performance aspects when using a programmable data plane.
- To demonstrate the framework by addressing diverse applications, such as cloud robotics applications, cryptographic hash functions and dynamic bandwidth allocation (DBA) in passive optical networks (PON).

1.4 Contributions

The proposal of this document was idealised through discussions and studies carried out in the LabNERDS-UFES and CONNECT Centre - Trinity College Dublin. Figure 1.2 gives an overview of how the publications are mapped to the different areas of interest in this thesis. The following contributions bring some clarification about the author's collaboration on each work, in reverse chronological order.

Publication A: D.R. Mafioletti, M. Martinello, M. R. N. Ribeiro, M. Ruffini, F. Slyne, "To Embed or Not to Embed SHA in Programmable Network Interface Cards", in 18th International Conference on Network and Service Management (CNSM), 2022.

Publication B: D.R. Mafioletti, F. Slyne, R. Giller, M. O'Hanlon, D. Coyle, B. Ryan, M. Ruffini, "Demonstration of a Low Latency Bandwidth Allocation Mechanism for Mission Critical Applications in Virtual PONs With P4 Programmable Hardware", in 2022 Optical Fiber Communications Conference and Exhibition (OFC), 2022, pp. 1-3.

Publication C: D.R. Mafioletti, R. C. de Mello, M. Ruffini, V. Frascolla, M. Martinello, M. R. N. Ribeiro, "Programmable Data Planes as the Next Frontier for Networked Robotics Security: A ROS Use Case", in 2021 1st Joint International Workshop on Network Programmability and Automation, 2021, doi: 10.23919/CNSM52442.2021.9615504.

Publication D: **D. R. Mafioletti**, F. Slyne, R. Giller, M. O'Hanlon, B. Ryan, and M. Ruffini, "A Novel low-latency DBA for Virtualised PON implemented through P4 In-Network Processing", in Optical Fiber Communication Conference (OFC) 2021, P. Dong, J. Kani, C. Xie, R. Casellas, C. Cole, and M. Li, eds., OSA Technical Digest (Optical Society of America, 2021), paper F4I.2, doi: 10.1364/OFC.2021.F4I.2.

Publication E: **D. R. Mafioletti**, C. K. Dominicini, M. Martinello, M. R. N. Ribeiro and R. d. S. Villaça, "PIaFFE: A Place-as-you-go In-network Framework for Flexible Embedding of VNFs", ICC 2020 - 2020 IEEE International Conference on Communications (ICC), 2020, pp. 1-6, doi: 10.1109/ICC40277.2020.9149240.

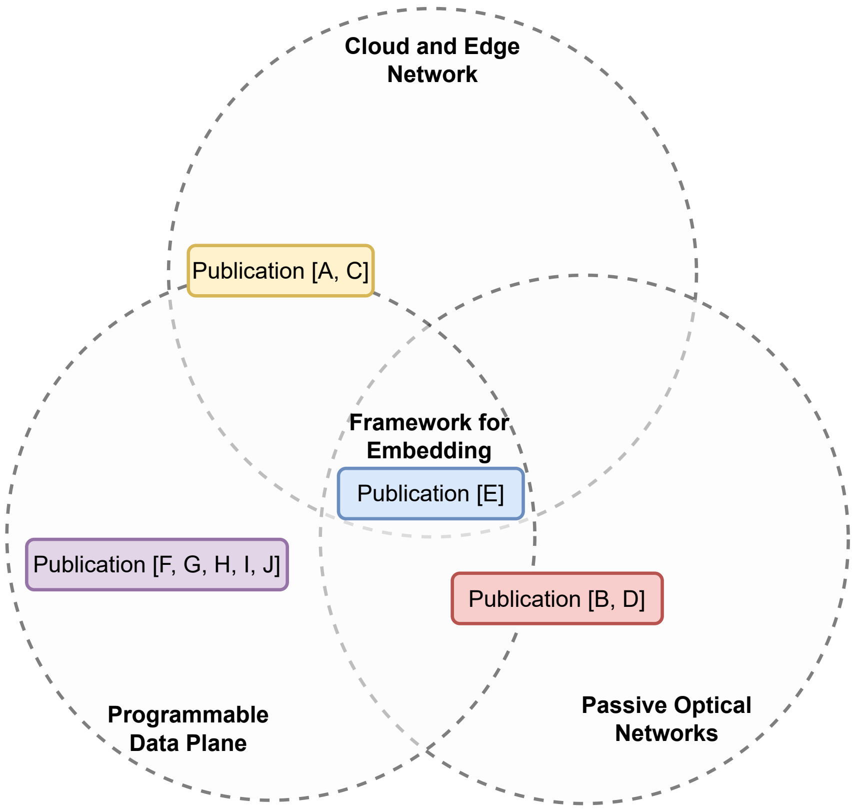


Figure 1.2: Correlation between the main topics and publications of the thesis with the intersection between them.

The following publications, listed in reverse chronological order, were developed during the PhD and are not directly linked to this proposal. However, they helped to gain important knowledge in the matter and, therefore, should also be considered relevant contributions.

Publication F: R. S. Guimaraes, C. Dominicini, V.M.G. Martínez, B. M. Xavier, **D. R. Mafioletti** et al., "M-PolKA: Multipath Polynomial Key-based Source Routing for Reliable Communications”, in IEEE Transactions on Network and Service Management, doi: 10.1109/TNSM.2022.3160875.

Publication G: C. Dominicini, R. S. Guimaraes, **D. R. Mafioletti** et al., “Deploying PolKA Source Routing in P4 Switches : (Invited Paper)”, 2021 International Conference on Optical Network Design and Modeling (ONDM), 2021, pp. 1-3, doi: 10.23919/ONDM51796.2021.9492363.

Publication H: R. S. Guimaraes, V. M. G. Martínez, R. C. Mello, **D. R. Mafioletti**, M. Martinello and M. R. N. Ribeiro, “An SDN-NFV Orchestration for Reliable and Low Latency Mobility in Off-the-Shelf WiFi”, ICC 2020 - 2020 IEEE International Conference on Communications (ICC), 2020, pp. 1-6, doi: 10.1109/ICC40277.2020.9148900.

Publication I: C. Dominicini, **D. R. Mafioletti** et al., “PolKA: Polynomial Key-based Architecture for Source Routing in Network Fabrics”, 2020 6th IEEE Conference on Network Softwarization (NetSoft), 2020, pp. 326-334, doi: 10.1109/NetSoft48620.2020.9165501.

Publication J: R. Valentim, C. K. Dominicini, R. S Villaça, M. Martinello, M. Ribeiro, D. R. Mafioletti, “RDNA Balance: Balanceamento de Carga por Isolamento de Fluxos Elefante em Data Centers com Roteamento na Origem”, in Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2019, pp. 1000-1013.

1.5 Text Structure

The remainder of this Doctoral Thesis is divided as follows:

Chapter 2 presents in more detail the comparison with related works. It covers crucial concepts related to In-Network Computing, Network Function Virtualisation, and cloud and edge computing, including Cloud Robotics and Passive Optical Networks, which will be explored during the Thesis.

Chapter 3 gives the building blocks to construct a framework for embedding applications into a programmable network card, positioning it into the state of the art of existing network designs and computer architecture for the development and deployment of applications.

Chapter 4 presents the design and implementation of our framework proposal to support the implementation of virtual network function on a programmable network interface card, offloading the network packet processing and sharing the resources between hardware and software levels using an offloading balancer.

Chapter 5 shows a different perspective of the proposal, now tackling a more specific use-case: a cloud/edge infrastructure allied with cloud robotics applications based on Robotic Operating System (ROS), depicting the security concerns over a programmable data plane and the implications on the local network infrastructure and discussing the possibilities to overhaul this weakness.

Chapter 6 shows the feasibility of using an embedded application constructed using the PIaFFE framework to run a secure hash algorithm onto a programmable network device, featuring new possibilities when offloading complex applications, in order to assure data security and reliability onto a limited-resources data plane.

Chapter 7 leads to migration from specific network functions (e.g. firewall, NAT, etc.) and applications (e.g. cloud robotics and cryptography) to more generic applications but with a specific purpose, such as a high-level dynamic bandwidth allocation (DBA) algorithm on Passive Optical Networks (PON), although keeping the essence of sharing resources between hardware and software. This chapter also outset the first attempt to deliver the control of the network latency to an embedded network function based on instructions made available inside the own network frame/packet.

Chapter 8 draws a conclusion for the Doctoral Thesis discussing the next steps and future works.

Chapter 2

Foundations and State-of-the-Art

This chapter describes the position of this thesis with related works in the area of cloud and edge domains and programmable network hardware, as well as towards more specific domains, such as applications running into the data plane, tackling security concerns on cloud robotics, as well as possible solutions using cryptographic functions, and also improving low-latency requirements for passive optical networks (PON).

Firstly, Subsection 2.1.1 and 2.1.2 position the capabilities of the cloud and edge domains and NFV, showing the limitations of the current scenario and exposing what we envision to address these limitations.

Furthermore, in 2.1.1.2 and 2.1.2.1 rely on specific application domains, starting by describing the cloud robotics network paradigm, showing the points that we can take advantage of when using a high-level abstraction for exploring the lack of security on a programmable data plane. In this same direction, we propose the offloading of cryptographic hash functions for helping to assure the security and reliability of data, demonstrating that even a small-resource device can afford such complex applications using the proposed techniques and methods.

Moreover, in 2.1.1.1 we expose the current limitations of actual architecture, advancing to the constraints of using programmable network hardware on Passive Optical Networks (PON), in order to improve the scheduling mechanism on upstream data, delivering low-latency traffic for the required application/service through co-processing PON frames inside the data plane.

To finish this chapter, in Subsection 2.1.3 we will discuss programmable network devices – which are one of the enablers of the proposal – and existing frameworks for programming and/or embedding the programmable data plane in Section 2.2, comparing them with the current proposal and showing the points on how the proposal differs from other existing works in 2.3.

2.1 Background

2.1.1 Cloud and edge computing

It is well known that cloud computing has been a promising architecture for next-generation networks once provides high flexibility resource virtualisation (e.g., servers with either CPU or GPU, networks, storage, and services) with cost-efficiency and centralised management [Le et al. 2016]. The cloud computing service models are categorised as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). They offer virtual resources (compute, storage, and network), software and development platforms (provided by the cloud infrastructure), and Internet-based applications (hosted on the cloud), respectively [Le et al. 2016].

As mentioned by [Nencioni et al. 2018], the possible future research challenges on softwarisation technologies such as NFV and SDN are the main enablers to develop novel solutions on top of new network generation (e.g., 5G network). The success of network virtualisation heavily relies on orchestrating end-controlling capabilities, which optimises the capital expenditure and operating costs (CAPEX and OPEX). In addition to the increasing softwarisation of infrastructure, content formats and production and consumption patterns are continuously changing as users demand improved Quality of Experience (QoE).

This trend requires adaptation processes that respond to aspects such as personalisation, localisation, interactivity, and mobility. Therefore, edge computing brings a new paradigm which shares computing, storage, and bandwidth resources as closely as possible with mobile devices or sensors. At the edge, the cloud services can deliver highly responsive time for local services [Satyanarayanan 2017]. The nodes at the edge DCs may perform many computing tasks, such as data processing, caching, service delivery, and privacy protection.

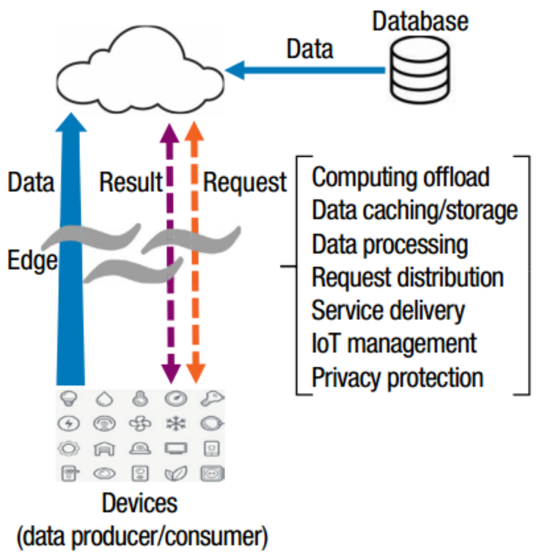


Figure 2.1: Edge computing paradigm. Source [Shi and Dustdar 2016].

ETSI defines the term Multi-Access Edge Computing (MEC) as a set of technologies that offers application developers and content providers cloud-computing capabilities and an IT service environment at the edge of the network, which extends the concept of edge cloud and can be used to assist C-RAN meeting the above 5G requirements. This environment is characterised by ultra-low latency and high bandwidth, as well as real-time access to radio network information that can be leveraged by applications [Sabella et al. 2019]. Those terms are viewed as necessary to enable specific use case classes defined, for instance, in 5G networks.

MEC brings highly efficient cloud computing and storage capabilities at the edge and can be used by a RAN to offer low-latency and high-bandwidth data processing for latency-critical applications. It can also offer content caching near end users in order to alleviate the overall network load on data transmission through caching and forwarding content at the edge of the network [Das and Ruffini 2020].

2.1.1.1 Cloud Radio Access Networks with Passive Optical Networks

Passive optical networks (PON) are considered one of the prominent access network solutions due to the high capacity and coverage they can provide using a point-to-multipoint scheme that enables a single optical fibre to serve multiple premises using passive components. A typical PON architecture is constituted of a centralised Optical Line Terminator (OLT) and a number of Optical Network Units (ONUs) located at the premises, as shown in Figure 2.2.

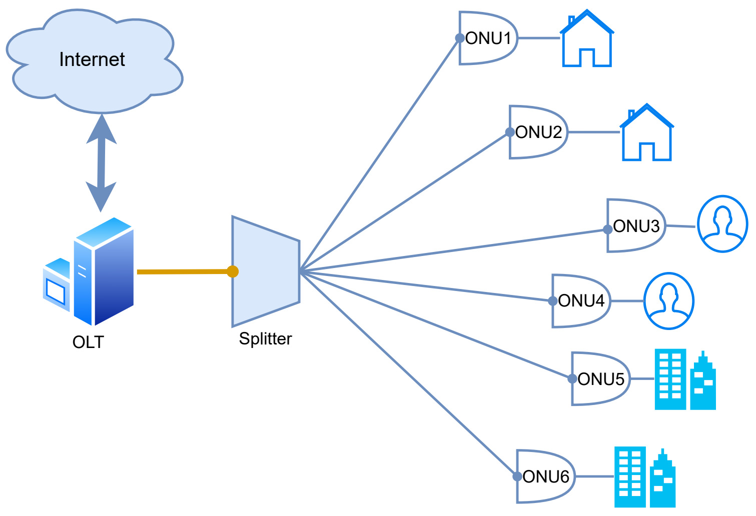


Figure 2.2: A typical PON architecture.

Meanwhile, the high Capital expenditure (CAPEX) required for PON deployment has been an obstacle to large-scale adoption, especially in rural areas with a lower number of users and bandwidth demand. To this point, multiple solutions have been proposed in the past to improve the business case of access fibre deployments, stemming from changes in the overall network architecture to the development of cost-effective transceivers for multi-wavelength PONs [Cheng et al. 2014].

A complementary approach to economic sustainability is to increase the revenue generated by the PON by increasing the number and types of services that can be supported, for example, including mobile back-haul [Alvarez et al. 2016] and front-haul [Chanclou et al. 2013], in addition to the enterprise and residential applications. Therefore, a scenario in which all the aforementioned services can coexist and operate on the same PON infrastructure is pivotal in increasing the utilisation of the infrastructure, thus generating new revenue streams.

PON uses a dynamic bandwidth allocation (DBA) mechanism (Figure 2.3) to prevent collisions between simultaneous upstream transmissions of multiple ONUs. DBA methods have surfaced with strategies to share limited resources in optical access networks with a growing number of users. Over the years, these networks have also vastly improved by increasing their bandwidth. In addition, Quality of Service (QoS) parameters are incorporated into the bandwidth-sharing procedure.

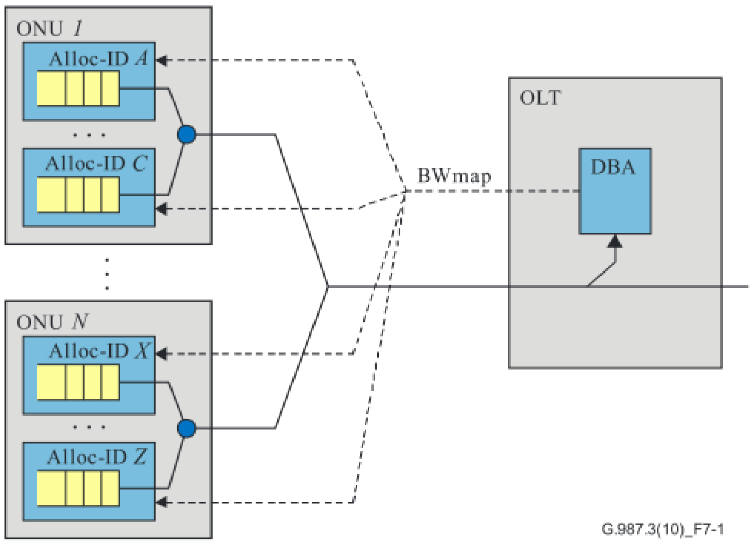


Figure 2.3: PON DBA abstraction [ITU-T 2014].

Cloud Radio Access Networks (C-RANs), along with functional split processing, are regarded as the most promising 5G radio technologies that support these requirements. In the 5G New Radio (5G-NR) architecture of C-RAN, the baseband processing functions are split into three parts: the central unit (CU), distributed unit (DU), and a remote unit (RU) [Das et al. 2020], as depicted by Figure 2.4.

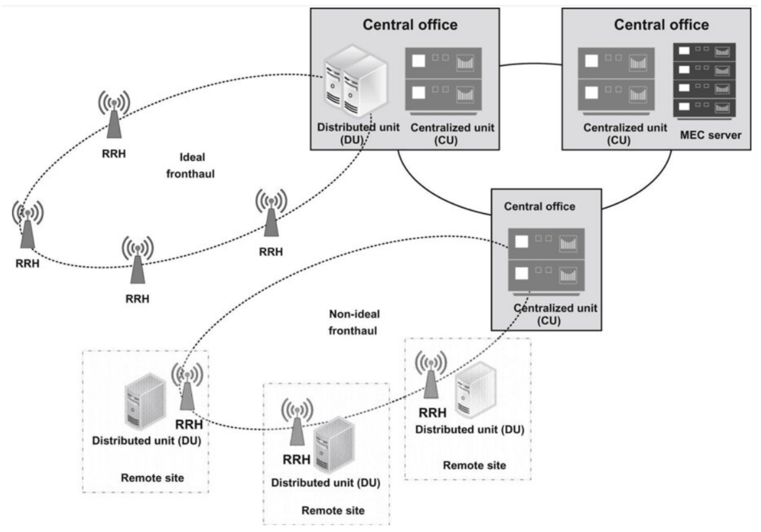


Figure 2.4: CU/DU-based C-RAN architecture [Ahmadi 2019].

As 5G networks will bring progressive densification of mobile cells, the cost of the optical transport network will soar to unsustainable levels, if cells are connected through dedicated point-to-point fibre. In addition, point-to-point solutions provide little flexibility in redirecting RUs connectivity between edge cloud nodes during migration events. For this reason, PON is being considered as a possible cost-effective solution to support optical Mobile Fronthaul (MFH) transport, as it can use an already deployed Optical Distribution Network (ODN) to provide fronthaul transport for RUs along with serving residential users.

However, achieving low latency for using C-RAN and 5G is a major challenge for TDM-PONs in the upstream direction, because of the Time Division Multiple Access (TDMA) nature of the operation, which employs the DBA algorithm to provide the time slots for transmitting to each ONU on the system. Network function virtualisation could be used to help overcome this limitation, but the extra layers added by the software layers (e.g. OS, hypervisor) add more latency to the process, capping the benefits early noted. In-Network Computing must be an enabler for low-latency applications since the offloading of virtual network functions running inside a hypervisor onto a network data plane will cut these extra layers added before by the NFV paradigm.

2.1.1.2 Cloud Robotics

Robotic systems have brought significant socioeconomic impacts to human lives over the past few decades. For example, industrial robots have been widely deployed in factories to execute various tasks, from repetitive ones to dangerous tasks. These programmed robots have been very successful in industrial applications due to their high endurance, speed, and precision in structured factory environments. To extend the functional range of these robots or to deploy them in unstructured environments, robotic technologies are integrated with network technologies to foster the emergence of networked robotics.

A networked robotic system refers to a group of robotic devices that are connected via a wired and/or wireless communication network. Networked robotics applications can be classified as either teleoperated robots or multi-robot systems. In the former case, a human operator controls or manipulates a robot at a distance by sending commands and receiving measurements via the communication network.

Networked robotics, similar to standalone robots, faces inherent physical constraints as all computations are conducted on board the robots, which have limited computing capabilities. Information access is also restricted to the collective storage of the network. With the rapid advancement of wireless communications and recent innovations in cloud computing technologies, some of these constraints can be overcome through the concept of cloud robotics, leading to more intelligent, efficient and yet cheaper robotic networks.

Cloud computing provides a natural venue to extend the capabilities of networked robotics. NIST [Mell et al. 2011] defines cloud computing as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” Through its three service models (i.e., software, platform and infrastructure), it enables tremendous flexibility in designing and implementing new applications for networked robotics.

The Robot Operating System (ROS) was developed as a framework that provided the software infrastructure for cloud robotics employment. As a framework, ROS provides a set of tools and libraries that simplify the task of creating a new robot and controlling it. Indeed, ROS is the most popular framework in robotics research and it also grows in terms of industrial use. This makes ROS a worthwhile target for attackers, especially since security is not addressed by the core framework itself, suffering from security issues due to its open architecture and flexibility project. These concerns could be increased if we bring the security flaw to the premises, based on a jeopardised device (e.g. a server or network device), assuming that these devices and the overall premises should be considered as a secure zone.

2.1.2 Network Function Virtualisation

The presence of proprietary hardware-based network appliances, known as middle-boxes, is a crucial part of the operation of today's computer and telecommunications networks. It supports a diverse set of network functions, such as firewalls, intrusion detection systems, load balancers, NAT, caches, and proxies [Martins et al. 2014]. For instance, in company networks, the number of these middleboxes is equivalent to the number of routers and switches deployed [Sherry et al. 2012].

However, the presence of these middle-boxes brings several problems, such as [Han et al. 2015]: networks have become very complex with a wide variety of proprietary elements; the time to bring new services and functionalities to the market is high, as it depends on the production of new hardware; the operation of the networks is costly and depends on specialised knowledge in each proprietary platform; the costs of acquiring equipment to meet network demands are high, but they quickly reach obsolescence; lack of flexibility and scalability, as resources, cannot be moved according to demands, and need to be scaled to the peak scenario; there are big barriers to innovation since it requires a significant investment to develop a device in hardware; the technologies of different manufacturers are incompatible with each other and do not allow reuse of hardware and software.

To address these problems and implement less costly and more flexible network infrastructure, NFV takes network functions of commercial appliances, actually dedicated hardware and software, to off-the-shelf equipment. By executing software-based functionality on commodity hardware with virtualisation technologies for processing, storage, and networking, as exemplified in Figure 2.5, NFV frees a wide range of innovative solutions for new networks.

The NFV objectives can be summarised as listed below [ETSI NFV ISG 2014]:

- Reduced CAPEX when compared to a specific target of hardware implementations: This goal is achieved by adopting commodity hardware and virtualisation techniques, reducing the number of different hardware architectures and resource sharing.

- Scalability and flexibility network functions: it is possible to decouple location from functionality and allocate network functions in the most appropriate places according to demands, increasing resiliency through visualisation, and making resource sharing easier. It becomes a strong groundbreaking mechanism for the next network generation as 5G networks.

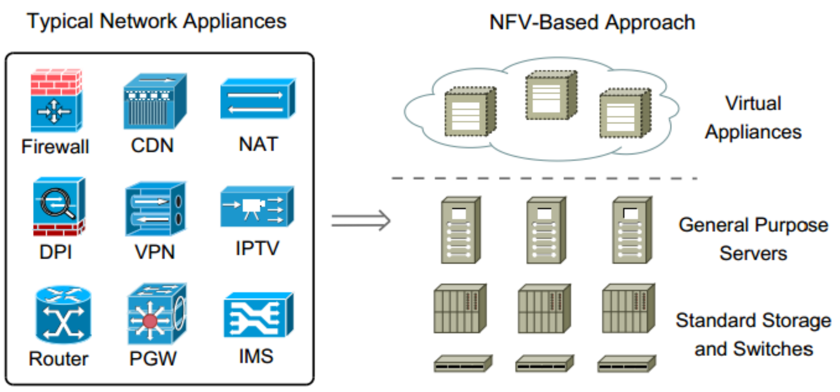


Figure 2.5: NFV implementation of network functions using virtualisation techniques over standard hardware. Source: [Han et al. 2015].

- Software-based implementation: it incentives innovation and is time-consuming for creating a new product which reaches a specific market share.
- Reduced OPEX (operational expenses) by automating the procedures and making faster decisions for a specific event (e.g., fail, workload, and new demand).
- Reduced power consumption by migrating workloads and shutting down unused hardware.
- Interoperability through open and standardised interfaces between the network functions, the underlying infrastructure, and the associated management entities. In this way, the elements of NFV architecture can be implemented by different vendors.

These objectives have a huge impact on the business model of telecommunications networks, specifically in the new 5G networks. Consequently, this sector has received more investments in NFV standardisation, as discussed in the next section. However, it is essential to emphasise that the paradigm is applicable in both computer networks and telecommunications networks, especially at a time when both are converging to a resource-delivery model based on cloud computing.

2.1.2.1 Cryptographic hash functions

Cryptographic hash functions are one of the most important tools in the field of cryptography and are used to achieve a number of security goals like authenticity, digital signatures, pseudo number generation, digital time stamping and others.

Hash functions map a large collection of messages into a small set of message digests and can be used for error detection, by appending the digest to the message during the transmission. The error will be detected if the digest of the received message, on the receiving end, is not equal to the received message digest. With the advent of public key cryptography and digital signature schemes, cryptographic hash functions gained much more prominence. Using hash functions, it is possible to produce a fixed-length digital signature that depends on the whole message and ensures the authenticity of the message. To produce a digital signature for a message M, the digest of M, given by $H(M)$, is calculated and then encrypted with the secret key of the sender [Sobti and Geetha 2012].

In order to provide security services, cryptographic hash algorithms need to guarantee some properties which are not necessarily guaranteed by general-purpose hash functions. The **one-way** or **preimage resistance** property of cryptographic hash functions implies that it is computationally infeasible to compute the message M given its hash $DM(M)$. The **second preimage resistance** property means that, given a hash value $DM(M)$, it is computationally infeasible to find a different message $M' \neq M$ that yields the same hash value. The **pseudo-randomness** property means that the hash value of a message must expose statistical randomness. Finally, the **collision resistance** property means that it is computationally infeasible to find a pair of messages $M1$ and $M2$ which produce the same hash value [Martino and Cilardo 2020].

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions defined by the National Institute of Standards and Technology (NIST) and published as the Federal Information Processing Standard (FIPS) 180, Secure Hash Standard (SHS). The algorithm is an iterative, one-way hash function that can process a variable-size message to produce a fixed-size condensed representation called a message digest. This algorithm enables message integrity verification, i.e., any change to the message will, with a very high probability, result in a different message digest. This property is useful in the generation and verification of digital signatures and message authentication codes, and in the generation of random numbers or bits [Dang et al. 2015]. The SHA-2 family of cryptographic hash functions was first announced in 2001 and includes SHA-224, SHA-256, SHA-384 and SHA-512, named according to the length of the message digest created by each one, suppressing the previous SHA-1 implementation due to security improvements.

As cryptographic hash functions are compute-intensive applications, which begs the question of extending P4, and its hardware platforms, into cryptographic algorithms. This would enable offloading secure applications/tasks to the data plane. The expected benefits are twofold: i) saving CPU resources for other applications running at the hypervisor and their tenants and ii) reducing latency and increasing the number of processed packets per time unit. The latter benefit will come from avoiding the operating system and hypervisor stacks in between the network and the SHA application.

2.1.3 Programmable network hardware

There is paradoxically flexible FPGA-based hardware that is limited by its complex hardware description languages and powerful ASICs are limited by a reduced set of functionalities made available by their vendors. SmartNICs emerge as a compromise solution. Figure 2.6 shows the architecture of a typical SmartNIC with an in-networking processor [CAVIUM 2019, NETRONOME 2019]. It is basically comprised of a multi-core NPU (Networking Processor Unit), local memory and an SR-IOV (Single Root I/O Virtualisation) interface. These architectural components connect each other using a high-performance interconnection bus. They also provide on-chip/off-chip accelerators to speed up certain operations (e.g., encryption, hashing, look-ups, buffer management, etc).

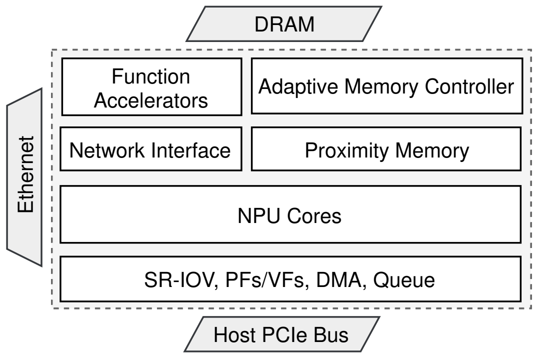


Figure 2.6: SmartNIC architecture simplified block diagram.

In SR-IOV-compliant hardware, each host connects to a privileged physical function (PF), while each virtual machine connects to its own virtual function (VF). A VF is exposed as a unique hardware device to each VM, allowing the VM, and consequently, VNF resided therein, direct access to the actual hardware, and yet isolating VM data from other VMs. An SR-IOV-compliant SmartNIC also has an embedded switch to forward packets to the right VF based on the MAC address. In addition to SR-IOV, Network Processing Unit (NPU) brings programmability to the data path by using some high-level programming language, such as P4 language. All this combined and properly used, enables improved throughput, reduced CPU utilisation, low latency, and scalability.

Despite the vast literature on offloading DC applications by in-network computing, few frameworks have tackled the NFV challenges. In particular, the state-of-the-art of NFV platforms has merely replaced monolithic hardware NFs with their monolithic software components. Interestingly, a large number of common functionalities (e.g., packet header parsing, packet classification, etc.) are repeated across different VNFs [Rahman Chowdhury et al. 2019]. Thus, our proposal brings its novelty by introducing a framework that allows programmers to decompose and deploy VNF applications to eNFs using a NIC-side in-networking processor.

2.1.3.1 P4: Programming Protocol-Independent Packet Processors

Programming Protocol-independent Packet Processors (P4) is a domain-specific language for network devices, specifying how data plane devices (switches, NICs, routers, filters, etc.) process packets. P4 programs define how the various programmable blocks of a target architecture are programmed and connected. While P4 was initially designed for programming switches, its scope has been broadened to cover a large variety of devices, which are defined by the generic term *target*.

As a concrete example of a *target*, Figure 2.7 illustrates the difference between a traditional fixed-function switch and a P4-programmable switch. In a traditional switch, the manufacturer defines the data-plane functionality. The control plane controls the data plane by managing entries in tables (e.g. routing tables), configuring specialised objects (e.g. meters), and processing control packets (e.g. routing protocol packets) or asynchronous events, such as link state changes or learning notifications [The P4 Language Consortium 2021].

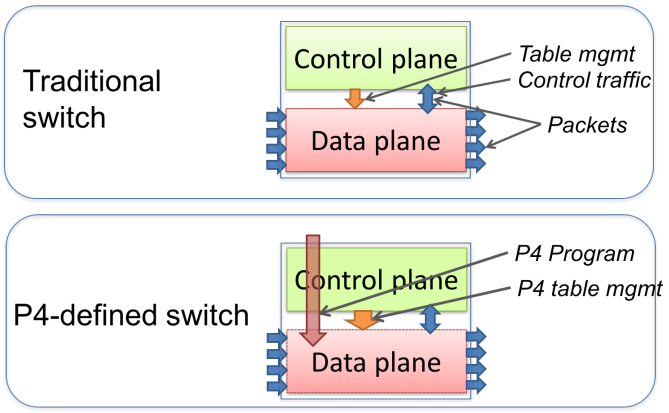


Figure 2.7: Traditional switches vs. programmable switches [The P4 Language Consortium 2021].

Portable Switch Architecture (PSA) Model is a target architecture that describes common capabilities of network switch devices that process and forward packets across multiple interface ports. The PSA Model has six programmable P4 blocks and two fixed function blocks, as shown in Figure 2.7. The behaviour of the programmable blocks is specified using P4 language. The Packet buffer and Replication Engine (PRE) and the Buffer Queuing Engine (BQE) are target-dependent functional blocks that might be configured for a fixed set of operations. [The P4.org Architecture Working Group 2020]

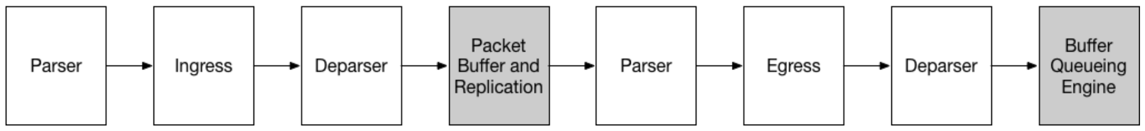


Figure 2.8: Portable Switch Pipeline [The P4.org Architecture Working Group 2020].

Incoming packets are parsed and validated, and then passed to an ingress match action pipeline, which makes decisions on where the packets go. The ingress *deparser* P4 code specifies the packet contents to be sent to the packet buffer, and what metadata related to the packet is carried with it. After the ingress pipeline, the packet may optionally be replicated (i.e. copies made to multiple egress ports), and then stored in the packet buffer. For each such egress port, the packet passes through an egress parser and match-action pipeline before it is *deparsed* and queued to leave the pipeline. Figure 2.9 shows all possible paths for packets that must be supported by a PSA implementation.

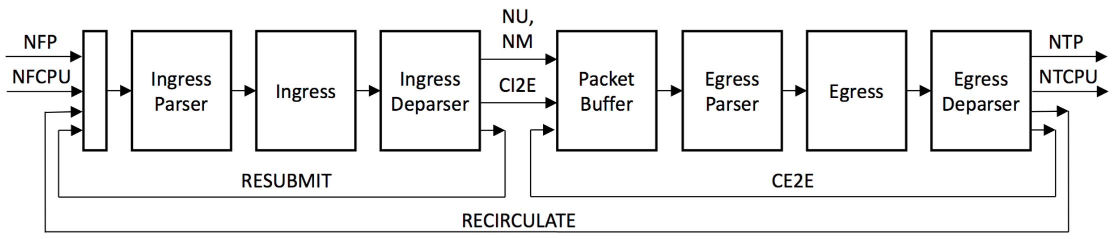


Figure 2.9: Packet paths in PSA [The P4.org Architecture Working Group 2020].

In P4, a table describes a match-action unit. The lookup table is a finite map whose contents are manipulated asynchronously (read/write) by the target control-plane, through a separate control-plane API. Syntactically a table is defined in terms of a set of key-value properties. The structure of a match-action unit is shown in Figure 2.10.

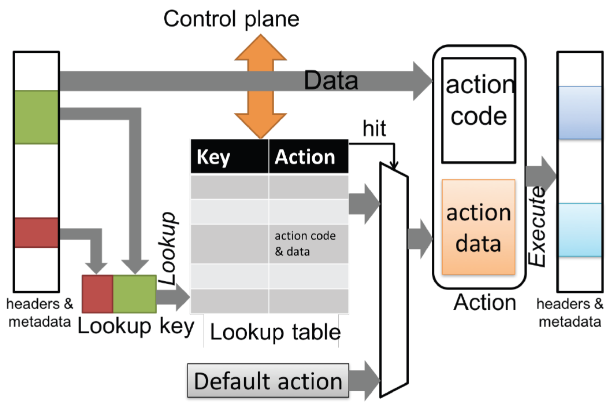


Figure 2.10: Match-Action Unit Dataflow. [The P4 Language Consortium 2021].

A P4-programmable target differs from a traditional device in two essential ways [The P4 Language Consortium 2021]:

- The data plane functionality is not fixed in advance but is defined by a P4 program. The data plane is configured at initialisation time to implement the functionality described by the P4 program (shown by the long red arrow) and has no built-in knowledge of existing network protocols.

- The control plane communicates with the data plane using the same channels as in a fixed-function device, but the set of tables and other objects in the data plane is no longer fixed since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

Hence, P4 can be said to be protocol independent, but it enables programmers to express a rich set of protocols and other data plane behaviours. The core abstractions provided by the P4 language [The P4 Language Consortium 2021] are also depicted by Figure 2.11:

- **Header types** describe the format (the set of fields and their sizes) of each header within a packet.

- **Parsers** describe the permitted sequences of headers within received packets, how to identify those header sequences and the headers and fields to extract from packets.
- **Tables** associate user-defined keys with actions. P4 tables generalise traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions.
- **Actions** are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control plane at runtime.
- **Match-action** units perform the following sequence of operations:
  - Construct lookup keys from packet fields or computed metadata,
  - Perform table lookup using the constructed key, choosing an action (including the associated data) to execute, and
  - Finally, execute the selected action.
- **Control flow** expresses an imperative program that describes packet processing on a target, including the data-dependent sequence of match-action unit invocations. *Deparsing* (packet reassembly) can also be performed using a control flow.
- **Extern** objects are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behaviour is hard-wired (e.g., checksum units) and hence not programmable using P4.
- **User-defined metadata**: user-defined data structures associated with each packet.
- **Intrinsic metadata**: metadata provided by the architecture associated with each packet—e.g., the input port where a packet has been received.

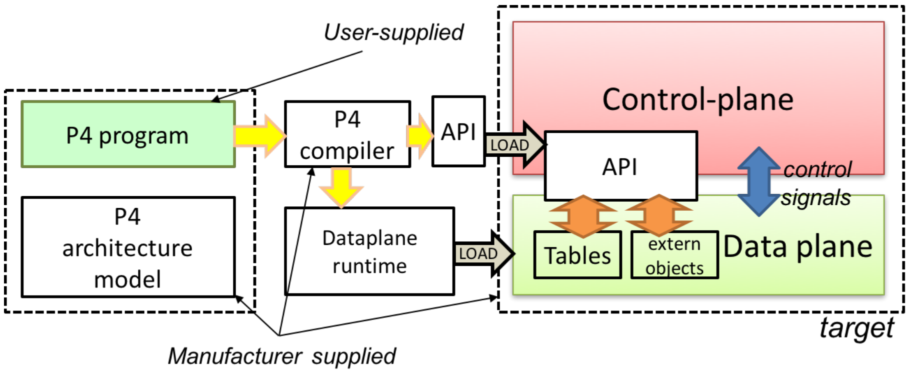


Figure 2.11: Programming a target with P4 [The P4 Language Consortium 2021].

2.2 Related works

This section describes the related works focusing on network device programmability and its proposed Frameworks. We position the capabilities of programming and deploying applications on hardware, showing the approaches of current solutions. Furthermore, here we compare our solutions with related works in the area of network programmability and its enablers, pointing out where PIaFFE differs from the current state-of-the-art proposals on data plane programmability.

Using network programmable devices presents the opportunity to run custom logic in the network at line rate speeds. However, programming them is not straightforward and presents many challenges. Various frameworks and languages have been proposed to program these devices, including high-level languages like C (high complexity), as well as domain-specific languages (low complexity) like Programming Protocol-Independent Packet Processors (P4). Thus, this section will expose the current solutions and the state of the art for network hardware programming, as well as how our proposal of framework stands apart from current implementations.

2.2.1 ClickNP

Click Network Processor (ClickNP) [Kun et al. 2016] is an FPGA-accelerated platform for highly flexible and high-performance NF processing on commodity servers. ClickNP addresses the programming challenges of FPGA by providing a modular architecture, resembling the well-known Click model [Kohler et al. 2000], where a complex network function is composed using simple elements. These elements are written with a high-level C-like language and are cross-platform and can be ported to binaries on CPU or low-level hardware description language (HDL) for FPGAs.

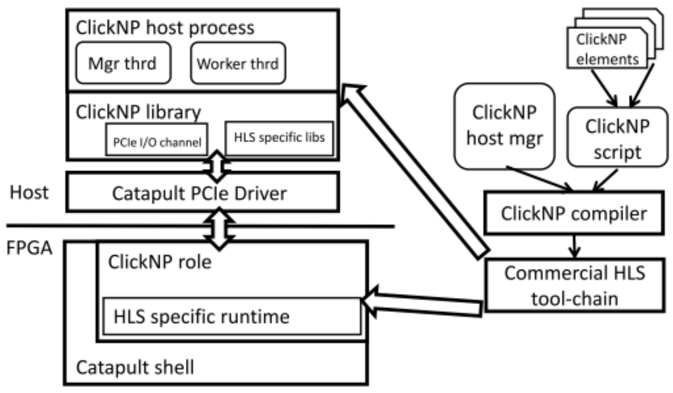


Figure 2.12: The architecture of ClickNP [Kun et al. 2016].

However, ClickNP may require the programmer of a functional element to apply specific hardware-related optimisations, when the compiler fails to apply its automated optimisation. Further, updating a function requires a new synthesis and flashing of the FPGA design, a process that takes a long time to be done, measured in hours. Thus, ClickNP was conceived for general network functions, so in ClickNP we can implement arbitrary packet parsers, states and actions, which is suitable for

FPGA-based hardware. On the other side, the P4 programming language assumes a pipeline of match-action tables and is more suitable for programmable switching chips or network interface cards, which are easy to program and use, leveraging our proposal of a framework for embedding applications using this domain-specific language.

2.2.2 Azure Accelerated Networking (AccelNet)

AccelNet provides near-native network performance in a virtualised environment, offloading packet processing from the host CPU to the Azure SmartNIC. Building upon the software-based Virtual Filtering Platform (VFP) host SDN platform for Azure, and the hardware and software infrastructure of the Catapult program, AccelNet provides the performance of dedicated hardware, with the programmability of software running in the hypervisor [Firestone et al. 2018].

The authors describe a mechanism called Generic Flow Table Offload (GFT), which is a match-action language that defines transformation and control operations on packets for one specific network flow. As depicted by Figure 2.13, the flow table may not contain a matching rule for a given packet. In these cases, the offload hardware will send the packet to the software layer as an $ExceptionPacket$. Exception packets are most common on the first packet of a network flow when the flow is just getting established. A special virtual port ($vPort$) dedicated to the hypervisor is established for exception packets. When the FPGA receives an exception packet, it overloads the 802.1Q VLAN ID tag in the packet to specify that it is an exception path packet, and forwards the packet to the hypervisor $vPort$. VFP monitors this port and performs the necessary flow creation tasks after determining the appropriate policy for the packet's flow. If the exception packet was destined for a VM on the same host, the VFP software can deliver the packet directly to the VM. If the exception packet was outbound (sent by a VM for a remote destination), then the VFP software must resend the packet to the SmartNIC, which it can do using the same dedicated hypervisor vPort. VFP also needs to be aware of terminated connections so that stale connection rules do not match new network flows. When the FPGA detects termination packets such as TCP packets with SYN, RST or FIN flag set, it duplicates the packet – sending the original packet to its specified destination, and an identical copy of the packet to the dedicated hypervisor $vPort$.

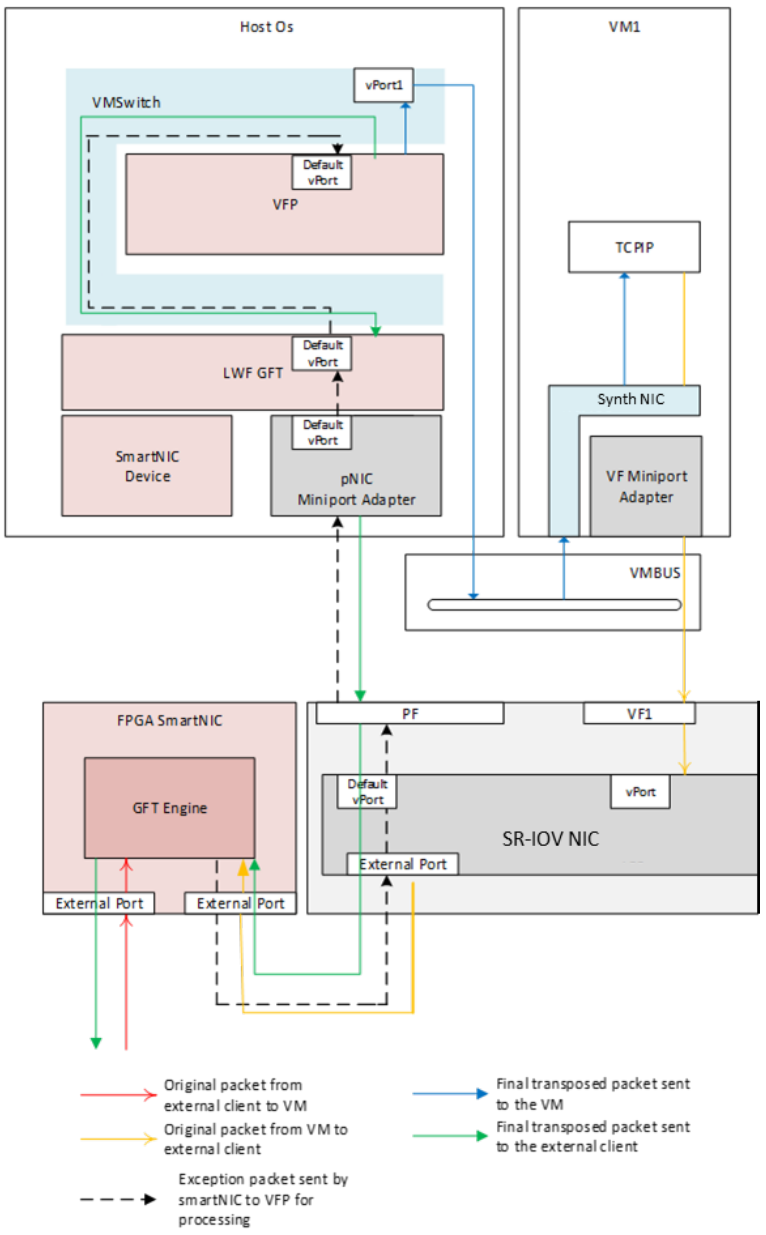


Figure 2.13: The SmartNIC GFT architecture, showing the flow of exception packets from the FPGA to software to establish a flow offloaded in hardware [Firestone et al. 2018].

In this work, the authors consider using FPGA-based SmartNICs due to their superior performance when compared with traditional SmartNICs. However, various aspects of the FPGA design ecosystem, such as the need to globally synthesise, placement and the difficulty of developing using low-level languages, should be raised in order to employ the overall proposal. Another aspect that needs attention is the proposed solution is designed for working only in the Microsoft Azure ecosystem, from the hardware to the software layers, limiting the scope for experimenting and implementation for testing. In contrast, the PIaFFE framework was conceived to be used in multiple environments, enabling the prototyping and deployment in different scenarios and virtualisation technologies (e.g. QEMU, Linux namespaces, Docker), using a programmable network interface for offloading and co-execution of applications inside this hardware.

2.2.3 FLOEM

FLOEM is a toolkit which provides language, compiler, and runtime tools for programming NIC-accelerated applications, enabling offload design exploration by providing programming abstractions to assign computation to hardware resources; control mapping of logical queues to physical queues; access fields of a packet and its metadata without manually marshalling a packet; use a NIC to memorise expensive computation and interface with an external application. The FLOEM compiler contains three primary components that: (1) translate a data-flow program with elements into C programs, (2) infer minimal data transfers across queues, and (3) expand the high-level caching construct into primitive elements [Phothilimthana et al. 2018], as depicted in Figure 2.14.

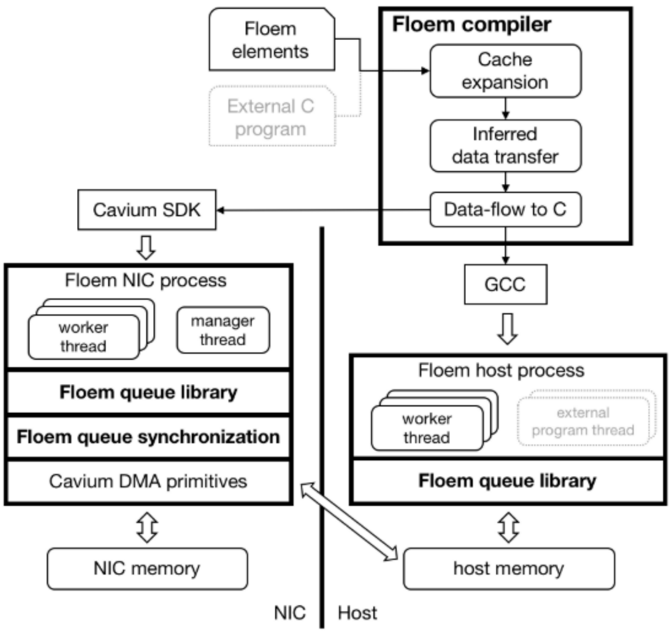


Figure 2.14: FLOEM system architecture. [Phothilimthana et al. 2018].

FLOEM aims to simplify the development of NIC-accelerated network applications by providing a unified framework to implement an application that is split across the CPU and NIC. However, it is based on a static offloading approach of specialised tasks, focusing on state migration between hosts other than developing a high-level application that should be able to interact with network traffic data at higher-level layers (after the traditional headers), as the PIaFFE framework can do through a mechanism for steering traffic flow between hardware and software, empowering network applications and saving resources in both physical hosts and virtual machines.

2.2.4 Lambda-NIC

The λ-NIC is a framework for running interactive serverless workloads entirely on SmartNICs. λ-NIC implements a new programming abstraction (Match+Lambda) along with a machine model – an extension of P4's match-action abstraction with more sophisticated actions – and helps address some shortcomings of SmartNICs by using techniques that explore the small nature of lambdas [Choi et al. 2019].

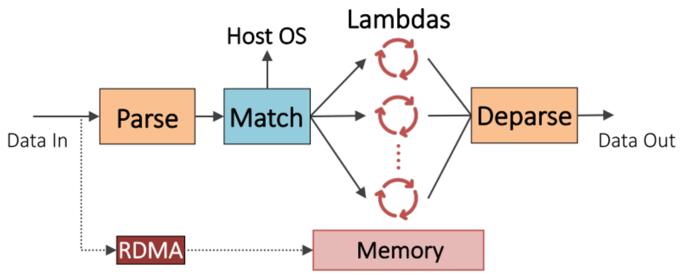


Figure 2.15: λ-NIC's abstract machine model. [Choi et al. 2019]

In λ-NIC, users write their Match+Lambda workloads against an abstract machine model, as shown in Figure 2.15. In this model: (1) lambdas are independent programs that do not share state and are isolated from each other; only a matching rule can invoke these functions; (2) The matching stage serves as a scheduler (analogous to the OS networking stack) that forwards packets to the matching lambdas or the host OS; (3) a parser handles packet operations (like header identification), and lambdas operate directly on the parsed headers [Choi et al. 2019].

The authors argue that these properties of the Match+Lambda machine model can make it easier for software developers to express serverless functions by separating out the parsing and matching logic, as well as for hardware designers to efficiently support the model on their target SmartNICs. Thus, the abstract machine model enables unique optimisations that let serverless workloads run as lambdas in parallel without any interference from each other.

However, this approach is restricted to serverless environments and does not take into account the possibility of parsing data from parts of the packet payload. PIaFFE can be extended to other paradigms – such as NFV – providing more flexibility when creating new micro-applications to be embedded into a programmable network card using P4 language, also enabling the possibility to parse data from packet payload, creating "pseudo-headers" based on application specification.

2.2.5 NICA

NICA is a hardware-software co-designed framework for inline acceleration of the application data plane on F-NICs in multi-tenant systems based on FPGA NIC Server Acceleration. A new *ikernel* programming abstraction, tightly integrated with the network stack, enables application control of F-NIC computations that process application network traffic, with minimal code changes. In addition, NICA's virtualisation architecture supports fine-grain time-sharing of F-NIC logic and provides I/O path virtualisation. Together these features enable cost-effective sharing of F-NICs across virtual machines with strict performance guarantees [Eran et al. 2019].

Figure 2.16 shows the main NICA components with a single physical Accelerator Functional Unit (AFU) [Corporation 2020]. NICA comprises three layers: application-visible OS abstractions and services inside a VM integrated with the network stack; the hypervisor layer for managing F-NIC resources and QoS; the hardware layer which includes the support for OS abstractions, physical AFU logic (pAFU), a virtualisation framework exposing virtual AFUs (vAFUs), and a hardware runtime with network I/O services for application-level message processing on AFUs.

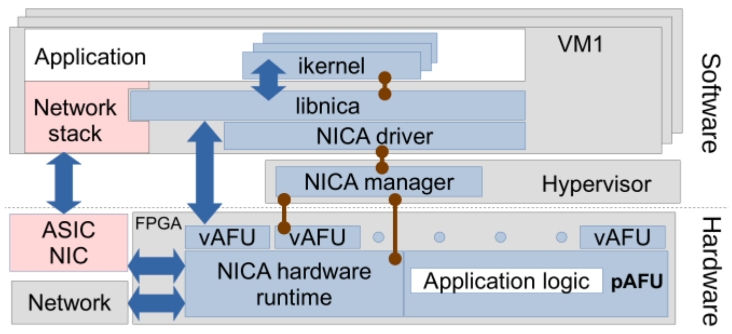


Figure 2.16: NICA overview. [Eran et al. 2019].

NICA is a software/hardware framework that introduces specific software abstractions that connect FPGA blocks with a user program running on a general-purpose CPU. However, applications can only be designed by composing the provided hardware blocks, not enabling the possibility to connect other abstractions than the pre-defined ones, hampering new implementations due to the difficulty of developing using a low-level programming language related to FPGA-based network cards. Therefore, PIaFFE uses a high-level domain-specific language, which is not only easy to be used but also fits with data plane programmability, with a number of constructs optimised for network data forwarding and processing.

2.2.6 iPipe

The iPipe proposal provides an actor programming model and a runtime system that includes the following components: (1) a lightweight work-conserving scheduler that maximises the NIC processor utilisation without hurting the target link bandwidth; (2) distributed object abstractions that enable efficient use of remote host memory and flexible actor migration; (3) a cross PCIe messaging tier and a shim networking stack for communicating with host processors and RX/TX ports; (4) a dynamic resource manager that supports weighted max-min fairness on the link bandwidth and provides execution latency guarantees for multi-application consolidations. Programmers can express applications using the actor model and rely on the runtime to automatically schedule (and/or migrate) the actor execution on either the in-networking processor or the host CPU [Liu et al. 2019].

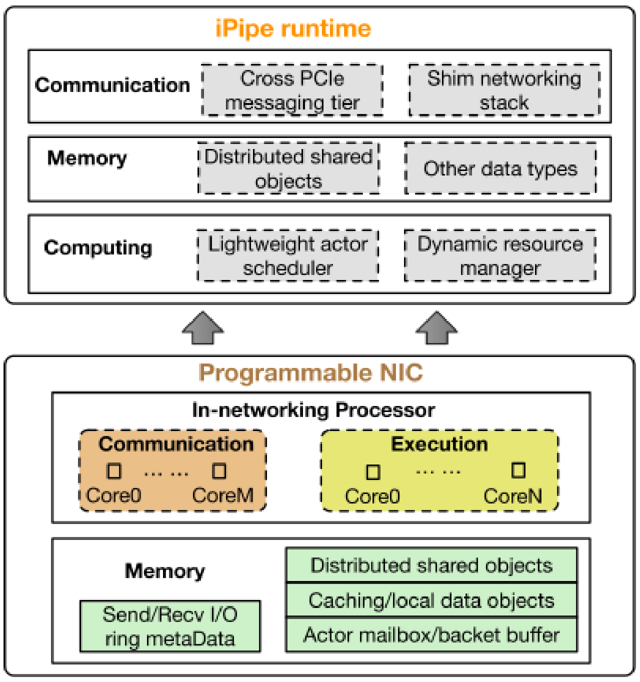


Figure 2.17: An overview of the iPipe runtime on the programmable NIC [Liu et al. 2019].

The iPipe runtime (see Figure 2.17) manages and schedules the computational resources of the programmable NIC and exposes APIs for managing the computing, memory, and communication resources. It provides a distributed shared object abstraction that enables flexible actor migration, as well as support for a software-managed cache and NIC-local objects. In the communication layer, iPipe provides a cross-PCIe message passing tier for communications between the host and the NIC, as well as a simplified networking stack that delivers packets from/to TX/RX ports. iPipe has an actor scheduler, which schedules (and/or migrates) actor execution instances on the host/NIC, and uses a resource manager that supports multi-tenancy based on user policies.

However, iPipe framework can offload only a portion of these applications to a NIC as a bump-in-the-wire and need CPUs to do the remaining processing. Furthermore, the system has proposed using the programmability for transport protocol offload and some essential network functions, such as load-balancing of congestion control, leaving a gap for application-related offloading, such as high-level applications running inside a network programmable hardware which is easily filled using PIaFFE framework for embedding a range of applications, from essential network-related functions to advanced applications that require dynamic offloading of network packet processing.

2.2.7 FlowBlaze

FlowBlaze is an open abstraction for building stateful packet processing functions on NetFPGA SmartNIC hardware using the Extended Finite State Machines (EFSM) model, introducing the explicit definition of flow state and allowing to leverage of flow-level parallelism. In order to accomplish it, FlowBlaze adapts match-action tables to describe the evolution of a network flow's state employing EFSM, [Pontarelli et al. 2019].

According to Figure 2.18, the FlowBlaze machine model consists of a stateless element and a stateful element. A stateless element is the same as a "match–action" table. And stateful element is split into Flow Context Table, EFSM table and update function. The Flow Context Table is used to save states of flows, while the EFSM table is used to implement functions abstracted by EFSM. The update function is responsible for executing the state update. Similarly, a small stash memory is used to handle hash collision, providing scalability for the Flow Context Table. FlowBlaze solves the race condition problem with a simple scheduler scheme to guarantee the consistency of flow states.

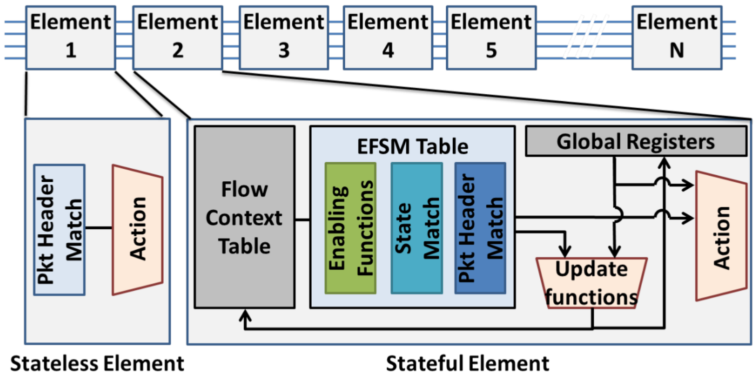


Figure 2.18: FlowBlaze machine model. [Pontarelli et al. 2019].

While FSMs provide a naturally suited abstraction for stateful packet processing, realising scalable stateful packet processing systems based on programmable data plane systems is still challenging. In particular, creating low-latency stateful applications in programmable ASICs could be unmanageable due to target-specific requirements and constrained memory and stateful ALU resources.

PIaFFE Framework defines a pathway to create applications into SmartNICs by finding the limits of the network programmable device, sharing resources between commodity servers and data plane, expressing the capabilities and limits of each platform, and leveraging packet processing and forwarding, which can be used to develop and deploy flexible embedded stateful applications in both levels: hardware and software.

2.2.8 FairNIC

FairNIC is a system to provide performance isolation between tenants utilising the full capabilities of a commodity SoC SmartNIC, based on Cavium LiquidIO 2360s. FairNIC provides a set of per-resource isolation techniques to ensure that each resource is partitioned (wherever possible) or multiplexed according to tenant service-level objectives. Thus, it effectively dedicates a portion of the SmartNIC's end-to-end packet processing pipeline to each tenant as shown in Figure 2.19, where individual resources may be allocated in different proportions depending on the needs of each tenant, in colours orange and blue. Some resources are consumed by FairNIC itself, depicted in pink [Grant et al. 2020].

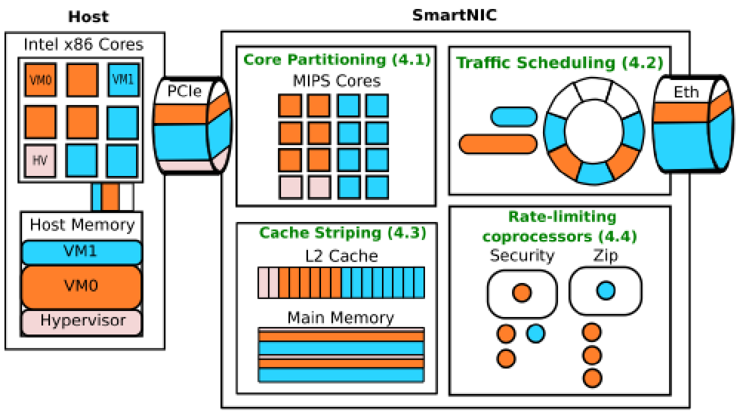


Figure 2.19: FairNIC sharing resources between two tenants. [Grant et al. 2020].

While using statically partitioning resources to reduce interference between tenants, FairNIC sacrifices CPU utilisation because each task must be provisioned with enough resources to accommodate peak load on the CPU. Moreover, the main focus of the FairNIC is a multi-tenant cloud environment and its sharing resources aspects, where one of the aspects of the PIaFFE framework is to define a way to coordinate the offloading at the edge, between a single application running on the host's software layer and an embedded network function running on SmartNIC hardware's layer (one-to-one relationship), which is more expected by the overall employment at the edge environment.

2.2.9 P4Runtime

The P4Runtime API is a control plane specification for controlling the data plane elements of a device defined or described by a P4 program. Figure 2.20 represents the P4Runtime Reference Architecture. The device or target to be controlled is at the bottom, and one or more controllers are shown at the top. A multi-master protocol allows more than one controller to participate, and a role-based arbitration scheme ensures only one controller has to write access to each read/write entity, or the pipeline config itself. Any controller may perform read access to any entity or the pipeline config. Later sections describe this in detail. For the sake of brevity, the term controller may refer to one or more controllers.

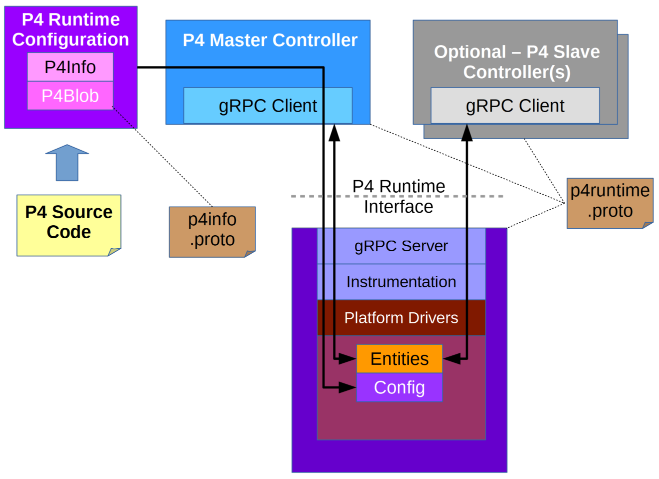


Figure 2.20: Programming a target with P4 [The P4.org API Working Group 2020].

The P4Runtime API defines the messages and semantics of the interface between the client(s) and the server. The API is specified by the p4runtime.proto Protobuf file, which is available on GitHub as part of the standard. It may be compiled via $protoc$ – the Protobuf compiler – to produce both client and server implementation stubs in a variety of languages. It is the responsibility of target implementers to instrument the server.

The controller can access the P4 entities which are declared in the P4Info metadata. The P4Info structure is defined by p4info.proto, another Protobuf file available as part of the standard. The controller can also set the $ForwardingPipelineConfig$, which amounts to installing and running the compiled P4 program output, which is included in the $p4\_device\_config$ Protobuf message field, and installing the associated P4Info metadata. Furthermore, the controller can query the target for the $ForwardingPipelineConfig$ to retrieve the device config and the $P4Info$.

P4Runtime is one of the most commonly used data plane APIs that is standardised in the API Working Group [The P4.org API Working Group 2020] of the P4 Language Consortium for controlling data plane elements which use P4 as the basis for programming. In such a manner, it does not support programmers for the development and deployment of micro-applications into programmable network hardware, thus it is not suitable for the main goal of the proposal of this Thesis, which is to create a framework for embedding applications into programmable network cards, covering from the development to deployment of such applications.

2.2.10 EP4

EP4 relies on an application-aware extended P4-based network architecture that offers hosted applications an extensible catalogue of services through its control plane. The latter configures a PDP that can achieve minimal parsing and processing for fast-tracked packets as well as customised processing and forwarding for other packets. An extended parser (eParser) performs the first task, which reduces the necessary latency experienced by packets. Alternatively, adaptive processing is achieved using an enhanced processor (eProcessor) that optionally parses customised headers using just-in-time programmable parsers. It then executes selected P4 packet processing pipelines implementing different services [Karrakchou et al. 2021].

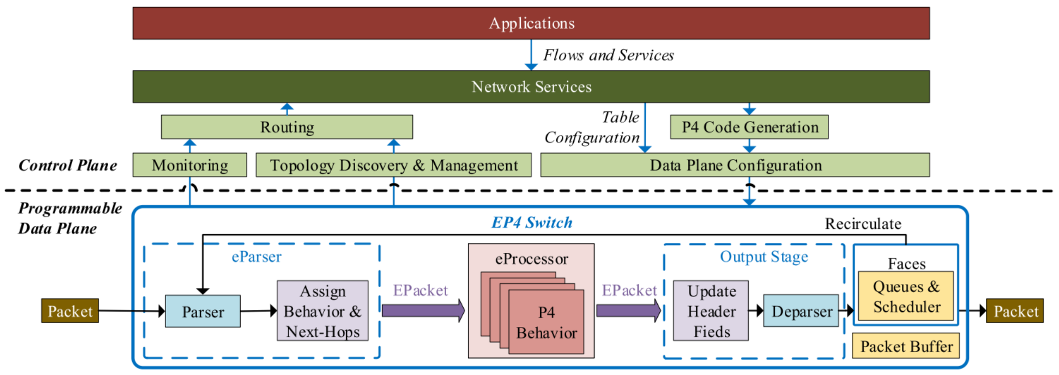


Figure 2.21: Overview of the EP4 architecture [Karrakchou et al. 2021].

As shown in Figure 2.1.3.1, the control plane must discover the network's topology and compute the different routes before placing instances of each network service in switches along the flow paths. For every flow and switch, the control plane will generate the P4 code corresponding to the chain of service instances placed on that switch. The data plane configuration generated by the control plane consists of table values, corresponding to the flows' route configuration, and the associated P4 code for every flow and switch. It is then installed in the data plane using the P4Runtime API.

The EP4 framework proposes an application-aware network architecture using a P4 programmable data plane based on an extensible catalogue of network services to hosted applications and translates them to configurations at the data plane. Despite the overall architecture, the proposal does not target the application implementation itself on the data plane, limiting the scope to network-related functions (e.g. best-effort service). PIaFFE can be used for the implementation and deployment of applications into a programmable data plane using a well-defined element for controlling the amount of data that will be processed by hardware and software, increasing the expressiveness of a range of applications that can be conceived using the framework.

2.2.11 DPPx

Data Plane Programmability and Exposure framework (DPPx) implements a framework for P4-based data plane programmability and exposure which allows for to enhancement of NFV services. They introduce data plane modules written in P4 which can be leveraged by the application plane. It integrates P4-compatible programmable devices into OpenStack, which is the main platform to build the Virtual Infrastructure Manager (VIM) for the ETSI MANO [ETSI NFV ISG 2014]. DPPx provides lifecycle management (LCM) for data plane modules, which are defined in the P4 language, and exposes northbound LCM API to external users [Osinski et al. 2019].

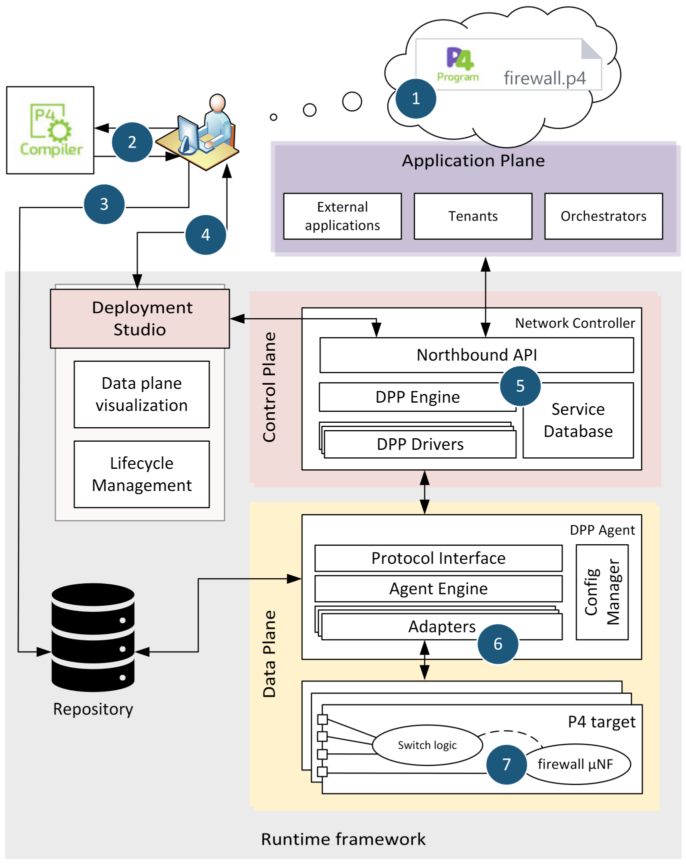


Figure 2.22: The DPPx system architecture [Osinski et al. 2019]

According to Figure 2.22 Data plane of the system is composed of programmable switches and DPP Agents managing them. DPP Agent uses a target-specific adapter to configure data plane modules. The DPP Agent is composed also of a northbound protocol interface to communicate with a control plane. The Agent Engine translates protocol messages and invokes a relevant adapter. The Config Manager of the DPP Agent is used to configure an adapter and access parameters associated with a particular target platform. A control plane of a system consists of a Network Controller, which is responsible for controlling the lifecycle of data plane modules. It implements northbound API and translates high-level commands to low-level protocol messages. A part of the Network Controller is a local Service Database, that stores record about currently installed data plane modules. The DPP Engine coordinates requests for handling and database transactions. DPP Driver may also invoke a third-party SDN controller to implement northbound APIs. Besides, the system architecture is composed of supplementary building blocks: a) P4 compiler translating P4 program into a target code to be executed on a programmable switch, b) repository, where compiled P4 programs are stored, c) application plane consisting of external systems, and d) Deployment Studio providing Graphical User Interface to manage data plane modules [Osinski et al. 2019].

In fact, DPPx provides a life cycle management framework to ease the deployment of P4 programs. Although this includes some examples of applications for P4, the framework has been only evaluated in a testbed environment using the BMv2 software switch and none of these P4 applications aims to enable the support of a new network protocol or applications with similar scope as the PIaFFE framework do, using a well-defined method for steering traffic between software and hardware implementations.

2.3 PIaFFE framework

Overall, programming abstractions provided by existing packing processing frameworks are insufficient for our target domain, and possibly for other hardware-based platforms, as discussed above. Consequently, we design the PIaFFE framework to help programmers explore how to offload their server network applications to a SmartNIC, not only the network traffic, leveraging the hardware/software co-design by using P4 language and processing raw packets at the data plane, intercepting and interacting with network flows in a transparent manner.

The applications which benefit from PIaFFE have computations that may be more efficient to run on the SmartNIC than on the CPU because of the SmartNIC's hardware-accelerated functions, parallelism, or even reduced latency when eliminating some system layers from the processing path.

However, while parts of typical server applications can be built by composing predefined elements, many parts cannot. In the selected target domain, developers often want to offload an application by reusing existing application code instead of writing code from scratch. Besides porting existing applications, some developers may prefer to implement most of their applications in P4 due to their low complexity for development and domain-specific nature.

This correlation between software applications and hardware functions can be addressed by using mechanisms available in the framework, helping to steer network flows among these two levels. Thus, the data plane could be used, for example, to interact directly with network flows, without the dependency on an application at the software level, acting like the application itself.

Table 2.1 shows a comparison between the PIaFFE framework with other works cited above. In this context, is worth mentioning some definitions in the table:

- We define an **embedded network function (eNF)** as a virtual network function running inside a programmable network device, working with network-related data in the link, network and/or transport layers (e.g. firewall, ISD, NAT) or even application layer data, inspecting and/or changing its data.

- We also consider an **embedded application (eApp)** a subclass of eNF, running inside a programmable network device and interacting with raw packets at application layer data (e.g. DNS, HTTP server, MQTT), classifying and building new headers in order to manipulate network flows. Both eNF and eApp rely on the co-design of software and hardware, leveraging the best of each architecture.

- **Intercepting/processing** are related to the ability to inspect packet headers and/or data for post-processing into a specific computational logic unit inside the data plane, interacting with both network flows and "software-level" applications running at a hypervisor in a run time fashion.

- **Functional decomposition (Func. Decomp.)** remits to a method for dissecting a complex process in order to examine its individual elements, easing the development and the consequent deployment in a limited-resource device, like SmartNIC; this methodology will be detailed in Chapter 3.

Framework	**Hardware Implementation**	**eNF**	**eApp**	**Intercepting/ Processing**	**Func. Decomp.**
ClickNP (2.2.1)	yes (FPGA)	no	no	no	no
AccelNet (2.2.2)	yes (sNIC-FPGA, Azure)	no	no	no	no
Floem (2.2.3)	yes (sNIC, Cavium LiquidIO)	no	yes	no	no
$\{\lambda\}$-NIC (2.2.4)	yes (sNIC, Netronome Agilio)	no	yes	yes	no
NICA (2.2.5)	yes (sNIC-FPGA)	no	yes	no	no
iPipe (2.2.6)	yes (sNIC, Cavium LiquidIO)	no	no	no	no
FlowBlaze (2.2.7)	yes (sNIC-FPGA, SUME)	yes	no	yes	no
FairNIC (2.2.8)	yes (sNIC, Cavium LiquidIO)	no	yes	yes	no
P4Runtime (2.2.9)	yes (PSA-based)	no	no	no	no
EP4 (2.2.10)	no (bmv2)	no	yes	yes	no
DPPx (2.2.11)	no (bmv2)	yes	no	no	yes
PIaFFE	**yes (sNIC, Netronome Agilio)**	**yes**	**yes**	**yes**	**yes**

Table 2.1: PIaFFE compared with other framework projects.

2.4 Chapter remarks

This chapter discussed the foundations of the proposal, as well as the related works and their limitations, which inspired the creation of our framework, giving a brief resume of the technologies that will be addressed during the next chapters, in the same way as how our work differs from the related works cited here and its contributions.

In this Thesis, we advocate that the most appropriate expressive development and deployment methods offer a flexible embedding for offloading network workload onto a programmable network device using a domain-specific programming language, such as the P4 language. A framework becomes crucial by eliminating the extra work-out during the development phase, which covers the choice of the virtualisation technology, the “connection” between software and hardware network applications and the final deployment of these applications.

Consequently, the correct balance between software and hardware execution could be acquired using this “connector” for offloading partially or totally network flows to the programmable data plane. Moreover, security-related concerns can be raised when using such network programmability, which also can be addressed using the same devices as enablers for offloading security applications, which will be covered by the proposal.

The next chapters will present the solution based on a *Framework for embedding network programmable cards* – from the development to deployment of network applications into a real programmable network device – following their specific use-cases exposed in chapter format and its applicability to network-related applications, showing and discussing the results, which compose our proposal of the Thesis.

Chapter 3

A framework for flexible software/hardware co-design of in-network computing

3.1 Overview

Emerging in-networking processor-based programmable network interface cards (NIC) offer a solution for offloading network traffic data, using domain-specific programming languages to express how packets should be processed in order to try to alleviate the high demand for network bandwidth and CPU computing power. These in-network computing capabilities allow hosts to offload general computations onto the programmable NIC while keeping the support of complex application logic on the hosts. By offloading lightweight but frequently invoked operations, we can accelerate cloud/edge applications while reducing the host CPU load without sacrificing the program generality.

Despite the option to use these programmable network devices as an accelerator for cloud and edge devices, vendors do not define how to develop and deploy a "fully integrated environment" using these domain-specific languages and the remaining applications running at the hypervisor, enabling the co-execution of data processing with minimal or even no changes into the original application.

This chapter describes the architecture and design principles of our framework proposal: **PIaFFE – a framework for embedding network programmable cards**, using an in-networking processor on a programmable NIC to co-execute distributed workloads, aiming for an automated environment for the development and deployment of applications, which can enable the reduction of both request execution latency on the fast path (network programmable hardware data plane), as well as host CPU computation load.

Specifically, we explore the feasibility of decomposition and deploying using a domain-specific programming language, such as P4 language, executing lightweight applications (aka embedded network functions – eNF) on a NIC-side in-network processor using a new technique for steering traffic to be processed between the hardware and applications running at a Virtual Machine (VM). We also describe the architectural options raised before, delimiting the selected elements for use with cloud and edge computing. Moreover, the building blocks of the proposal are described here, raising the main aspects of the framework and how it can be adapted to tackle different targets.

The initiative opens new avenues for programmers to develop data centre applications using an in-networking processor effectively. PIaFFE framework provides a method for decomposing and deploying network applications using the P4 language into SmartNICs, in conjunction with traditional VMs at the hardware data path.

3.2 Proposal

This section describes the design principles, architecture and enablers, and the potential limitations of our approach using PIaFFE to develop and deploy micro-applications onto network programmable hardware. We also discuss the relationship between the PIaFFE framework with P4 language and programmable network devices. In finishing the chapter, we also review the relationship between the PIaFFE foundation and the End-to-End principle, positioning the proposal over the main aspect of network and Internet functionality.

3.2.1 Design principles

According to [Johnson 1997], a framework is a larger-scale design that describes how a program is decomposed into a set of interacting objects, reusing the main program and delivering to the developer the decision of what is plugged into it and making new components that are plugged in.

Based on the framework definition cited early, the design principles of our proposed framework were motivated by the challenge of deploying micro-applications on resource-limited network hardware, keeping their expressiveness and functionality even running on these restricted devices, besides using high-level definitions to specify the overall project from software to hardware levels, as shown in Figure 3.1.

Following the Figure 3.1, PIaFFE architecture is comprised into two main levels:

- **Project definition**:
  This level is responsible for the software-related constraints, enumerated by their respective processes as follows:

  (1.1) **Virtualisation specifications**, or the type of virtualisation that can be defined at the start of the project (e.g. Linux libvirt, Linux namespace, Docker, etc.), which will be deployed during the experiment lifecycle.

  (1.2) **Functional decomposition**, permitting the application to be broken down into its most simplistic components and functions for rapid application development. As functional decomposition is a broad topic, we will discuss it in more detail further in the text.

  (1.3) **Application description** is one of the steps in this level, characterising the applications which will be used in the project, including the kind of network application (e.g. a layer 3/4 firewall, a network load-balancer, a cryptographic network function, etc), the main source-code of the prototype software and its relation with the programmable network device.

(1.4) **P4 table entries** for matching with the specific use case, enabling the communication between software and hardware levels and also through the application and programmable data plane.

(1.5) **Monitoring** is another characteristic implemented using PIaFFE, in order to collect data for providing visibility on the framework, which can be defined or implemented using a third-party tool such as gRPC or even using In-band Network Telemetry (INT).

(1.6) **Network topology** also needs to be defined in order to create the testbed used during the experiment, including the network nodes (physical or virtual ones) connected through the SmartNIC.

(1.7) **connection between software and hardware levels**, the "path" used to forward packets throughout these two levels according to the definitions and process listed above.

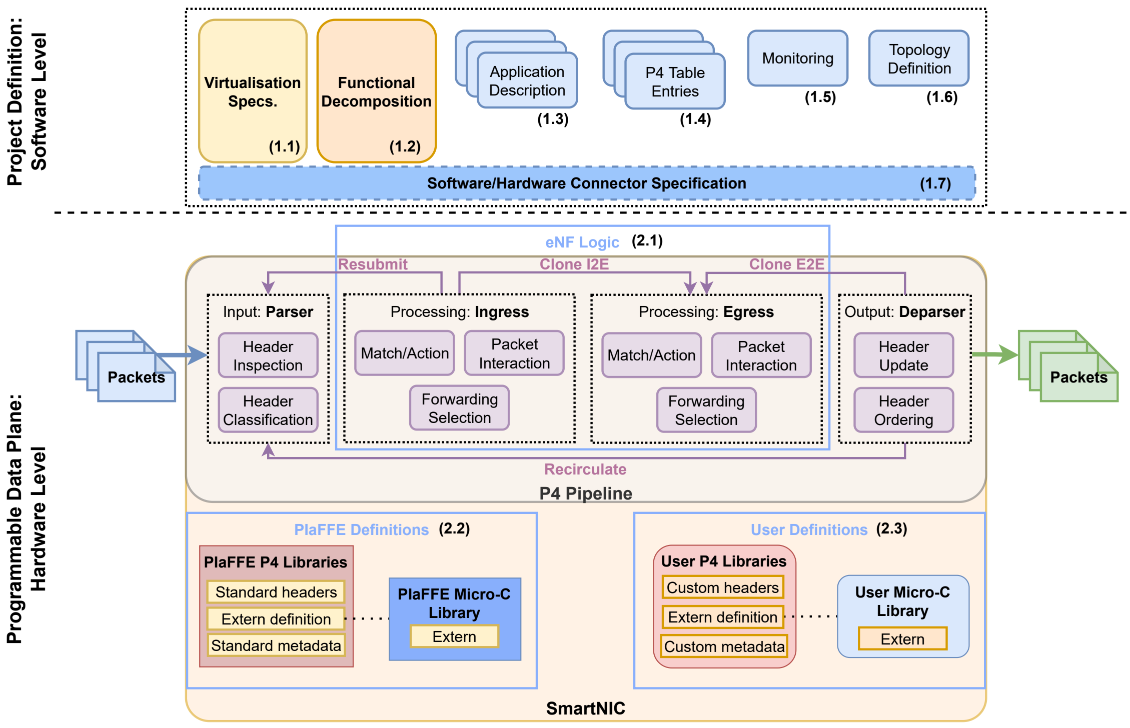


Figure 3.1: PIaFFE framework architecture.

- **Programmable data plane**:
  Despite the "Software Level" development process, the PIaFFE framework defines the steps enabling the programmability into the SmartNIC, creating the three main building blocks during the development and after the deployment:

(2.1) **Embedded Network Function (eNF) logic**, which is the main P4 program embedded into the hardware, constructed over the P4 target pipeline and strictly linked with its software counterpart application, thus, both software and hardware applications have a tightly integrated development process, as one complements the other in terms of functionality and usability.

(2.2) **PIaFFE definitions**, based on a collection of pre-built P4 and micro-c libraries with standard protocol headers (e.g. Ethernet, IPv4, Ipv6, TCP, UDP, etc.), already shipped with the PIaFFE framework.

(2.3) **User definitions**, where the users can freely create and define new libraries and *externs*, custom headers and metadata, according to the application requirements and constraints.

To address the early-mentioned challenges, as well as the framework definitions and its processes cited above, we design the solution based on the following enablers:

- **Introduction for functional decomposition**:
  Most network functions are deployed as a single monolith, whereby all components are bundled together into one entity. This can lead to maintenance challenges, as well as slow down the trailing of new technologies. Decomposition is therefore a significant step in evolving VNFs to be cloud-native and much more agile and scalable. It will be undertaken for the majority of the data plane, control and signalling VNFs. In this context, we define decomposition as breaking a monolithic network function into a set of small applications, or "micro-applications".

  Decomposition also allows for common functions to be stripped away from the core logic of the applications. This allows the applications themselves to be more lightweight – which makes them easier and quicker to develop, manage, and deploy into programmable network hardware – as well as centralise core functions and operate in an "as-a-service" model. We also want to be able to reuse common functionality, but not have to pay for it multiple times. The process of decomposition also gives micro-application owners the opportunity to remove unnecessary functionality from the application logic [Richardson 2018].

  However, the main challenge in realising any micro-application-based software using decomposition is to identify the set of functionalities, and this is difficult because it requires domain-specific knowledge. One way to approach decomposition is to leverage domain expertise (e.g., by consulting with domain-specific developers) or to study existing open-source network applications, identifying smaller functional units.

  As a key to employing functional decomposition, the PIaFFE framework defines a roadmap to create an embedded Network Function (eNF) using an approach based on the application network-related characteristics (e.g., packet parsing, classifying, processing and forwarding), re-architecting the network programming ecosystem, in order to make a feasible and practical placement of these micro-applications into an in-network programmable device.

For example, using for reference two monolithic network applications, the first one based on an L2/L3 packet forwarder and the second one based on the SHA-256 algorithm, we can outline the steps for decomposing these two applications into a new micro-application, as shown in Figure 3.2. The first step for decomposing multiple applications is to identify the network-related characteristics of each one, in this case, we can enumerate the primary operations, starting with the **L2/L3 Forwarder**:

i. parse headers;
ii. L2 header classification;
iii. L3 header classification;
iv. L2/L3 forwarder logic
v. forwarding table;
vi. drop action;
vii. the forwarding action as it is.

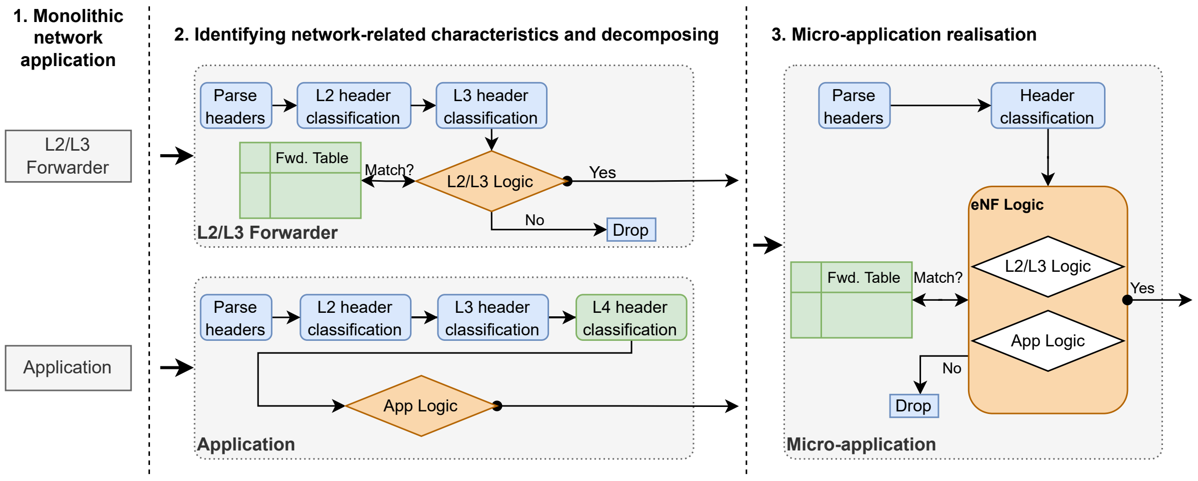


Figure 3.2: Illustrative example of functional decomposition using a set of network applications.

In the same way, we enumerate the characteristics of the next "generic application" to be developed and deployed, using more stages for header classification just for illustrating the overall process:

viii. parse headers;
ix. L2 header classification;
x. L3 header classification;
xi. L4 header classification;
xii. application logic.

As we can see, Figure 3.2 uses the same `blue` colour to highlight the overlapped characteristics and `green` colour for specific functions. We also standardise the orange colour pattern to emphasise the main logic behind both applications. As illustrated by the colour map, we can merge the "duplicated" functions of the applications into the **micro-application realisation**, using a single structure representing the two main applications, making it possible to be embedded in programmable network hardware and customising the data plane behaviour.

- **Multi-level forwarding between hardware and software**:
  PIaFFE framework implements the PIaFFE Intercepting and Forwarding element (PIIF) to steer network traffic through the embedded Network Function or send it up to the application at the virtualisation layer, using a bottom-up approach as shown in Figure 3.3. As long as the network traffic arrives at the SmartNIC, it is intercepted and forwarded to the OS level, or kept at the hardware level, and processed using the SmartNIC network hardware.

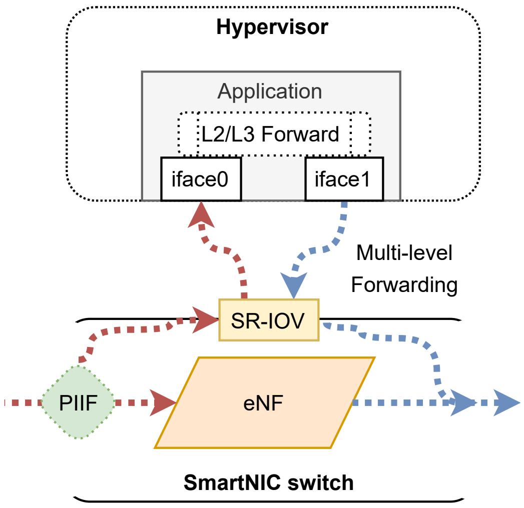


Figure 3.3: Multi-level forwarding using PIaFFE Intercepting and Forwarding (PIIF) element.

The multi-level forwarding can be used for allowing to determine the amount of the network traffic which will be offloaded via hardware or processed into the software level. It can be implemented via:

i **hash table**: a fundamental data structure in network applications, including route lookup, packet classification and monitoring [Song et al. 2005], providing time- and space-saving data structure for packet lookup, which is very useful when ported to a resource-restricted data plane like SmartNICs;

ii **bloom filter**: a well-known randomised probabilistic hashing data structure that answers set membership queries, which can provide fast detection of heavy flows [Tarkoma et al. 2012], for example, allowing the correct balance between hardware and software packet processing by steering selected traffic between them;

iii **external trigger**: which can be implemented via an operating system call using gRPC or P4 table interaction, running aside of the data plane and reacting according to a specific system condition (e.g. external sensor, CPU or memory usage) or a manual intervention by the user.

In Listing 3.1, it is presented a fragment of the P4 source code illustrating the PIIF element. This code defines a basic $(key, value)$ data structure. Whenever an incoming packet hits a table's entry (by its source IPv4 and UDP port), then it is steered to the software, otherwise, it is sent to the eNF pipeline that applies a specific function. This approach fits with the **external trigger** option, as an application could interact with traffic by changing the P4 table.

```
  ...
  @name(".piif_table")
  table piif {
    key = {
      standard_metadata.ingress_port: exact;
      hdr.ipv4.srcAddr: exact;
      hdr.udp.srcPort: exact;
    }
    actions = {
      NoAction;
    }
    size = 512;
    default_action = NoAction;
  }
  ...
  apply {
    if (piif.apply().hit) {
      sw_forward.apply(); // network traffic to software
    } else {
      do_micro_application(); // executes an application at hardware
      hw_forward.apply(); // forward traffic via hardware
    }
  }

```

Listing 3.1: P4 source code of the PIIF

Splitting the functionality of network applications into two levels could raise new constraints related to keeping state and consistency between hardware and software. For example, a network function may need to use a counter to register some kind of traffic (i.e. heavy-hitter detector), but as defined previously, we have the same network function in both levels. In this specific case, we can delegate the task to the hardware level, sharing the data with the software level. However, we need to weigh each use case separately, as the hardware data plane could be a bottleneck if another kind of data structure overflows its capacity.

3.3 Architecture design and enablers

This section describes the hardware and techniques employed to build the PIaFFE framework. Also, it provides some guidelines on designing the P4 code and metadata structures that will be embedded into a SmartNIC and how the multi-level forwarding can be used to steer the network traffic between software and hardware.

According to Figure 3.4, the PIaFFE framework can be used to implement a set of applications using two main architectures as the basis:

i. **x86 servers**, to development and deployment of traditional VNFs;

ii. **SmartNICs**, for the deployment of eNFs for offloading network traffic and computation.

Computational resources are inversely proportional to the packet processing performance, due to the generic processing and operational system stacks on general-purpose processors (x86 servers). Moreover, SmartNICs are faster than x86 servers when running basic operations in network packets (e.g. forwarding, small processing), but it loses hardware resources and overall computational power. It is natural a merge these two platforms, each one complementing their constraints and improving the network packet processing performance using in-network computing to alleviate the network packet processing tasks.

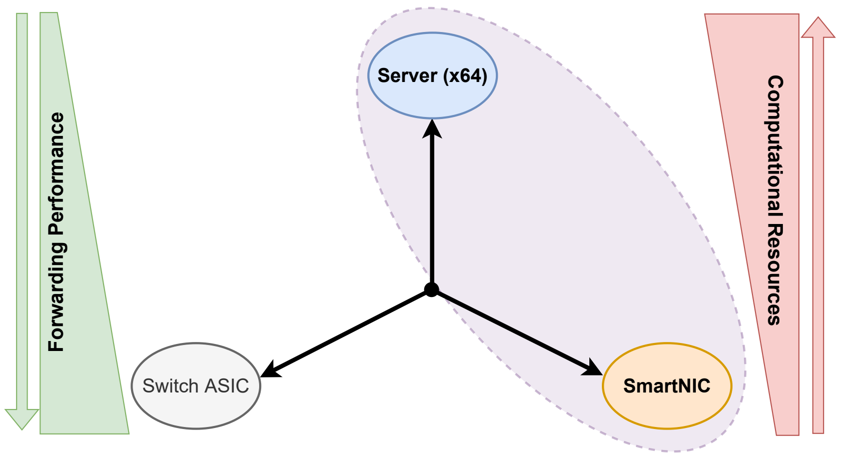


Figure 3.4: Architectural options for PIaFFE.

From an engineering perspective, the PIaFFE framework has been built over traditional x86 Intel-based servers and the Netronome Agilio SmartNIC SDK. We take advantage of the P4 toolchain (compilers, linker and loader) provided by Netronome Network Flow Processor SDK to build and burn the customised firmware generated using the framework. It is worth mentioning that the framework was designed in a modular fashion, and can be easily adapted to other SDKs that follow the same logic as the one used, as we will discuss better during this Section.

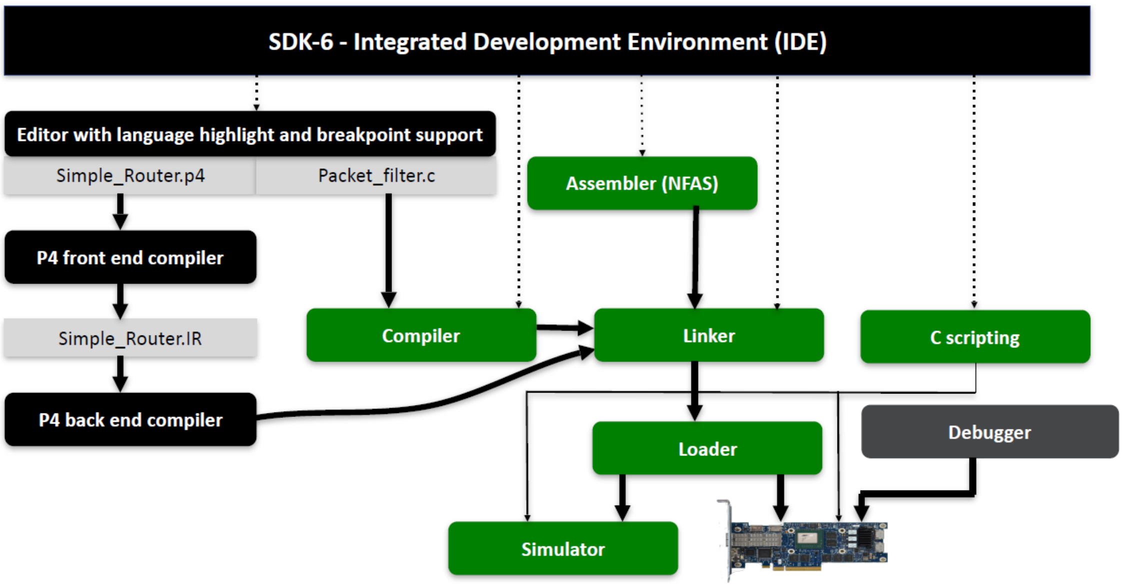


Figure 3.5: Netronome P4 SDK architecture.

The building blocks are described in Figure 3.6, where PIaFFE starts by using the **Functional decomposition** to break down the functional relationship of complex elements into smaller units, easing the comprehension of each task inside a network function and enabling the reconstruction of those parts into an embedded network function (eNF) by using P4 or micro-C languages.

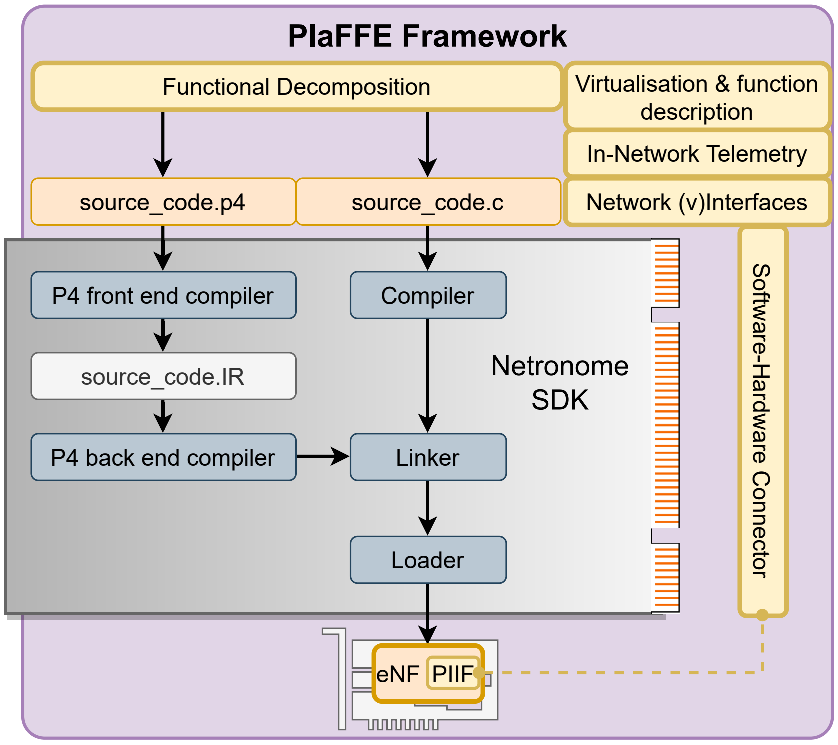


Figure 3.6: PIaFFE framework building blocks.

In parallel, we have the **Virtualisation and function description**, crucial for defining the kind of virtualisation that will be utilised during the experiment, as well as the network function capabilities and definitions, according to their network-related actions. **In-Network Telemetry (INT)** is another part of the process, used to describe the fields to be acquired during the experiment for measurements, as well as the headers which will be inserted into the network packets.

Simultaneousness, the **Network (virtual) Interfaces** will be explicit, in order to expose the hardware capabilities to the hypervisor and the network functions running inside it. Furthermore, it will be an important component of the next component, enabling the communication between the eNF and the application through the **PIIF** element.

Following the last depiction, the **software-hardware connector** is responsible for attaching the high-level application to its hardware counterpart, redirecting (offloading) the network traffic using a *bottom-up approach*: the traffic comes from the SmartNIC hardware and can be forwarded to the software or just kept in the hardware, following the **PIIF** element instruction set.

As we described before, the PIaFFE framework was conceived to be coupled into the Netronome SDK, reusing specific elements of the platform, as shown in Figure 3.5. Following the Netronome SDK, we can expand PIaFFE's building blocks from the main PIaFFE architecture design in Figure 3.1, addressing the leading points connected to the original SDK, highlighted in Figure 3.6.

Hence, the only portion of the Netronome SDK which is kept is comprised of the compilers, linker and loader binaries – the toolchain responsible for building the firmware to the desired target architecture – and all other parts are stripped out to make room for the PIaFFE framework modules, each one with its respective function, as described in the previous section. Again, is important to note that PIaFFE can be adapted to any programmable interface card SDK, as long as they employ a similar structure (e.g. compilers, linker and loader elements).

3.3.1 Network programmability levels

PIaFFE framework employs a software development approach based on network programmability levels, starting at the hardware level using P4 programming language, which can be leveraged by micro-C libraries and functions – pre-built or user-defined – and, finally, using an application running on the software level in a general-purpose processor (x86).

1. **P4 language for network programming**:
   At this level, PIaFFE provides the main P4 source-code structure, using pre-designed libraries and routines, to enable programmability via an in-network programming device based on SmartNICs. The complexity for creating applications of this level is low when compared with the micro-C level, however, PIaFFE using P4 can express sophisticated, hardware-independent packet processing algorithms using solely general-purpose operations and table look-ups. Such programs are portable across hardware targets that implement the same architectures and extensive to different architectures using the next software development level – micro-C *externs* – which will be detailed below.

2. **Micro-C *externs* as a complement**:
   PIaFFE employs *externs* implementation in the P4 language via micro-C (a C-like language), extending the features not supported in the P4 language, such as loop control and hardware timestamp support. Despite the complexity imposed by using this level of development being high compared with the P4 language, we can advance the programmability using in-network programming devices, unlocking some constraints imposed by the P4 language design, and exploring the specific resources of each target, enabling the optimisation of the solution to be implemented.

3. **Network software application**:
   At the highest level of the stack, PIaFFE defines the connectors between the application running at x86 general-purpose processors and the embedded network function running into the programmable network card. Thus, at this level, we can define the complex applications which need more intensive computation to be completed, such as machine learning, computational vision and other compute-intensive tasks.

3.4 PIaFFE automated deployment

In software engineering, software deployment may be defined to be the process between the acquisition and execution of software. This process is performed by a *software deployer* who is the agent that acquires the software, prepares it for execution, and possibly executes the software. Software deployment may be considered to be a process consisting of a number of inter-related activities including the **release** of software at the end of the development cycle; the **configuration** of the software, the **installation** of software into the execution environment and the **activation** of the software [Dearle 2007].

PIaFFE framework also simplifies the development and deployment process of the application, as shown in Figure 3.7, covering all activities mentioned before, both at the software and hardware levels. Consequently, all stages are "protected" and guided by the framework using the predefined descriptions set during the **Application description** and **Virtualisation specifications** phases in the development process, fostering the automated deployment of the desired execution environment. Hence, the embedding of the network application occurs in a transparent manner for the developer that aims to program a data plane for offloading applications running at a specific virtualisation technique.

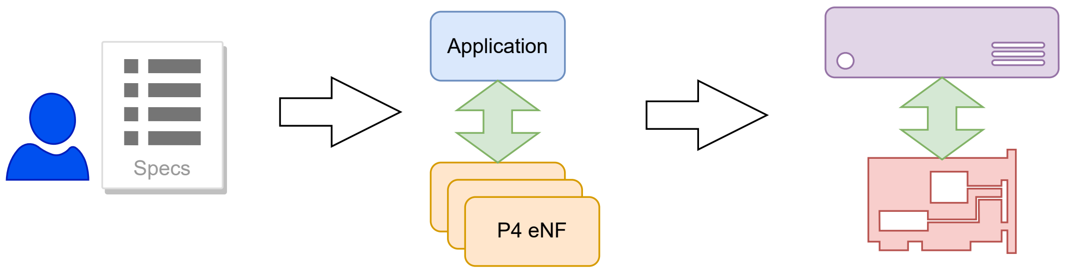


Figure 3.7: Resumed steps for embedding applications into a programmable network device using the PIaFFE framework.

The deployment stage needs to be “protected” once the firmware burning process changes the hardware characteristics of the SmartNIC (e.g. the number of available virtual functions), leading to a total release/renewal of resources allocated for the network programmable hardware. When these resources are directly attached to a Virtual Machine (VM) and this release of resources happens, the operating system tends to crash, due to a lack of communication between the firmware’s loader and the hypervisor.

To illustrate the process we can cite an example, when a developer needs to deploy an application using the software level, in parallel with an embedded network function (eNF) running at the SmartNIC hardware, we need to take some precautions:

- The application (or the OS) running at the hypervisor needs to be turned off **before** the firmware deployment at the SmartNIC, due to possible access violation issues when using a pass-through technique, such as SR-IOV, to expose the network hardware to the OS running on the virtual machine.
- All traffic flowing through the SmartNIC must be stopped in the same way, in order to guarantee the system integrity during the deployment process.
- All instantiated virtual machines need to be stopped as well, in order to release the virtual functions allocated in the hypervisor to the original operating system.
- Besides, after a successful deployment, the environment needs to be started again, from the virtual hosts to the new eNF built into the SmartNIC, enabling the testbed for experimentation.

All these steps cited above are fully covered by the deployment process of the PIaFFE framework: from checking the successful compilation and link process to the interruption of all resources allocated to the experiment, avoiding user interaction – and possible errors and disruptions – during the development and experimentation flow cycle, leveraging the general operation for prototyping an environment that requires a programmable data plane operating in conjunction with an application in an automated fashion.

3.5 Chapter remarks

In this chapter, we propose the PIaFFE Framework focusing on a place-as-you-go approach that exploits the flexibility of embedding applications in programmable SmartNICs. The framework provides a guide to programmers to decompose and deploy VNF applications to an eNF using a NIC-side in-networking processor with traditional applications running at a commodity server.

Thus, key aspects of the practical use of programmability and offloading constraints were addressed in PIaFFE Framework by creating programming abstractions as building blocks using the P4 language and proposing novel multi-level offloading through eNFs in SmartNICs.

However, we still need to validate the framework, offloading traffic processing using consolidated virtualisation technology, such as Network Function Virtualisation (NFV), and improving the multi-level forwarding element to tackle both levels – software and hardware – simultaneously. Thus, the PIaFFE framework can be employed to deliver the building blocks to architect a fully-compliant offloading system for use with multiple virtualisation techniques, outlining latency improvements and saving CPU resources, as part of processing can be done via in-network computing.

In the next chapters, we will explore multiple domains: starting with a wide-range NFV paradigm, then going into specific network applications, and finally diving into the network layers and passive optical networks, in order to validate and evolve the proposed framework, using the PIaFFE framework for prototyping, compiling, burning and running the environment for verification and analysis, showing the feasibility of the proposal for embedding programmable network cards.

Chapter 4

A co-design use-case on Network Function Virtualisation (NFV)

4.1 Overview

Network operators ubiquitously deploy hardware middle-boxes (e.g., NATs, Firewalls, WAN Optimisers etc.) to implement a whole range of different network services [Sherry et al. 2012]. Despite being an integral part of modern enterprise and telecommunication networks, middle-boxes are usually proprietary and have little or no programmability. They are also hard to vertically integrate with other packet processing elements. Such a closed and inflexible ecosystem explains in part the high capital and operational expenditures incurred by network operators. This led to the Network Function Virtualisation (NFV) movement initiated in 2012 [Canada et al. 2017].

NFV proposes to disaggregate the tightly coupled Network Functions (NFs) and hardware middle-boxes and deploy the NFs as Virtual Network Functions (VNFs) on commodity servers. Through this disaggregation, NFV promises to reduce CAPEX by consolidating multiple NFs on the same hardware and reduce OPEX by enabling on-demand flexible service provisioning.

In-network computing can leverage the NFV environment by offloading lightweight but frequently invoked operations onto a programmable network device, accelerating data centre applications while reducing the host CPU load without sacrificing the program generality. However, vendors do not define how to deploy a fully integrated environment using these domain-specific programming languages using the NFV paradigm.

In this chapter, we will expose the benefit of using the PIaFFE framework proposal, exploring the use-case of an in-networking processor on a programmable NIC to co-execute data centre workloads employing NFV, so that one can reduce both request execution latency on the fast path, as well as host CPU computation load. Specifically, we explore the feasibility of VNFs decomposition and deploying utilising a domain-specific programming language, such as P4 language, executing a lightweight VNFs (aka embedded network functions – eNF) on a NIC-side in-network processor that can process and/or redirect traffic to traditional VNFs running at a Virtual Machine (VM), employing a multi-level Service Function Chaining (SFC).

However, this intuitive scenario is not trivial to be implemented considering that the following points should be addressed:

i. a proper mechanism for exchanging network traffic between VNFs and eNFs must be found (i.e. service function chaining);

ii. a compact virtual network function using a domain-specific programming language that fits into the SmartNIC hardware must be built;

iii. the correct balance of network traffic that can be offloaded without degrading the overall performance must be found.

We evolve the initial implementation of the PIaFFE framework for addressing the above challenges. This initiative aims to open new avenues for programmers to develop edge VNF applications using an in-networking processor. PIaFFE provides a method for decomposing and deploying VNFs mainly using the P4 language into SmartNICs, implementing a multi-level chaining technique that allows the partial or full VNF traffic offloading into programmable network hardware.

4.2 NFV use-case

This section describes the design principles, architecture and enablers, and the potential limitations of our approach using PIaFFE to develop and deploy Virtual Network Functions onto network programmable hardware. Here, PIaFFE will tackle the Network Function Virtualisation scenario as a use case, positioning it into the ETSI reference architecture and enabling the deployment of embedded network functions using multi-level chaining for VNFs and eNFs.

4.2.1 Experimentation design

For the sake of demonstration, we choose a use-case motivated by the challenge of deploying network functions on resource-limited network hardware, when high-level VNFs remain running on the top-level applications, sharing resources between both. Thus, this chapter includes the implementation of VNFs using PIaFFE, aiming at the data link, network and transport layers, as depicted in Figure 4.1. Hence, PIaFFE will tackle these network layers using the NFV paradigm allied with in-network computing for offloading processing using a SmartNIC.

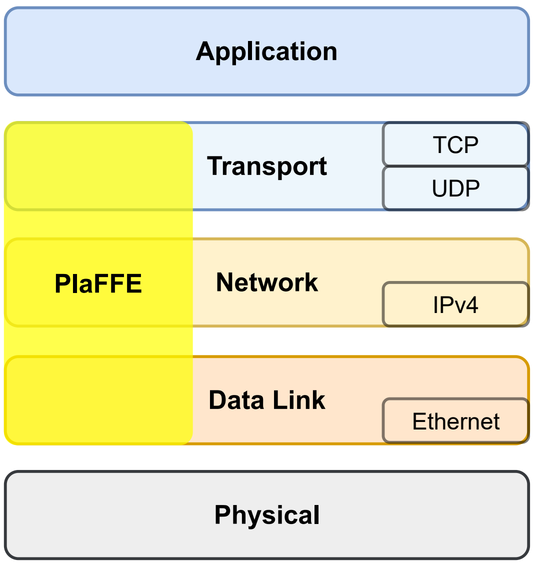


Figure 4.1: Network layers handled by PIaFFE in this Chapter.

To address the early-mentioned challenges, we design the solution based on the following enablers:

- **Functional decomposition in NFV**: PIaFFE employs a functional decomposition of multiple VNFs, re-architecting the network programming ecosystem [Rahman Chowdhury et al. 2019], in order to make a feasible and practical placement of these VNFs into an in-network programmable device. For example, let us consider an infrastructure that has to place a set of network functions such as a simple authentication/authorisation application, a firewall and an Intrusion Detection System (IDS), as shown in Figure 4.2. Each part of a VNF has its own role in a monolithic implementation, and developers will need to implement these parts separately, on their respective VNFs. Therefore, when a packet goes through the VNFs in an SFC, this would require more CPU cycles due to the repeated execution of the same functionality (e.g. parse headers and header classification).

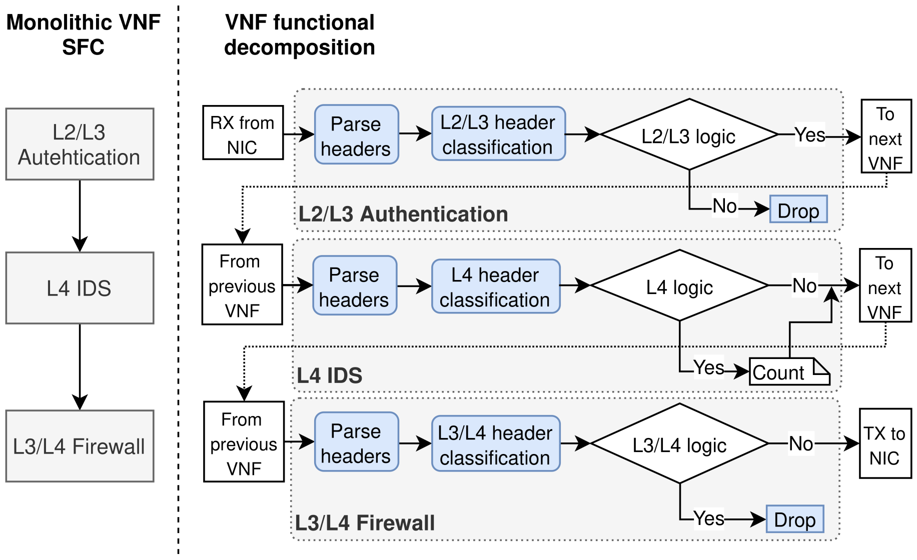


Figure 4.2: Functional decomposition of monolithic VNFs, inspired by [Rahman Chowdhury et al. 2019].

- **Network application deployment using P4 language**: Decomposing a network application brings us a clear vision of the parts that have overlapped functions. For instance, to receive a packet from NIC, parse packet header and classification, and forward to NIC or to next VNF, as depicted in Figure 4.2. Once we have identified the common parts, the next step is to merge all of them into a new clean layer which will enable the creation of an eNF to be placed into a SmartNIC. The last step consists of mapping the common functions into *modular eNFs* so that the PIaFFE framework may create their respective P4 primitives (such as parse headers, header classification, drop and count actions) and logic (eNF Logic), as shown in Figure 4.3.

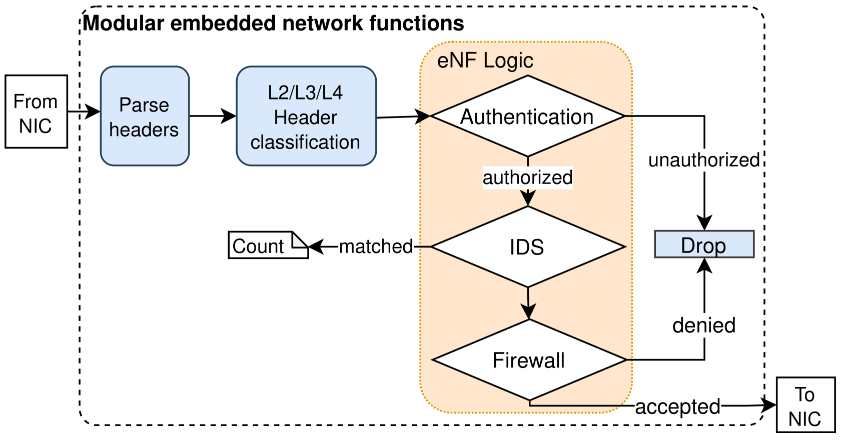


Figure 4.3: Modular eNFs and P4 logic mapping for network functionalities.

- **Multi-level chaining**: PIaFFE implements multi-level chaining for VNFs and eNFs using an Offload Balancer to steer traffic through the eNF_i at the hardware level or send it up to the VNF_i at the virtualisation layer, as shown in Figure 4.4.

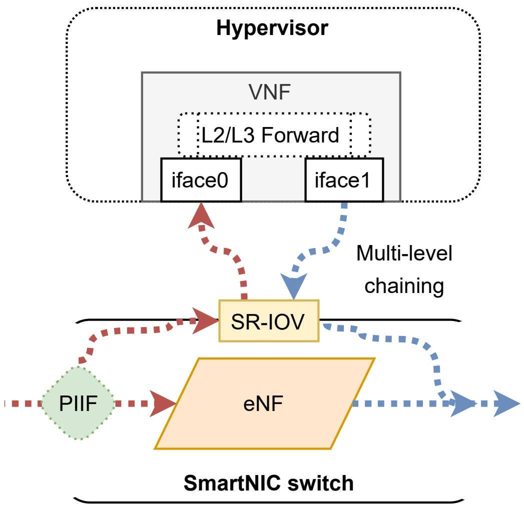


Figure 4.4: Multi-level Chaining of Network Functions.

The Offload Balancer can be defined using three modes: 1) **internal**, based on packet interception and classification (e.g. load balancer, classification policy); 2) **external**: based in an external event (e.g. trigger, command line); and 3) **dynamic**: based on probabilistic data structures or queue policies (e.g. bloom filters, hash tables, round robin).

In this use-case, the Offload Balancer was implemented via **internal mode**, using a load balancer to select the path of the network flow, as illustrated on the pseudo-code 4.1. Thus, when certain condition matches, the network traffic is managed by the SmartNIC switch, processing the network traffic data into its own hardware; otherwise, the traffic is steered to the VNF at the software level, where it will be processed and then forwarded to the next function, according to the specific VNF purpose (e.g. if a firewall blocks the particular flow, it will be stopped on the current position of the function chaining). We have made use of the AND bitwise operator – which is more computationally efficient than executing a traditional modulo operation – for defining the percentage of traffic which will be processed through the eNF or the VNF. It is worth noting that the AND bitwise operator can replace the traditional modulo operation only when x is a positive integer and power of two; also, this is a constraint in Netronome SmartNIC hardware implementation, which can only handle the modulo operation when the same premise is true, as represented by the following equation:

$$x \,\%\, 2^n \;==\; x \,\&\, (2^n \;-\; 1) \tag{4.1}$$

4.2.1.1 Source-code fragment for the Offload Balancer

In Listing 4.1, it is presented a P4 source code fragment of the Offload Balancer element. This simple code redefines a $(key, value)$ data structure from the PIIF element examined previously in Chapter 3, now adapted to be used with a VNF. The underlying logic remains the same: whenever an incoming packet satisfies the statement – here based on a bitwise operation result – then it is steered to the

VNF, otherwise, it is sent to the pipeline eNF that applies a specific function. We choose the bitwise operation because it generates less overhead to the hardware, as the calculation was done using fewer cycles when compared with the division reminder and others, enabling the offloading using a percentage fashion, according to the result of the operation. Thus, the main difference here is that we can use a load balancer based on the bitwise operation, in order to distribute network or application traffic across the hardware and/or software, increasing the capacity and reliability of VNFs and eNFs.

```
...
@name(".firewall")
  table firewall {
    key = {
      hdr.ipv4.srcAddr: exact;
      hdr.ipv4.dstAddr: lpm;
      hdr.tcp.dstPort: exact;
    }
    actions = {
      // L3 firewall action provided by PIaFFE
      do_firewall;
      drop;
      NoAction;
    }
    size = 512;
    default_action = NoAction;
  }
...
apply {
  // traffic steering based on bitwise operation
  // the operation will split the traffic on a 50% scale (x % 2 == x & 1)
  if ((hdr.ipv4.identification & 1) == 0) {
    // network traffic goes to VNF
    firewall_vnf.apply();
  } else {
    // offload to eNF
    firewall_enf.apply();
  }
}
```

Listing 4.1: P4 source code example of an Offload Balancer

4.2.2 Usage of the framework

This section describes the PIaFFE architecture enablers, starting with its position on ETSI NFV reference architecture, the hardware description and techniques employed to build the PIaFFE framework onto the NFV paradigm. Also, it provides some guidelines on designing the P4 code and metadata structures that will be embedded into a SmartNIC, and how the multi-level chaining can be used to steer the network traffic between VNFs and eNFs.

Figure 4.5 shows the position of PIaFFE in ETSI NFV reference architecture. The framework defines the functional blocks and the reference points needed to support the infrastructure services in the operator's network. Within NFV, the infrastructure services are referred to as network services, which are provided by the NFs.

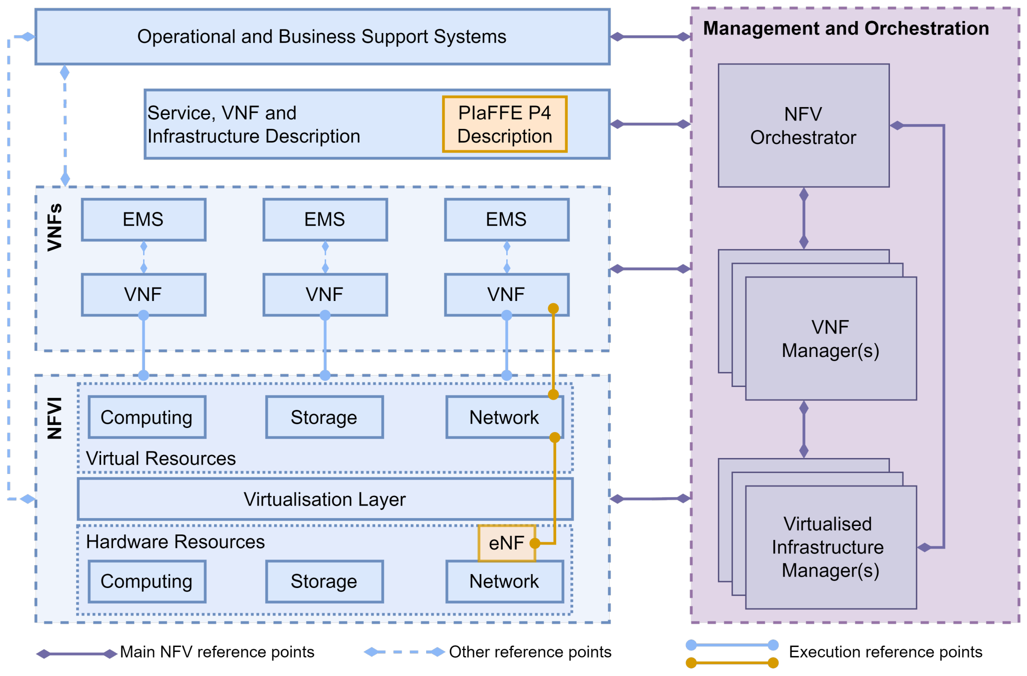

Operational and Business Support Systems
Management and Orchestration
Service, VNF and Infrastructure Description
PIaFFE P4 Description
NFV Orchestrator
VNFs
EMS
VNF
VNF Manager(s)
NFVI
Computing
Storage
Network
Virtual Resources
Virtualisation Layer
Hardware Resources
eNF
Virtualised Infrastructure Manager(s)
Main NFV reference points
Other reference points
Execution reference points

Figure 4.5: PIaFFE outlined onto ETSI NFV reference architecture framework [Canada et al. 2017]

To show compliance, we map the components of PIaFFE to blocks in the framework, starting with the Service, VNF and Infrastructure Description, where the P4 application will be described, based on its capacities and available resources. Each VNF is an implementation of a network application running atop the Network Function Virtualisation Infrastructure (NFVI) resource; PIaFFE assigns an eNF directly to the hardware resource based on a programmable network device, in this case, a SmartNIC, which is mapped to its VNF pair through the Virtualisation Layer and the Virtual Resources, exposing a virtual function of the SmartNIC to the VNF, which will be used as a communication channel to exchange data and offloading processing to the hardware. Data plane reconfiguration was not addressed by the work, as the proposal is to enable offloading workload from an NFV to an eNF, and it is out of the scope of the current work.

The PIaFFE framework prototype has been built on the Netronome Agilio SmartNIC [NETRONOME 2019]. As discussed in Chapter 3, we take advantage of the compilers, builder and linker provided by Netronome Network Flow Processor SDK. The process from compilation to firmware burning into the SmartNIC is fully controlled by PIaFFE, in the same way as loading the user configuration. After firmware had been generated, it is loaded to be executed in the SmartNIC, then the user configuration is enabled on the SmartNIC, starting the network environment using the desired virtualisation technology.

4.3 Experimental evaluation

Our evaluation aims to answer the following questions:

1. Compared with host-side execution, what are the latency savings of running parts of the applications on an in-network processor?

2. How much throughput can an application achieve using multi-level chaining?

4.3.1 Experimentation setup

For the benchmark tests, we used the environment described in section 4.2 and illustrated in Figure 4.6. The test consists of generating traffic that goes through a sequence of multi-level function chaining and measuring latency and throughput performance. As a proof-of-concept, we selected the following chaining sequence: pAuth $\rightarrow$ pIDS $\rightarrow$ pFW or vFW.

In this way, the multi-level chaining is deployed only for the firewall application, which consists of an iptables-based firewall at VNF and a similar implementation at the eNF level based on a table using match/drop actions, both with a few dozen pre-inserted rules. This setup allows isolating the effects of offloading a single VNF.

UDP packets are generated at line rate (10 Gbps) using pktgen-dpdk[1] with the built-in Lua[2] script library for throughput and latency measurements, which automates the experiment and data collection. In our experiments, the traffic generator follows the RFC2544[3] specification for frame sizes to be used on Ethernet, starting by 64-byte and ending by 1518-byte, besides the methodology using a tester and a DUT (device under test) in a closed loop.

To evaluate the impact of partial or full offloading of the firewall application, we divided the experiment by varying load between eNF and VNF. In order to filter similarities and simplify the results, we decided to synthesise the experiment into four main types:

1. 100%*@VNF*: all traffic is processed by VNF;

2. 50%*@eNF*: half of the traffic is processed by eNF, and the other half is processed by VNF;

3. 90%*@eNF*: 90% of traffic is processed by eNF, and 10% is processed by VNF; and

4. 100%*@eNF*: all traffic is processed by eNF.

4.3.2 Testbed specification

Our testbed consists of two machines connected back-to-back without any switch, as shown in Figure 4.6. One of them hosts the traffic generator, while the other hosts the PIaFFE prototype plus SmartNIC. Each machine is equipped with a 1x6-core Intel Xeon E5-2620 v3 2.4GHz CPU and 2 threads per core (hyper-threading enabled), 16GB memory, and a DPDK-compatible Intel X710-2 10G Ethernet adapter on the traffic generator side.

[1]http://git.dpdk.org/apps/pktgen-dpdk/
[2]https://www.lua.org/
[3]https://www.ietf.org/rfc/rfc2544.txt

4.3.2.1 Multi-level chaining for network functions

In the right side of Figure 4.6, the chaining of three Network Functions is deployed as PIaFFE proof-of-concept. They will be used throughout the chapter for different evaluation scenarios: (1) *Authentication*: check if MAC/IP address is registered into the system accounting management. (2) *Intrusion Detection System (IDS)*: takes account of packets based on a table tuple match (e.g. count packets from defined source IP address, or based on destination TCP/UDP port). (3) *Application Firewall*: block traffic based on IDS accounting or a proactive rule.

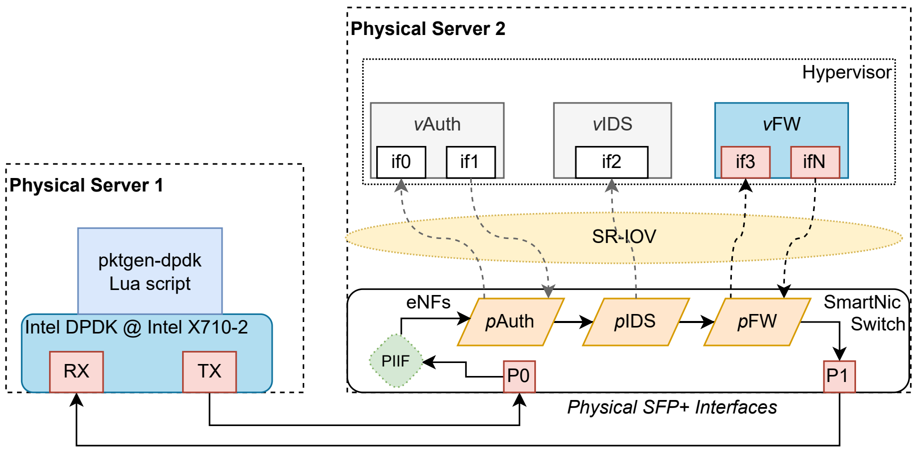


Figure 4.6: Prototype implementation and testbed.

4.3.2.2 Traffic generator

Physical Server 1 runs Intel pktgen-dpdk version 19.10 application to generate traffic at line rate – in this case, 10 Gbit/s, following the limit of the SmartNIC physical interface, while **Physical Server 2** acts as an edge data centre and uses the PIaFFE framework to process packets using in-network eNF and traditional VNFs, like in Figure 4.6. Each VNF runs on their respective qemu-kvm VM and has a dedicated virtual function from SmartNIC, provided by SR-IOV pass-through on the hypervisor, with 4GB RAM and 2 CPU cores unpinned, in order to create a more realistic environment as possible. As proof of concept, we deploy VNFs using Linux netmap and Linux OS tools (e.g. the application firewall VNF uses the iptables tool to implement this function). The physical servers run Ubuntu Linux Server 18.04, with kernel version 4.15.0-60, as same as the OS running on VMs that are hosting each VNF.

4.3.3 Latency results

Figure 4.7 shows that partial or full offloading of a VNF to an eNF reduces latency for all the test scenarios. Particularly, the reduction in the end-to-end latency is more significant with small packet sizes (< 1280 bytes). For instance, the difference reaches approximately 76x when we compare the 100%@VNF scenario ($\approx 3300 \mu s$) with the 100%@eNF scenario ($\approx 43 \mu s$) for packet sizes of 64 and 128 Bytes.

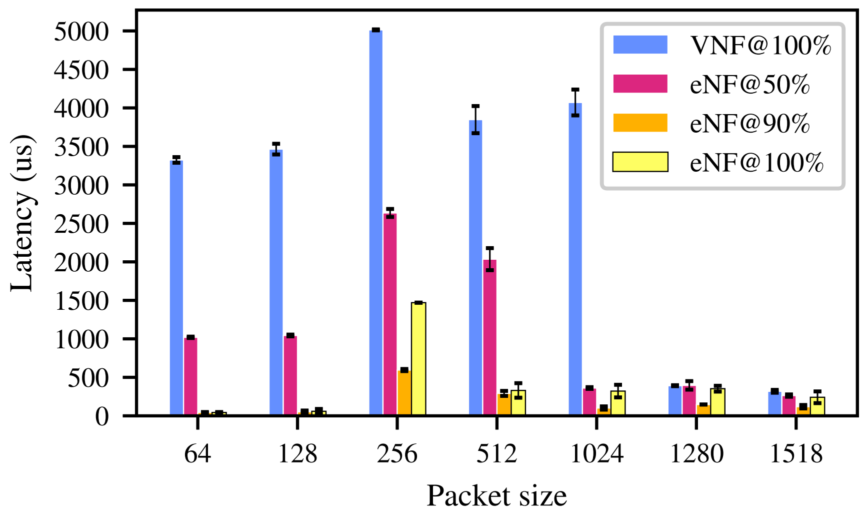


Figure 4.7: Latency comparison for different scenario types.

For big packets, partial offloading also causes latency reduction. For 1280-Byte packets, the latency decreases approximately 62% when we compare the 100%@VNF scenario with the 90%@eNF scenario. For 1518-Byte packets, the latency decreases approximately 63% when we compare the 100%@VNF scenario with the 90%@eNF scenario.

However, when the packet sizes increase, the full offloading to an eNF can cause a negative impact on the latency measurements, due to the trade-off between packet processing and forwarding at SmartNIC. For example, with 1518-Byte packets, the latency difference between the two extremes (no offloading and full offloading) is only $\approx 100 \mu s$, and the latency for 90%@eNF offloading is smaller than the full offloading scenario. This benchmark is important to evaluate the trade-off on how to determine the traffic that can be processed at the SmartNIC and when partial offloading is better than full offloading.

4.3.4 Throughput results

Figure 4.8 shows the packet throughput measurements in Mpps for different test scenarios. For small packets, the throughput gain of VNF offloading is evident. For example, it reaches up to +10x with 64-Byte packets, and the gain decreases as the packet size increases.

For packet sizes greater than 1024, the offloading does not affect significantly the throughput performance, whereas the latency, in contrast, has been substantially reduced, as explained in the previous experiment (see Figure 4.7).

Figure 4.9 complements the visualisation of throughput measurements now in Mbps for the same tests. Given these results, it is clear that partial and full offloading help to reduce packet losses for small and medium size packets.

For packet sizes smaller than 1280 Bytes, we can observe that the VNF without offloading is not able to process packets at a line rate and offers very poor performance. On the other hand, the partial and full offloading scenarios have a great impact on increasing the total traffic that can be processed by the firewall application. Note that the percentage of traffic processed by eNF plays a crucial role for small packets: for the 90%@eNF scenarios, the throughput achieves line rate speed for packet sizes equal to or greater than 256 Bytes, while the 50%@eNF scenarios can achieve line rate speed for packet sizes equal or greater than 512 Bytes.

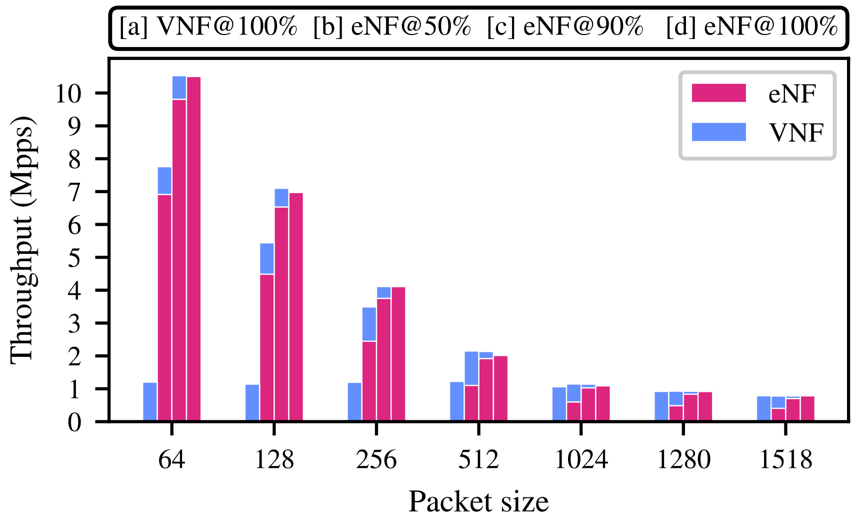


Figure 4.8: Throughput in Millions of Packets per Second (Mpps).

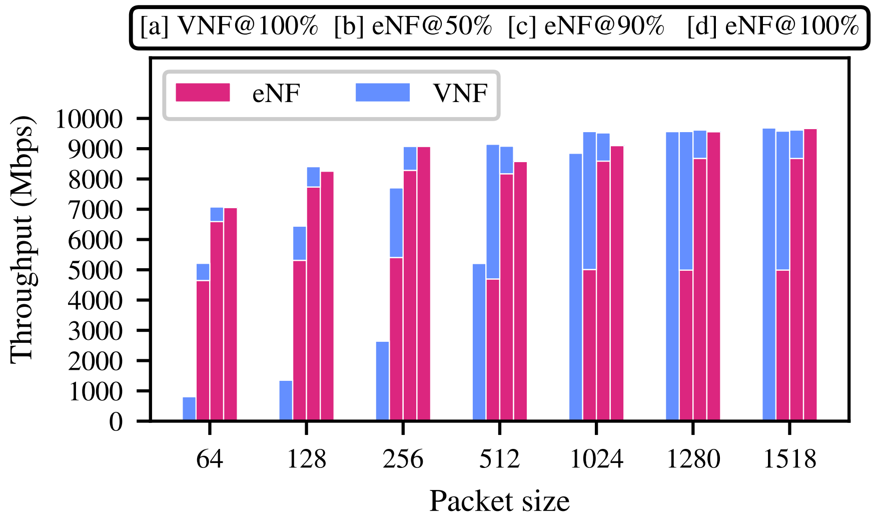


Figure 4.9: Throughput in Megabits per second (Mbps).

Similarly to latency tests, throughput tests also show that full VNF offloading may slightly reduce performance when compared to partial VNF offloading in some cases. This is also due to the trade-off between packet processing and forwarding. However, this difference could be even greater for a more complex eNF and reinforces the idea that one should consider partial offloading of a VNF based on how much processing can be shared between an eNF and a VNF.

4.3.5 vCPU usage

The vCPU usage for the VNF vFW, considering small packets (64 Bytes) and big packets (1518 Bytes), are presented in Figure 4.10. The experiment allocates 2 vCPUs to the VNF (out of 12 total vCPUs) at Physical Server 2. The chart shows the influence of VNF processing for different offloading cases, except for full offloading when all traffic is processed at eNF and only SmartNIC CPU is used. For 1518-Byte packets, there is 100% vCPU consumption in the scenario with no offload (100%@VNF), and a substantial decrease in the vCPU consumption (less than 5% vCPU usage) for the offloading scenarios. Besides, little difference could be seen when comparing the two partial offloading cases (50%@eNF and 90%@eNF), since the processing demand is not high in terms of Mpps (see Fig. 4.8).

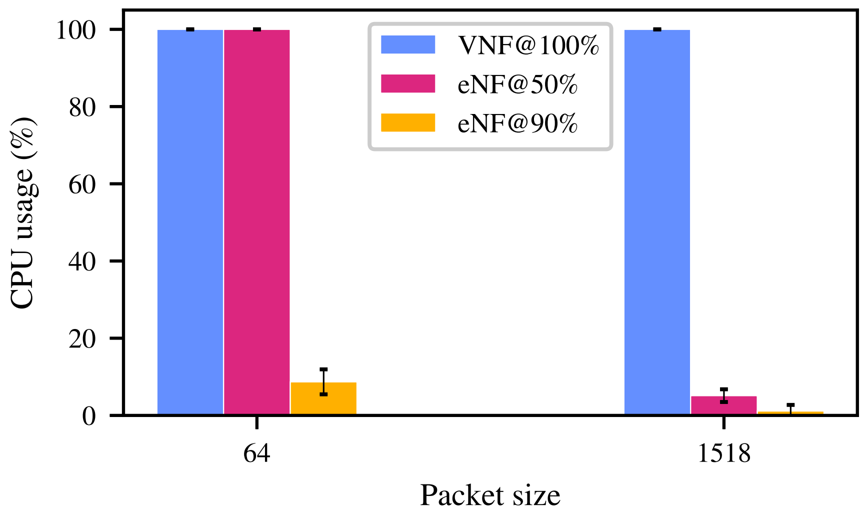


Figure 4.10: VNF CPU usage for two extreme packet sizes.

For 64-Byte packets, the processing demand is much higher in terms of Mpps (see Fig. 4.8). If we correlate the vCPU usage with the throughput of Figure 4.9, we can observe that the scenario with no offload (100%@VNF) presents very poor performance (less than 1 Gbps), while the scenario 50%@eNF achieves higher throughput (more than 5 Gbps) with the same vCPU consumption (100% vCPU usage). Moreover, the scenario 90%@eNF achieves even higher throughput (more than 7 Gbps) with much lower vCPU consumption (8% vCPU usage), as it offloads most of the processing to the SmartNIC.

These results demonstrate experimentally that partial offloading can improve latency and throughput performance while saving vCPU usage and releasing resources for traditional VMs that run on the same hypervisor as the VNFs, which enables a better allocation of resources by the tenant.

4.4 Chapter remarks

In this chapter, we expose the practical use of NFV, i.e., programmability and offloading constraints, using PIaFFE [Mafioletti et al. 2020] framework by creating VNF programming abstractions as building blocks using P4 language and the P4 Data Structure element for multi-level chaining to offload VNFs using eNFs in SmartNICs, which can be dynamic or static -defined triggers.

We demonstrated PIaFFE flexibility in a use case with three eNFs and one VNF. These eNFs were deployed on a commodity programmable SmartNIC, while its VNF counterpart on a kernel-based hypervisor, the latter emulated a tenant environment acting like a cloud platform.

Moreover, we have improved the data structure from the original proposal, now implementing an Offload Balancer concept, which enables the full or partial offload of network traffic acting by three different fashions: internally, externally or dynamically triggered. Our evaluations show that even with partial traffic offloading to the SmartNIC, latency and throughput can be significantly improved alongside reducing vCPU usage at virtual hosts and, consequently, at the main host, increasing the capacity for virtualisation and resources.

Chapter 5

Hacking programmable data planes in cloud robotics: intercepting and modifying ROS messaging

5.1 Overview

Envisioned as a key factor for the upcoming generation of service robots, cloud-enabled robots will also play important roles in areas such as eHealth and Industry 4.0 [Saha and Dasgupta 2018]. Given resource constraints imposed by embedded hardware, the possibility of offloading processing into a programmable element closer to the robots (e.g. edge) allows more cost-effective robots to cooperate in unstructured environments [Saha and Dasgupta 2018].

Network-related issues (e.g., latency) may prevent further advances in cloud robotics, and edge computing techniques have the potential to alleviate such constraints [Mello et al. 2019]. Nevertheless, edge computing is an expensive architecture solution when compared to the cloud and may not suit some latency-sensitive and critical applications in production environments (e.g., lower-level controllers). Thus, there is room for exploiting this market using state-of-the-art network programmability. In-network computing is a promising field that uses the capabilities of programmable network devices (e.g., programmable switches and NICs) to offload computing to the network [Nour et al. 2020].

Figure 5.1 presents the concept of In-Network Edge (INE), in which a programmable data plane connects robots to edge and cloud servers, and allows for robotic functionality to be instantiated within the network. Leveraging in-network applications based on a consolidated network programming language, such as the P4 language, may enhance management and control at the edge. Since network devices are in physical proximity to robots and distributed sensors, the use of in-network computing for robotics also reduces the overall latency, which is especially interesting for time-critical applications. In this context, INE may be enabled by the P4 language and an NFV framework [Mafioletti et al. 2020], with the potential to unleash real-time functionality in networked robotics.

Nevertheless, routing data from networked robotic systems via programmable network devices opens another window of opportunity for attackers trying to get

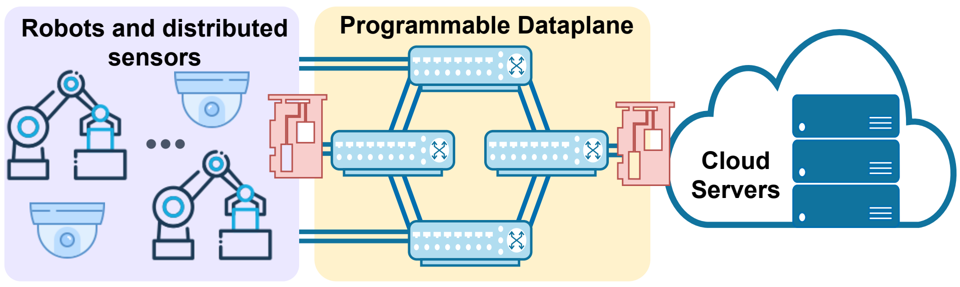


Figure 5.1: Cloud Robotics and Programmable Data Plane.

access to the system. In this sense, in-network computing is a double-edged sword: vulnerabilities in how data is transmitted and interpreted can be explored from within the network. Thus, in a threat model in which malicious applications are running inside programmable devices, aspects related to data security, integrity and validity must be taken into further consideration. Multiple works have advocated for the use of in-network computing for robotics (e.g., [Glebke et al. 2019, Cesen et al. 2020, Kunze et al. 2021]), but to the best of our knowledge, this is the first work to address security concerns introduced by programmable network devices to networked robotic systems.

We argue that most current robotic systems are vulnerable to simple attacks in a programmable data plane. We discuss two threat models in which robotic systems might be driven to unstable conditions by a compromised network device. We demonstrate such vulnerabilities using a networked robotics system based on the Robot Operating System (ROS), which is currently the most adopted robot development framework, communicating over a P4-enabled network device. The main contribution of this work is to cast light on how attacks that are well described in the literature can be refactored to be launched from the network itself.

5.2 ROS use-case

This section describes the potential constraints, the design and the enablers of our following proposals to exploit the security in a programmable data plane.

5.2.1 Experimentation design

For demonstrative purposes, we select a use-case inspired by the challenge of exposing security concerns about cloud robotics using the Robotics Operating System (ROS) as a proof-of-concept. Here, the framework needs to parse and modify raw packets, situated at high-level layers, i.e. transport and application network layers, in order to achieve the proposed security exploit.

Thus, this chapter includes the implementation of a VNF using PIaFFE which is capable of intercepting and changing application data inside higher layers of the network stack, proving that a jeopardised device could be used to compromise the security of the Cloud Robotics system. In order to do this, the PIaFFE framework will cover the implementation of an eNF that acts in almost all network layers, from the link and network layers for providing packet forwarding, to the transport and application layers, changing protocol and data, as shown in Figure 5.2.

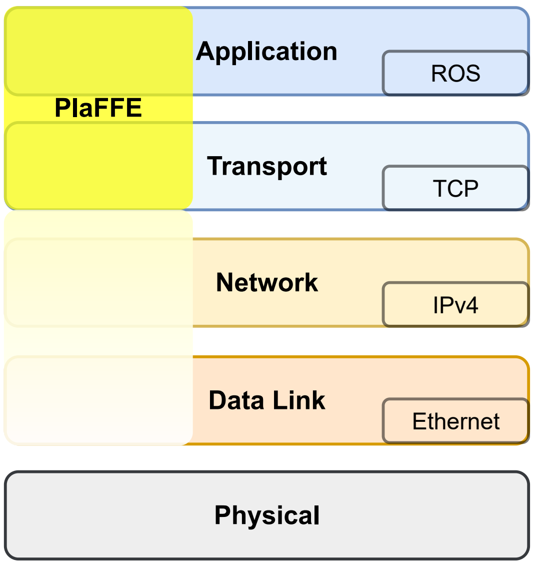


Figure 5.2: Network layers handled by PIaFFE in this Chapter.

To address the early-mentioned challenge, we have envisioned the design based on the following enablers:

- a mechanism capable of intercepting, storing and modifying ROS standard messages;
- keeps the current implementations of both ROS and robot simulator, working transparently to the applications.

5.2.1.1 Placement of embedded network function and its security

In NFV, the VNF placement is normally defined by where and how many instances of each network function should be placed and allocated. When using this paradigm in conjunction with programmable data planes, we have to define the most desirable place to execute a specific network function, in order to optimise both hardware and software resources.

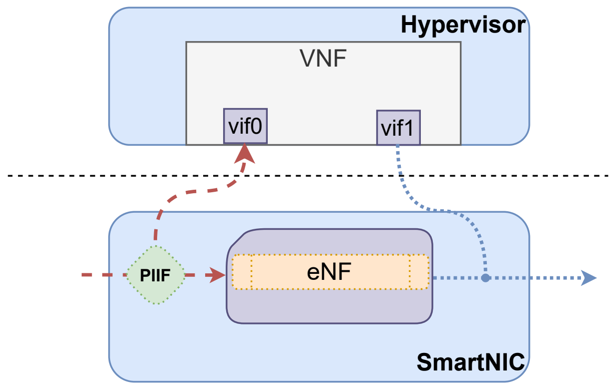


Figure 5.3: PIaFFE and multi-level chaining concept.

PIaFFE [Mafioletti et al. 2020] is a framework that uses decomposing and deploying approaches to transform Virtual Network Functions (VNFs) into small embedded Network Functions (eNFs) on in-network processors, allowing the correct placement and balance between hardware capabilities from a programmable NIC, using the flexibility of traditional VNFs running on virtual hosts. As depicted by Figure 5.3, the PIaFFE framework can be used to deploy micro-applications into programmable data planes, creating small functions that can cope with network-related services (i.e. firewall, routing) and/or high-level network applications (i.e. data encryption, telemetry), enabling the inference on the upper application stack into the network packets. As long as the network traffic arrives at the SmartNIC, PIaFFE employs the Intercepting and Forwarding (PIIF) element – which can be a hash table, a bloom filter or a simple trigger, for instance – to steer traffic through the eNF or send it up to the VNF at the virtualisation layer, using SR-IOV as an efficient communication channel.

In this use-case, we propose the use of PIaFFE for creating ROS micro-applications that are able to interact with the network in a transparent fashion, using the PIIF with an external trigger to enable/disable the in-network computing, processing network packets when forwarding them, exposing the potential disruption that can be achieved using a programmable data plane with a malicious embedded code.

5.2.2 Usage of the framework

This section describes the patterns to build an exploit using programmable hardware, acting in a transparent fashion for robots and the cloud.

Figure 5.4 shows how the eNF parses and recognises ROS data inside network packets. A standard library was reused from the PIaFFE framework to identify the traditional network headers (e.g. Ethernet, IPv4, IPv6, TCP), plus a new one was generated to parse and classify ROS message headers and ROS data contained by the payload of the packet. The eNF acts like a layer 2 forwarder application, but concurring with the forwarder was another application, which is able to check and store data from ROS messages for further use.

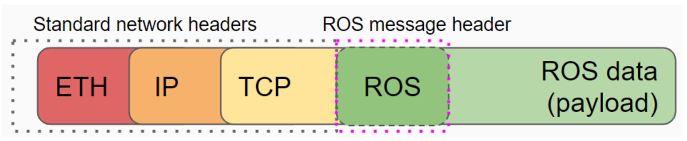


Figure 5.4: Network packet headers, ROS message header and ROS data.

Moreover, the PIaFFE framework enables the user to define new libraries and headers, permitting the extension of types of messages that can be interpreted into an eNF inside the network programmable device, extending the interaction between the programmable data plane and Cloud Robotics applications.

5.2.3 Background review

As with other embedded systems, robotics manufacturers place a high priority on safety, development cost, speed to market, and providing customer features. Cybersecurity is a lower, and sometimes, forgotten priority in part because security is not a primary consideration for customers [Clark et al. 2017]. The end-user demands more concerns on cost, usability, features and functionality [Mirjalili and Lenstra 2008]. However, due to their direct interaction with human beings, robotics applications must be required to be more secure and safe than other embedded systems. Overall, various security concerns, issues, vulnerabilities, and threats are constantly arising, including the malicious misuse of these robots via cyberattacks, which may result in serious injuries and even death [Kirschgens et al. 2018, Ángel Manuel Guerrero-Higueras et al. 2018].

5.2.4 Vulnerabilities in programmable network devices

As desirable properties for securing network communications, we can list confidentiality, message integrity, end-point authentication and operational security [Kurose and Ross 2016]. Together, these terms define a desirable secure environment for devices to ensure that the messages exchanged between them are authentic and to increase the network's confidence.

Programmable devices are extremely effective in improving performance and giving flexibility to the data plane, through the use of domain-specific programming languages and compilers which could exploit that emerging programmable data plane devices. However, they bring novel security concerns, such as targeted denial of service and state exhaustion attacks, and data plane attacks [Dargahi et al. 2017], which need to be exposed both to programmers and the end-user.

A compromised device can disrupt the overall network's confidence since all packets need to be routed through this type of device, and given the location on the premises, which is generally considered safe, even a small malicious exploit inserted on it may result in disastrous problems. Notwithstanding, non-programmable network devices are also susceptible to these problems [CVE 2020, Wang et al. 2020], and now with the possibility to customise the data plane using high-level programming languages, these constraints can escalate to a new level, leaving room for new types of exploits that can be pre-inserted and programmed to catch information or simply to disrupt all network traffic passing through these devices.

5.2.4.1 Security in ROS systems

ROS [Quigley et al. 2009] is the *de facto* standard middleware for robotics. It separates application management issues and the communication of data, which is abstracted by the middleware. This decentralisation of components pairs well with networked robotic systems and ROS can be used as a base to connect multiple components, even when parts of the system are in the cloud. Given its widespread use, ROS is ideal to demonstrate how programmable data planes can be used to explore vulnerabilities in robotic systems.

To briefly explain how ROS works, the pieces of software that compose the system (i.e., nodes) are executed on top of an operating system (OS) and use the underlying ROS infrastructure to communicate with other nodes. Communication is carried out by transmitting well-defined data structures (i.e., messages) via topics in

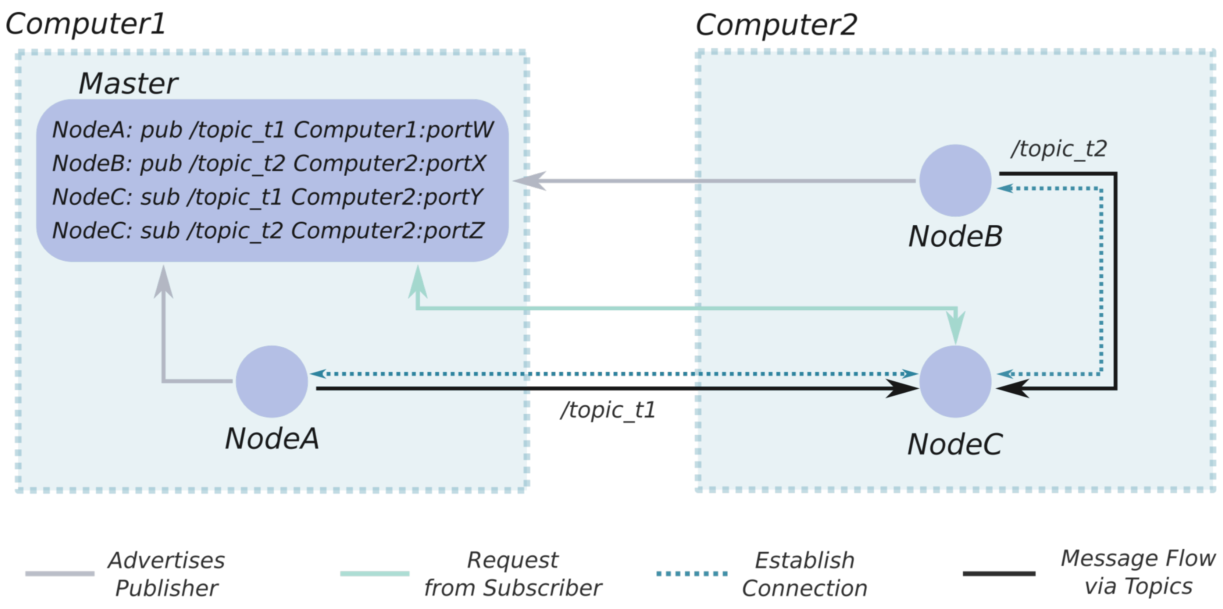

Computer1
Computer2
Master
NodeA: pub /topic_t1 Computer1:portW
NodeB: pub /topic_t2 Computer2:portX
NodeC: sub /topic_t1 Computer2:portY
NodeC: sub /topic_t2 Computer2:portZ
/topic_t2
NodeB
NodeA
/topic_t1
NodeC
Advertises Publisher
Request from Subscriber
Establish Connection
Message Flow via Topics

Figure 5.5: ROS operation: the master registers and provides information for publishers and subscribers to establish connections and allow nodes to exchange information directly via topics.

a pub/sub architecture. A ROS master is implemented using XMLRPC, a stateless HTTP-based protocol, and provides naming and registration services to the nodes in the system. The master is used to connect nodes and, once a publishing node locates a subscriber they can communicate peer-to-peer. This process is illustrated in Fig. 5.5: upon startup, nodes *NodeA* and *NodeB* advertise to the master that they will publish messages to topics named */topic1* and */topic2*, respectively; to subscribe to those topics, *NodeC* communicates with the master to register two subscribers, each associated with one topic. Then, as there are both publishers and subscribers registered to each topic, the master sends instructions to the nodes to establish a connection and start communicating with one another.

Traditionally, custom protocols based on TCP/IP or UDP/IP are used in ROS (TCPROS and UDPROS, respectively) [Quigley et al. 2009]. Upon initialisation, a publisher node provides the ROS master with the topic name and associated data structure (i.e., ROS message type). The master then informs the node about all of the other publishers and subscribers. When a node becomes a publisher on a topic, it will connect to subscribers to that specific topic and there is no access control for topics beyond the data type MD5 hash [Rivera et al. 2019].

In general, no security mechanisms regarding confidentiality, integrity or authenticity are implemented out of the box. Data is transmitted unencrypted and the only information needed to decode intercepted ROS packets is the type of the message being transmitted, which describes the associated data structure and is usually standardised in the ROS framework. Thus, ROS messages can be intercepted and decoded either by direct inspection of the payload. Notwithstanding, the stateless API does not take into account what is happening in the network.

The Secure ROS (SROS) was introduced to add cryptography and security measures to ROS, thus addressing some of its vulnerabilities [White et al. 2016]. The SROS enables TLS support for encrypted communication within ROS, access control policies and user-space tooling to generate node key pairs. Nevertheless, the use of the SROS hampers performance and such a trade-off must be taken into account. Alternatives to the SROS have been presented, but none seem to have gained traction. Dieber *et al.* [Dieber et al. 2020] identified that compromised user space libraries can break communication's confidence in ROS. We argue that a compromised programmable data plane can be exploited to the same end.

It is worth noting that ROS' evolution, ROS2 tackles some of the security concerns with ROS. It uses DDS as the messaging layer, which supports a security standard for protecting messages between parties with access control enforcement. The SROS2 is the current initiative to integrate DDS security and ROS2 and has been found to significantly affect system performance [Kim et al. 2018], which is also observed in other approaches [DiLuoffo et al. 2018]. ROS' user base is much larger than ROS2's, making it more relevant to the security concerns we raise, some of which also apply to ROS2 and will be addressed in our future work.

5.3 Threat models exposed

In this section, we explore two vulnerabilities in the use of ROS within the paradigm of programmable data planes. First, we show how insecure communication can be leveraged by a compromised network device to take over robot control. Then, we demonstrate how even systems implementing secure communication are susceptible to accepting old of repeated data as valid. Given the projects described in the literature and in the open-source community, the majority of ROS systems would be vulnerable to such attacks. In the remainder of this section, for each vulnerability, we describe a corresponding threat model and demonstrate how it can be exploited transparently via in-network computing.

5.3.1 In-network hijacking: man-in-the-middle attack

Despite ROS' distributed nature, security aspects in communication are not implemented. Common message types encapsulate raw data and it's safe to assume that most ROS' users take no further steps into securing it. Thus, if one can isolate a packet flow and identify the associated data structure interpreting the data is straightforward. More than that, it becomes easy to tamper with the flow by directly modifying the data being transmitted; this can be done in a coherent manner, replacing the actual data for feasible – but incorrect – data, thus making it harder for any automated function to detect that the system has been compromised.

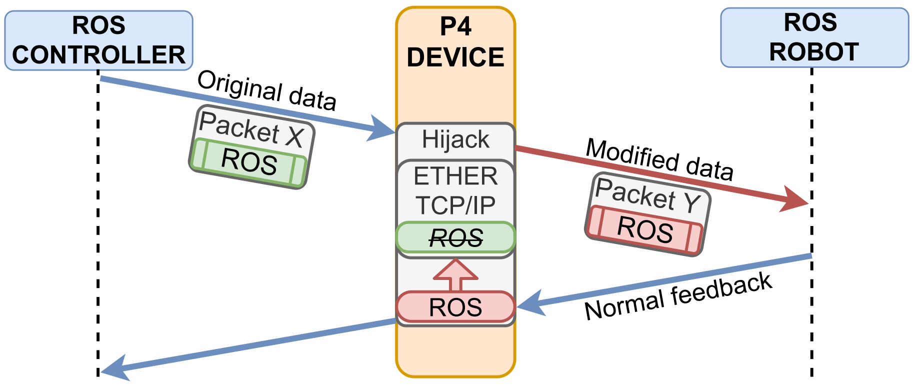
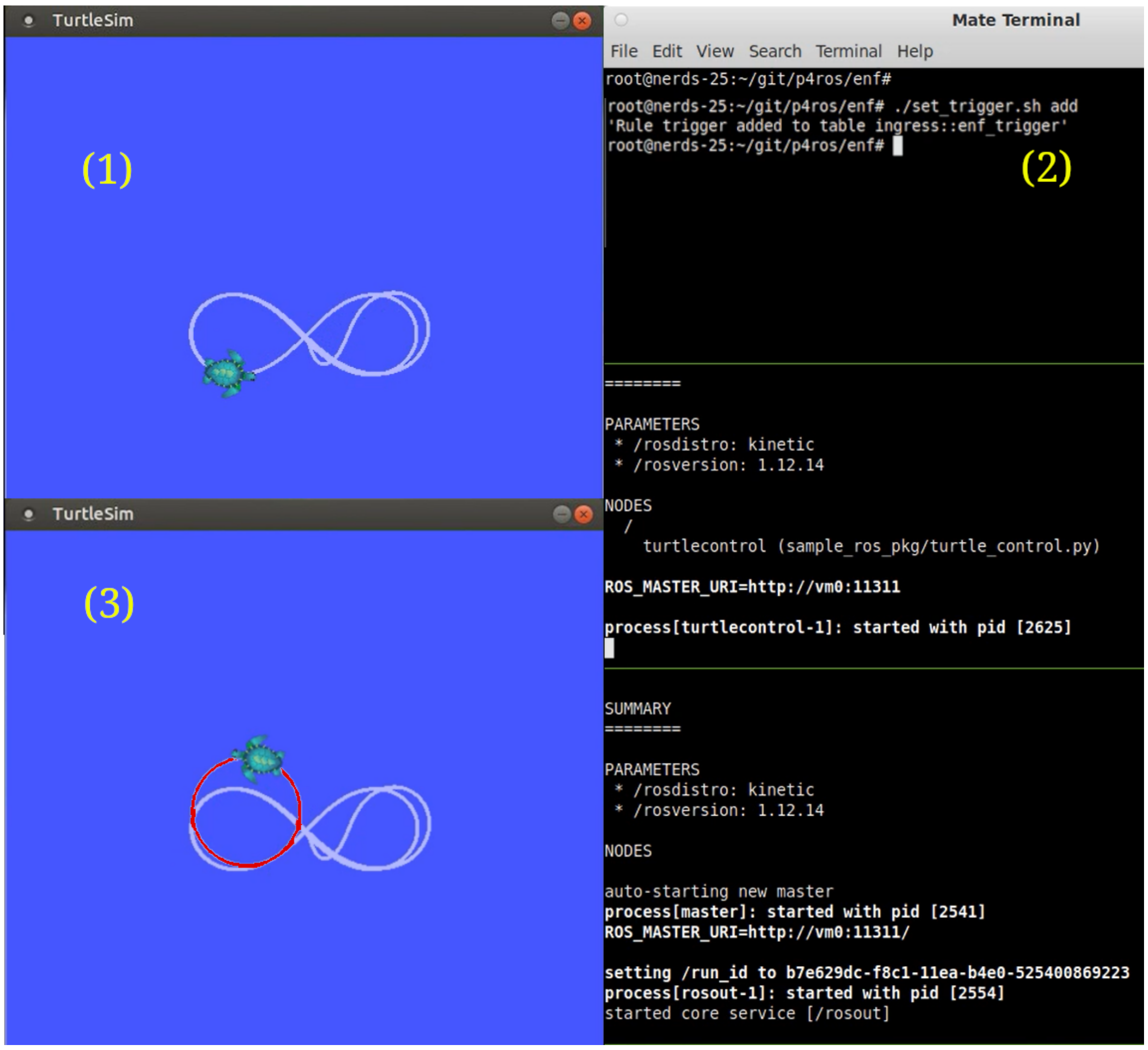


Figure 5.6: Threat model: hijacking.

We consider that programmable network devices have transparent access to data exchanged in a ROS system. Such devices may be used to identify a given flow and alter the data being transmitted. There are three assumptions: (i) ROS' standard libraries and messages are used; (ii) ROS' nodes are distributed among different machines, and; (iii) transmitted data is not encrypted. All of these assumptions are compatible with the standard ROS operation and are present in most systems.

To demonstrate how to explore such vulnerabilities using programmable network devices, we implement a ROS system composed of two VMs communicating over a P4-enabled SmartNIC. One VM simulates a robot and the other VM instantiates a navigation controller. In this feedback loop, the robot sends its current position to the controller, which generates velocity commands for the robot. In the SmartNIC, we implement a P4 library relating packet size and part of the payload to standard message types. The SmartNIC parses its traffic to single out packets identified as messages coming from the simulated robot's controller, as exemplified in Figure 5.6. Once the target flow is identified, its payload is altered to a given valid instruction, thus hijacking the robot's motion.



```
root@nerds-25:~/git/p4ros/enf#
root@nerds-25:~/git/p4ros/enf# ./set_trigger.sh add
'Rule trigger added to table ingress::enf_trigger'
root@nerds-25:~/git/p4ros/enf#
```

```
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES
  /
    turtlecontrol (sample_ros_pkg/turtle_control.py)

ROS_MASTER_URI=http://vm0:11311

process[turtlecontrol-1]: started with pid [2625]
```

```
SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [2541]
ROS_MASTER_URI=http://vm0:11311/

setting /run_id to b7e629dc-f8c1-11ea-b4e0-525400869223
process[rosout-1]: started with pid [2554]
started core service [/rosout]
```

Figure 5.7: TurtleSim robot simulator: (1) Normal robot behaviour; (2) Setting a trigger via P4 table; (3) Abnormal robot behaviour (red track)

We use the *TurtleSim* package to simulate a robot being controlled to follow an eight-shape trajectory. Given an external trigger, the hijacking takes place and the messages sent from the controller to the robot are modified to trick the robot into following a spurious trajectory. Figure 5.7 illustrates the outcomes of our demonstration. The controller tracks an eight-shaped trajectory and, once the attack begins, the robot receives tampered instructions to make it follow a circular path. By changing the payload directly, it is possible to inject any instruction to pose as a legitimate control output. A similar approach could be used to modify sensor data, introducing artificial noise to hamper system performance without leaving clear signs of an attack.

In our demonstration, we rely on live per-packet detection of a given ROS flow. A more sophisticated approach would be to identify ROS' control packets exchanged among nodes and the ROS master to identify publish/subscriber pairs and the type of message exchanged among them. Thus, to identify the flow associated with a given pair of nodes, one only needs to parse the TCP header in such packets.

Encryption could be used in all communications within a ROS system to overcome such a threat, especially in production environments. Nevertheless, encryption algorithms introduce processing and bandwidth overheads that must be considered since they can degrade the overall performance. In case the internal network is considered to be secure and encryption is only used when communicating with the cloud, a compromised programmable network device within the local network might also be exploited for a similar result.

5.3.2 In-network replay attack

A replay attack occurs when an individual eavesdrops on secure network communication, intercepts the packets, and then delays or re-sends it to misdirect the receiver into doing what the attacker wants, or simply messes with the final proposes of the overall communication. Thereafter, programmable network devices are not only capable of modifying packet data but can also clone or create packets, inserting them into a given flow.

Thus, it becomes possible to clone valid ROS messages to mislead subscribers, whether data is encrypted or not. Since ROS' subscribers rely on a middleware-level buffer for relaying incoming messages, overflowing the buffer may lead to undefined behaviour in the ROS system. This could be achieved by inserting considerably lower levels of throughput than it would be necessary to disrupt the robot network. By being done transparently, from the inside of the network, such an attack could be hard to be detected and could demand direct inspection of incoming packets in the robot.

By default, neither ROS nor the new ROS2 implements mechanisms to verify if the arriving data is duplicated. A sequence field is present in the header of some messages in ROS, but it was deprecated in earlier versions. This means that, unless the developer explicitly implements methods to avoid message repetition, every message extracted from the overflowed buffer would be considered valid. One way to mitigate such a threat without directly addressing it is to discard messages with old timestamps but, even if a short time-to-live mechanism is present, the buffer size would have to be tuned accordingly to limit the amount of accepted duplicate data.

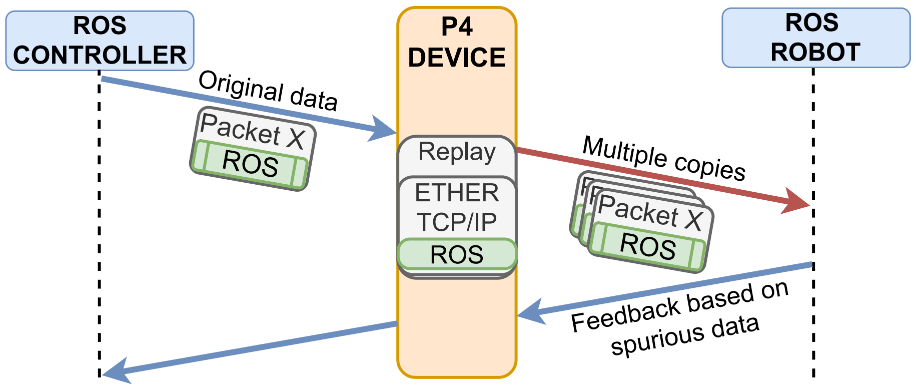


Figure 5.8: Threat model: replay attack.

For this threat model, we consider that a programmable network device may insert packets in publisher/subscriber flows in ROS systems, as illustrated in Figure 5.8. A given flow can be targeted and have its packets cloned to overflow subscribers' buffers. There are two assumptions: (i) ROS' nodes are distributed among different machines, and; (ii) subscriber nodes do not implement any mechanisms to discard duplicate messages. These assumptions are compatible with the standard ROS operation and are present in most users' ROS systems.

We use the same setup described in the Subsection 5.3.1 to demonstrate how to generate unstable behaviours in robot systems by flooding a ROS subscriber. In the SmartNIC, we implement a P4 library relating packet size and part of the payload to common standard ROS message types. The SmartNIC is now programmed with P4 code that parses its traffic to single out a given flow and replicate its packets. In separate experiments, we target the subscriber in the controller node, which receives the current robot position, and the subscriber in the robot's motion node.

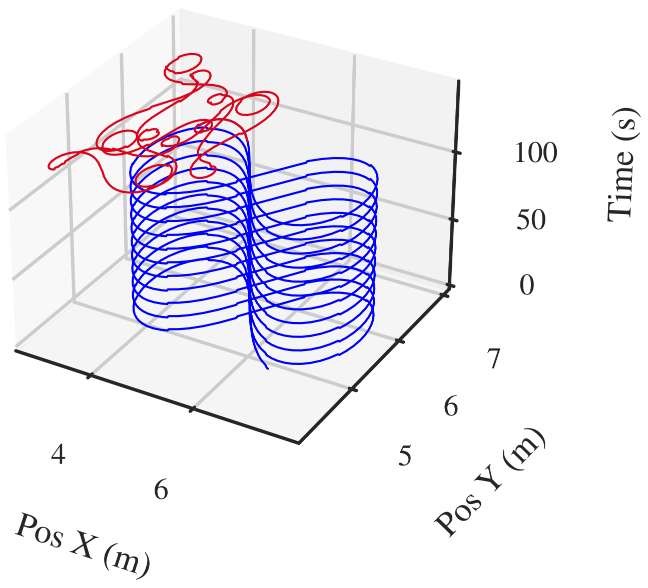


Figure 5.9: Replay attack: a) robot position on plane varying with time, normal (blue) versus disturbed (red) operation

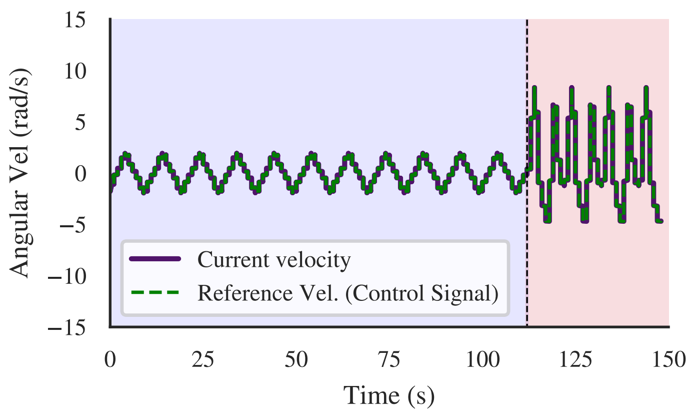


Figure 5.10: Replay attack: angular velocity of the robot under normal operation (blue zone) and under attack (red zone); the red dashed line marks the beginning of the attack.

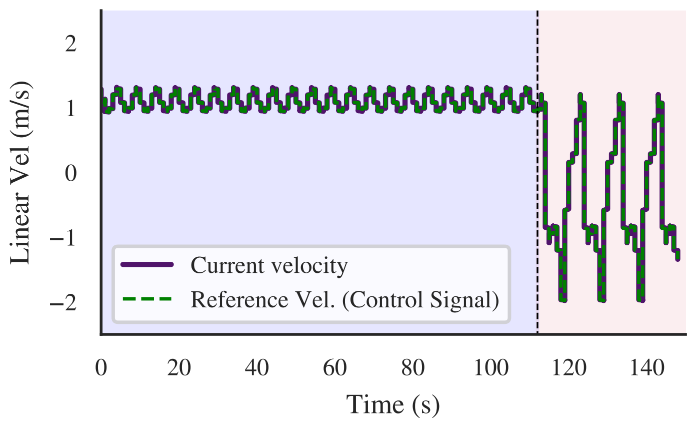


Figure 5.11: Replay attack: linear velocity of the robot under normal operation (blue zone) and under attack (red zone); the red dashed line marks the beginning of the attack.

As we can see in Figure 5.9, the desired robot trajectory (blue) was disturbed by the in-network replay attack, forming a different pattern (red) due to the overload generated by the eNF on the network device. Following Figure 5.10 and 5.11, we also have the angular and linear velocities of the TurtleSim, captured before (blue zone) and during (red zone) the attack, showing the erratic behaviour of the system as a result of the disruption caused by the replay attack. In a real scenario of robotics, this could lead to a catastrophic outcome, since the robot would supposedly be receiving orders from the controller, which in turn would not be aware of what the robot was doing.

5.4 Chapter remarks

In this Section, we discuss concerns about the effects of jeopardised programmable devices on networked robotic systems. We show that micro-applications can be inserted into the data plane to intercept and modify network packets. In particular, we implement such micro-applications to interact with packets of ROS systems, causing problems for both controller and the robot fleet. We also discuss the main threat models, pointing at some aspects of the current implementations of networked robotics, and showing the results using P4-enabled network hardware. Our results confirm the possibility of exploiting programmable network devices as attack vectors for robotics systems.

Although the results shown here go against the end-to-end principle discussed before, the main objective was to raise concerns about a device with compromised security, intentionally or not, revealing the weaknesses of a programmable network device. Therefore, knowing the weaknesses, we can devise a way to ensure the confidentiality and reliability of the data, adding and checking information via hash algorithms, such as SHA-256, and/or cryptography using an embedded network function (eNF) near the robot fleet, increasing both security and performance using the hardware network to offloading processing.

Chapter 6

Providing data integrity by interacting and modifying raw packets: an SHA use-case

6.1 Overview

In the area of networking, one of the most CPU-demanding activities is the securitisation of communications using cryptography. Payload data processing and specialised cryptographic hash functions are commonly employed in secure and resilient communication, avoiding unauthorised access, data manipulation and modification. In-network processor-based programmable network interface cards (NIC) offer a solution for offloading network traffic, allowing hosts to process general computations onto the programmable NIC while keeping the support of high-level applications on them.

The advent of new programming paradigms like P4 [Bosshart et al. 2014] for high-speed packet processing platforms has enabled a wide variety of networking applications. For instance, in-network caching [Jin et al. 2017], heavy-hitter detection [Sivaraman et al. 2017, da Silva et al. 2018], and network load balancing [Grigoryan et al. 2019] enable these functions that were primarily designated for running into commodity servers to migrate to the data plane, using hash-based data structures like bloom filters, count–min and hash tables to track network flows directly into the data plane.

However, the P4 language currently only supports a few non-cryptographic hash algorithms based on cyclic redundancy check (CRC) or checksum computations typically used in network protocols like TCP, IPv4 and IPv6 checksums due to target hardware constraints. However, there is still a need for packet-based functionalities involving basic data security and verification. In those cases, “true” cryptographic hash functions are required, for example, hash-based message authentication codes (HMAC) [Turner 2008], providing message authentication, or any other cryptographic hash functions used to increase resilience against hash collisions for hash-based applications cited early.

To address advanced secure applications implementation using in-network computing and also the disaggregation of services into different virtualised functions, we argue that complex cryptographic hash functions extensively used nowadays, like Secure Hash Algorithm 2 (SHA-2), should be ported onto P4 targets. For this end, however, one has to deal first with the limited computational resources, such as the ones available at SmartNICs [NETRONOME 2019], when enabling the offloading of secure applications to the data plane.

To the best of our knowledge, this is the first proof-of-concept implementation of an SHA-2 variant library on a commodity programmable in-networking processor using the P4 language. Our strategy is to implement a shared object library as an *"extern"* to overcome language-related restrictions. This way we manage to embed a complex algorithm that includes loop statements and other non-native features.

A testbed is also created for a benchmark study on finding out whether SHA-2 embedding as Network Function (NF) is worth not considering throughput and latency features. Those metrics are key for providing SHA along with Service Function Chaining (SFC) in a Network Function Virtualisation (NFV) modern infrastructures. The SmartNIC embedded Network Function (eNF) prototype implementation is checked against other two software data plane implementations, namely, Open vSwitch software switch (OvS) and Intel Data Plane Development Kit (DPDK), both ported with the same SHA-2 library.

However, this intuitive scenario is not trivial to be implemented considering that the following points should be addressed:

i. a proper mechanism for exchanging network traffic between software and hardware must be found;

ii. an application using a domain-specific programming language and its *extern* option that fits into the SmartNIC hardware must be built;

We evaluate the initial implementation of the PIaFFE framework for addressing the above challenges. This initiative aims to open new avenues for programmers to develop applications using an in-networking processor which requires cryptographic hash functions running into a programmable network device.

6.2 SHA use-case

This section describes the design principles, architecture and enablers, and the potential limitations of our approach using PIaFFE to develop and deploy a secure hash algorithm onto network programmable hardware. Here, PIaFFE will be used to develop and deploy a complex algorithm using a limited resources network programmable hardware, checking the usage of the framework for this category of application to further enable the usage of cryptographic algorithms not present in the SmartNIC's original design.

6.2.1 Experimentation design

The design principles of our proposed use case for experimenting, and consequently, evolving our framework were motivated by the challenge of deploying complex network applications on resource-limited network hardware, providing support for modern applications that require such a type of cryptographic hash algorithm for assuring data integrity. Thus, this chapter includes the implementation of the Secure Hash Algorithm v2 (SHA-2) using PIaFFE, covering almost all network layers, from transport to data link layers for forwarding and at the application layer for acquiring data to be hashed, as depicted in Figure 6.1. Thus, PIaFFE will tackle these network layers using in-network computing for offloading processing data.

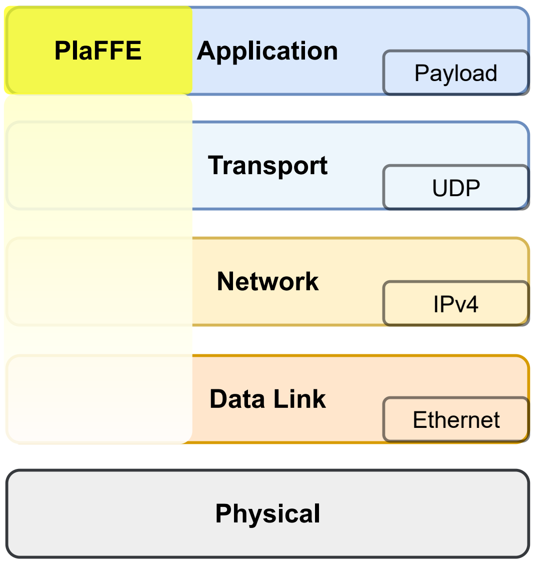


Figure 6.1: Network layers handled by PIaFFE in this Chapter.

6.2.2 Usage of the framework

Hash functions play a key role in various network applications being fundamental for modern network communication. As more and more functionality is being moved into programmable data planes, supporting hash functions with strong cryptographic properties will be a key enabler for various networking use cases. In order to address the early-mentioned challenges, we design a simple architecture based on the following enablers from the PIaFFE framework:

i. functional decomposition for deployment;

ii. multi-level forwarding between hardware and software.

6.2.3 Functional decomposition for deployment

Most network functions are deployed as a monolith application, whereby all components are bundled together into a single piece of code. This can lead to maintenance challenges, as well as slow down the trailing of new technologies. Decomposition is therefore a significant step towards evolving VNFs to be cloud-native and much more agile and scalable. And decomposing must be undertaken for the majority of the data plane, control and signalling VNFs. In this context, we define decomposition as breaking a monolithic network function into a set of small applications, or “micro-applications”.

Decomposition also allows for common functions to be stripped away from the core logic of the applications. This allows the applications themselves to be lighter – which makes them easier and quicker to develop, manage, and deploy into programmable network hardware. In addition, it centralises core functions and operates in an "as-a-service" model. We also want to be able to reuse common functionalities, but not have to pay for them multiple times. The process of decomposition also gives micro-application owners the opportunity to remove unnecessary functionality from the application logic [Richardson 2018].

The challenge in realising any micro-application-based software using decomposition is to identify the set of functionalities, and this is difficult because it requires domain-specific knowledge. One way to approach decomposition is to leverage domain expertise (e.g., by consulting with domain-specific developers) or to study existing open-source network applications, identifying smaller functional units. As a key to employing functional decomposition, PIaFFE Framework [Mafioletti et al. 2020] defines a roadmap towards creating an embedded network function (eNF) using an approach based on the application network-related characteristics (e.g., packet parsing, classifying, processing and forwarding), re-architecting the network programming ecosystem.

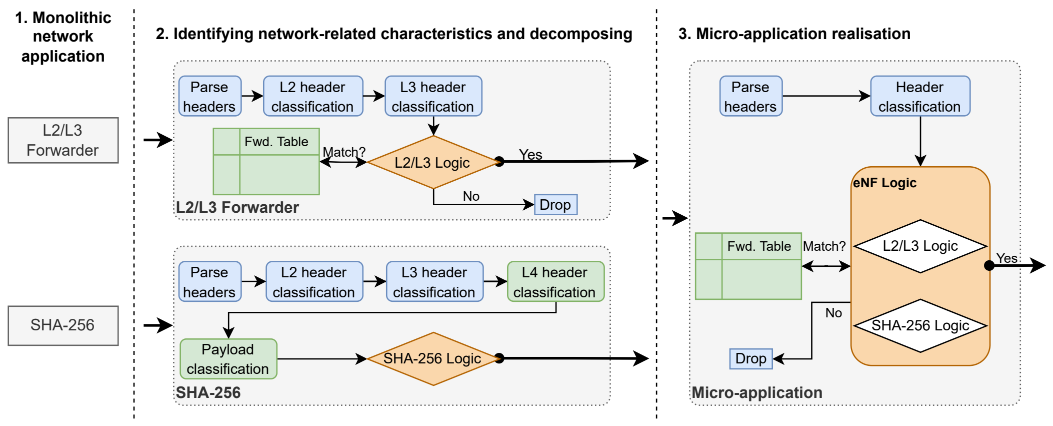


Figure 6.2: Illustrative example of decomposition using a set of network applications.

From the previous approach defined in Chapter 3, we extend the decomposition method for a specific network function, now using for reference two monolithic network applications, the first one based on a **L2/L3 packet forwarder** and the second one based on **SHA-256 algorithm**, we can outline the steps for decomposing these two applications into a new micro-application, as shown in Figure 6.2. The first step for decomposing multiple applications is to identify the network-related characteristics of each one, starting with the **L2/L3 Forwarder**.

In the same way, we enumerate the characteristics of the next application, in this case, **SHA-256 algorithm**: here, we use the same colour (blue) to highlight the overlapped characteristics and another colour (green) for specific functions. We also used a specific colour (orange) to emphasise the main logic behind both applications, keeping the pattern established before. Based on the colour map, we can merge the repeated functions of the applications into the **micro-application realisation**, using a single structure representing the two main applications, making it possible to be embedded in limited-resource programmable network hardware.

6.2.4 Multi-level forwarding between hardware and software

In order to switch traffic between hardware and software implementations, we employ the Intercepting and Forwarding element (PIIF) from PIaFFE, steering network traffic through the embedded Network Function or sending it up to the application at the virtualisation layer, using a bottom-up approach as shown in Figure 6.3: as soon as a packet arrives at the SmartNIC, it is either intercepted and forwarded to the "OS level", or kept into the "hardware level", being processed using the SmartNIC network hardware.

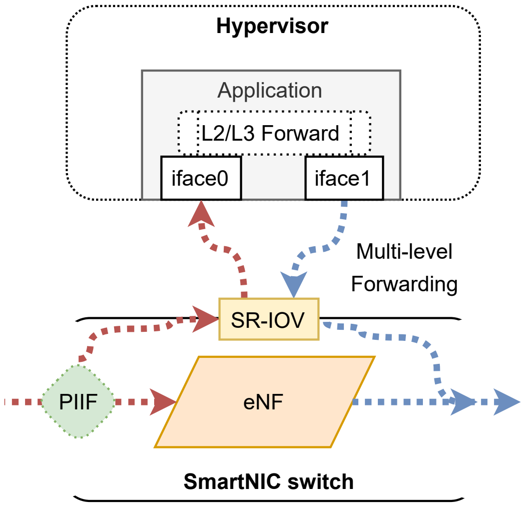


Figure 6.3: Multi-level forwarding using Intercepting and Forwarding (PIIF) element.

In Listing 6.1, it is presented a fragment of the P4 source code illustrating the PIIF element utilised in this use-case. This code defines a basic $(key, value)$ data structure. Whenever an incoming packet hits a table's entry (defined by its source IPv4 and UDP port), then it is steered to the software, otherwise, it is sent to the eNF pipeline that applies a specific hash function.

```
...
@name(".piif_table")
table piif {
  key = {
    standard_metadata.ingress_port: exact;
    hdr.ipv4.srcAddr: exact;
    hdr.udp.srcPort: exact;
  }
  actions = {
    NoAction;
  }
  size = 512;
  default_action = NoAction;
}
...
apply {
  if (piif.apply().hit) {
    sw_forward.apply();
  } else {
    do_sha256();
    hw_forward.apply();
  }
}
```

Listing 6.1: Pseudo code of the PIIF

6.2.5 Hash algorithm implementation

Our implementation is focusing on porting SHA-256 to a P4 programmable device for evaluating the performance of the basic cryptographic hash operations, which could be applied for a range of use cases further, such as HMAC calculations, hash-based networking applications and others.

P4 language lacks a loop control flow statement, and this limits the possibilities for implementing complex applications, such as SHA-256 which requires that for stages like shuffling and compressing data. To overcome this limitation, we have created an *extern* based on micro-C language (a variant of C language), as shown in the fragment of code in 6.2, which can be called from the P4 program as a "function", interacting with the P4 pipeline and its data/metadata.

```
#include "pif_plugin.h"
#include "plugin.h"
...
int pif_plugin_do_sha256(
            EXTRACTED_HEADERS_T *ext_hdrs,
            MATCH_DATA_T *match_data)
{
 PIF_PLUGIN_payload_T *payload;
 BYTE buf[SHA256_BLOCK_SIZE];
 SHA256_CTX ctx;
 /* Grab a pointer to the payload 'header' */
 payload=pif_plugin_hdr_get_payload(ext_hdrs);
 BYTE text[] = payload->block0;
 /* begin sha256 calculation */
 sha256_init(&ctx);
 sha256_update(&ctx, text, strlen(text));
 sha256_final(&ctx, buf);
 /* end sha256 calculation */
 /* storing hash into the specific field */
 payload->sha256_field = buf;
 return PIF_PLUGIN_RETURN_FORWARD;
}
```

Listing 6.2: A fragment of micro-C extern SHA-256 function.

The target hardware used for the implementation of our architecture (i.e., Netronome SmartNIC), has limited resources for hosting large applications. Thus, the firmware and NIC "storage" memory capacity must be properly used, otherwise, the image can no longer be loaded onto the SmartNIC. To tackle this constraint, we have developed a slim P4 code for L2/L3 packet forwarder based on static table entries in conjunction with the SHA-256 algorithm running as an *extern* as cited above, fitting with NIC's firmware image size.

As mentioned before, the micro-C code fragment 6.2 starts by exposing the headers and match data from the P4 program, which can be accessed and modified inside the *extern*, including the packet payload, which is represented by a "payload header" split into blocks of 512-bit size, due to header size restrictions imposed by the used target hardware. The SHA-256 function is called in three steps, in order to accomplish all stages of the algorithm, namely, the padding process, shuffling and compression. In the end, the hash is added to a specific header field previously defined in the P4 program. Here is defined by the $sha256\_field$ variable, being transmitted with the original network packet. In this phase of the processing, we are also using this approach in order to measure the time to generate and insert hashes using in-network computing, however, the functionality can be extended to accomplish more elaborated scenarios in future implementations.

One may even think of a combination of an optimised integration with native dedicated hardware acceleration components. However, crypto security accelerators are not usually accessible nor documented when using P4 language shipped with the NIC's SDKs. Thus, this integration is beyond the scope of this work.

6.3 Experimental evaluation

The performance of the secure hash function, in terms of latency and processing time (processing plus forwarding packets), is critical for high-performance applications. Thus, our evaluation aims to answer the following key question for NFV applications:

i. Compared with host-side execution, what are the expected latency savings of forwarding and processing network traffic through an in-network device?

ii. What are the latency statistics when running such micro-applications on different platforms?

iii. How close can we get to the maximum possible throughput in terms of packets per second (PPS)?

6.3.1 Benchmarking the virtualisation techniques

As far as test scenarios are concerned, we are comparing the in-networking computing capability with network forwarding technologies used on the Linux OS doing the same operation, as follows:

1. **Open vSwitch switch**: modified Open vSwitch software switch executed at kernel-mode as a layer 2 network forwarder with a selectable SHA-256 application, as shown in Figure 6.4.

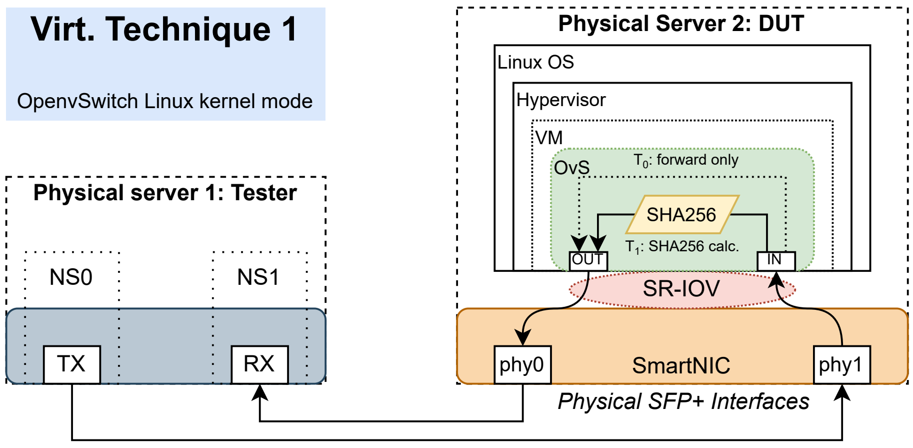


Figure 6.4: Virtualisation techniques: Open vSwitch software switch

2. **Intel DPDK**: Intel DPDK L2FWD application acting as a layer 2 forwarder and SHA-256 algorithm which can be turned on/off according to the traffic, as same as previous Open vSwitch switch, as depicted in Figure 6.5.

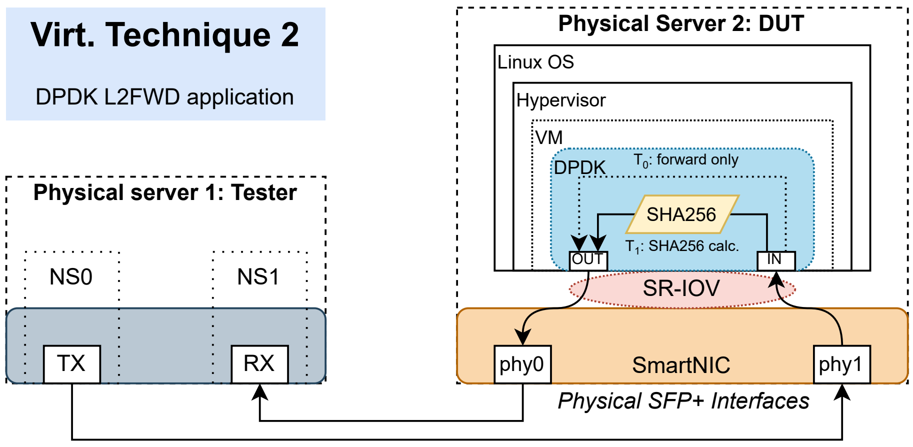


Figure 6.5: Virtualisation techniques: Intel DPDK L2 forwarder application

3. **P4-enabled hardware**: using a Netronome SmartNIC [NETRONOME 2019] with a layer 2 forwarder embedded network function, also running a ported SHA-256 application dynamically enabled, as we can see on Figure 6.6.

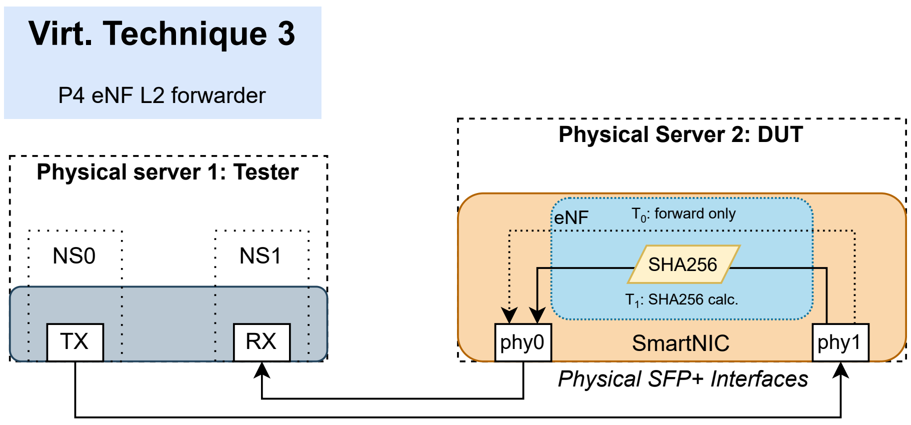


Figure 6.6: Virtualisation techniques: P4-enabled hardware

In all virtualisation techniques, the topology and network functionality are the same in each level (software and hardware levels): packets are generated in one host, being forwarded through the SmartNIC hardware in another host running Linux OS and network drivers in three different modes, respectively: 1) Linux netdev mode, 2) Intel DPDK mode or 3) P4 mode. All details about the testbed will be discussed below.

6.3.2 Experimentation setup

For the benchmark tests, we used the environment described in 6.2 and illustrated in Figure 6.7, using the technique explained in Figure 6.8.

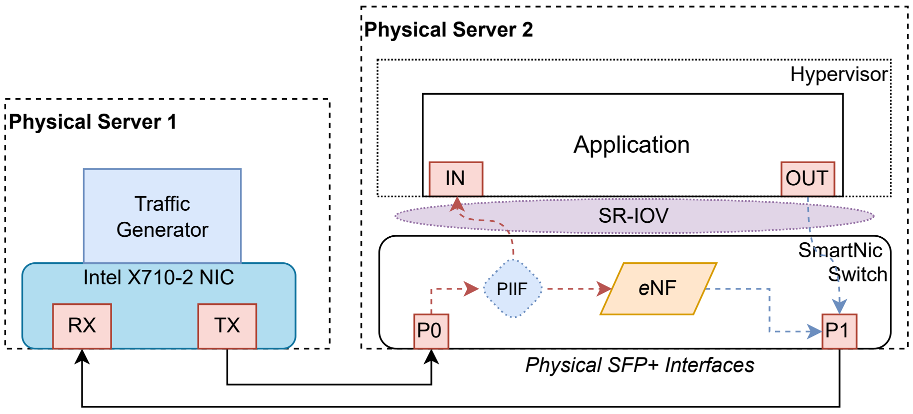


Figure 6.7: Testbed specification: Prototype implementation and testbed.

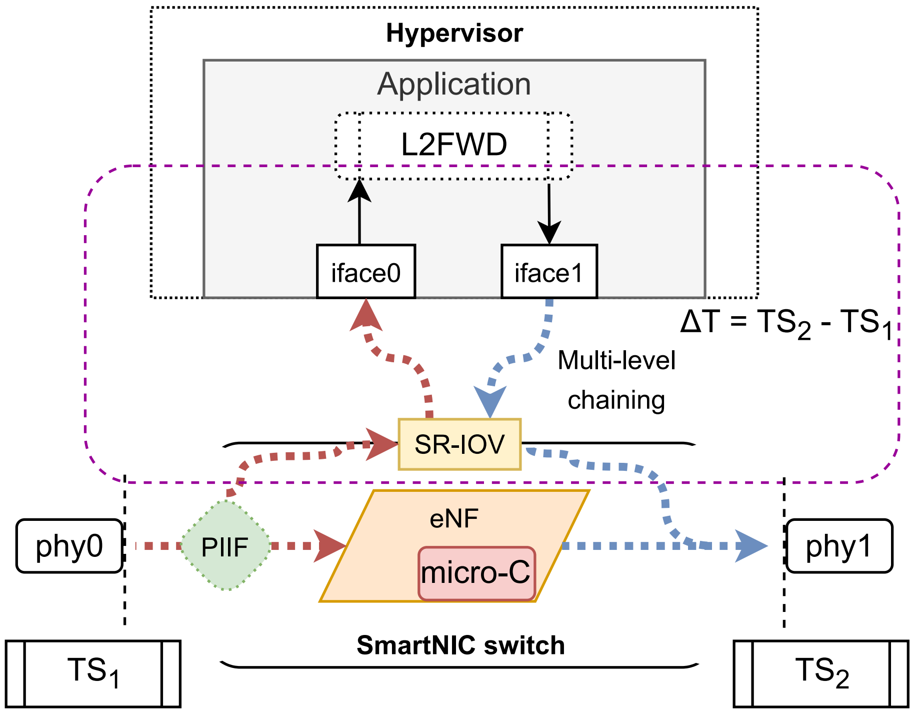


Figure 6.8: Using hardware timestamp to measure the time to process/deliver a packet.

The test consists of generating traffic that goes through each virtualisation technique specified before, measuring latency performance to forward-only and forward-calculate using a specific case. As a proof-of-concept, the Secure Hash Algorithm 256 (SHA-256) library, is widely used for authentication and encryption protocols, including SSL, TLS, IPsec, SSH, PGP, and secure password hashing on Linux OS. This algorithm is public and open-source, portable for most platforms. We built a library using our previous work, allowing us to check the feasibility of the development and deployment of complex applications using the framework to deliver a micro-application onto a programmable network device.

Hence, UDP packets are generated using pktgen-dpdk through the TX interface and received using the same application in RX. These packets are diverted according to the purpose of each experiment: selecting either the SmartNIC hardware only or the software level using a virtual machine. This virtual machine runs a forwarder application with the SHA-256 algorithm inside, that can be enabled or disabled dynamically during the experiments using a different UDP source port in TX, permitting the measurement for each virtualisation technique.

6.3.2.1 Testbed description

Our testbed consists of two machines connected back-to-back without any switching element in between, as shown in Fig. 6.7. One of them hosts the traffic generator, while the other bears the prototype and the Netronome SmartNIC NFP-4000 2x10G. Each machine is equipped with a 1x6-core Intel Xeon E5-2620 v3 2.4Ghz CPU and 2 threads per core (hyper-threading enabled), 8GB memory, and a DPDK compatible Intel X710-2 2x10G Ethernet NIC on the traffic generator side.

To collect the latency results, we use the SmartNIC's internal hardware clock to calculate the timestamps of each packet before it goes out of the SmartNIC to an application and inserts another timestamp when it comes back to the SmartNIC. This gives us more precision to calculate the total cost to run an application inside and outside of the SmartNIC, using a nanosecond scale, as shown in Figure 6.8. To collect the throughput (and consequently, the processing capacity) of the target, we employ a Lua script[1] to automatise the experiment and to collect packet data into a comma-separated value (CSV) file, to later be processed and generated the results.

6.3.2.2 Traffic generator and DUT

Physical Server 1 runs Intel pktgen-dpdk[2] version 20.04 traffic generator, while *Physical Server 2* acts as a Device Under Test (DUT), using applications developed and deployed into a network device, processing packets using in-network processor or a traditional application running on a qemu-kvm hypervisor, like in Figure 6.7. The application runs on their respective qemu-kvm virtual machine and has a dedicated embedded network function from SmartNIC, provided by SR-IOV pass-through on the hypervisor, with 8GB RAM and 2 CPU cores unpinned, in order to create a more realistic environment as possible. The physical servers run Ubuntu Linux Server 20.04, with kernel version 4.15.0-60, as same as the OS running on virtual machines.

6.3.3 Results

As we can see, in Figure 6.9, the Open vSwitch software switch, despite running on kernel-mode, presents the worst latency level: $\approx 197\ \mu sec$ to forward only and $\approx 216\ \mu$s to calculate the SHA-256 function ($+9.89\ \%$). DPDK implementation improves it significantly down to around $67\ \mu s$ in both cases (with and without SHA-256 function), showing that, as might be expected, the Linux kernel bypass method used by DPDK is a key technology for such applications. And finally, using an eNF, we can achieve the best result in terms of latency, reaching 834 nanoseconds to forward a packet (not visible in the chart, due to its small value), but this value increases to $\approx 40.8\ \mu$s when we run the SHA-256 function on the hardware, rising at least 40x the time to process and forward a network packet. This highlights the cost of running such a complex application on a resource-limited network programmable device. However, the latency values stay below the best software use-case (DPDK), showing that the network device could be used for offloading the calculation of SHA-256, alleviating the host-side processing.

[1]https://www.lua.org
[2]http://git.dpdk.org/apps/pktgen-dpdk

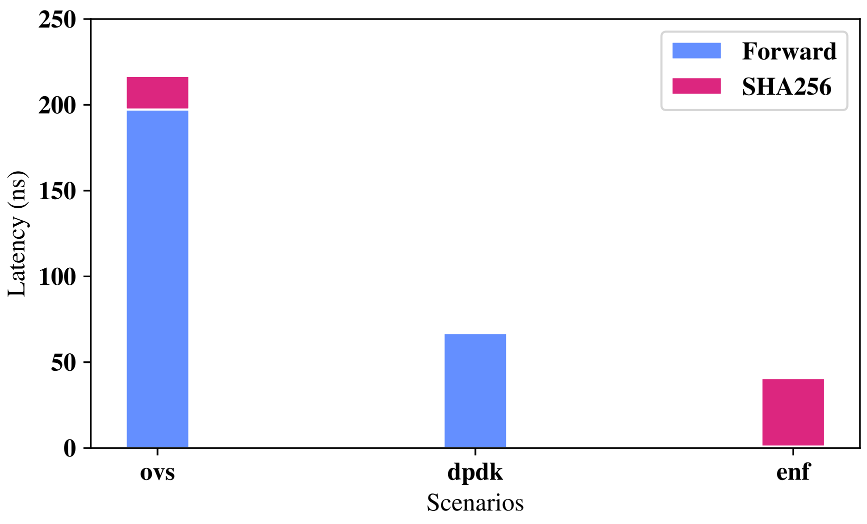


Figure 6.9: Latency (Processing and forwarding) for software and hardware micro-applications running SHA-256 algorithm and network forwarding: Total latency using different types of forwarding/calculation virtualisation techniques.

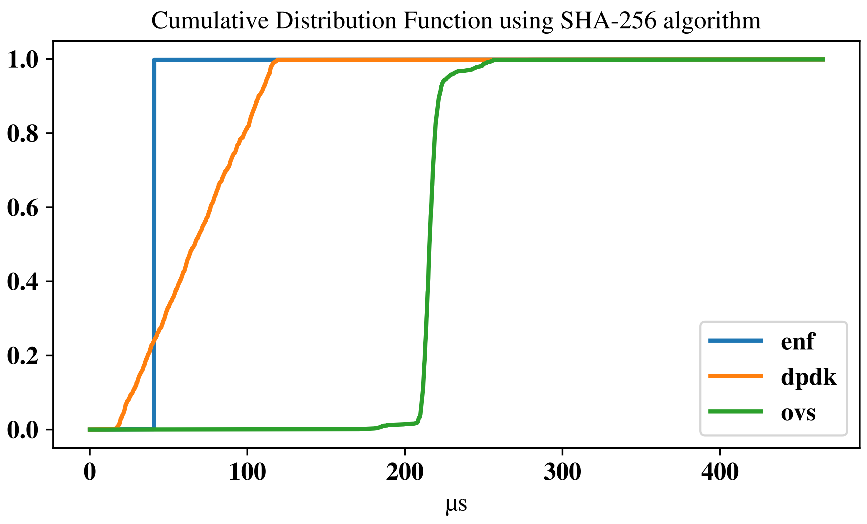


Figure 6.10: Cumulative Distribution Function (CDF) Probability for SHA-256 calculation and forwarding.

As far as Cumulative Distribution Function (CDF) is concerned, Figure 6.10 allows us to see that the eNF has a small variability (almost a deterministic behaviour) when compared with other implementations (Open vSwitch and DPDK) based in software. This is due to the randomness added by the stacks in the OS used to forward the packet from the NIC to the software and then back to the NIC. The DPDK-based application, which enables a shortcut to the user-level applications, the CDF resembles a well-behaved uniform distribute function but with enlarged variability when compared to the OvS implementation. This is important in NFV because latency higher-order moments are important for real-time applications.

There may be a trade-off between end-to-end latency and jitter that may work against choosing the DPDK solution.

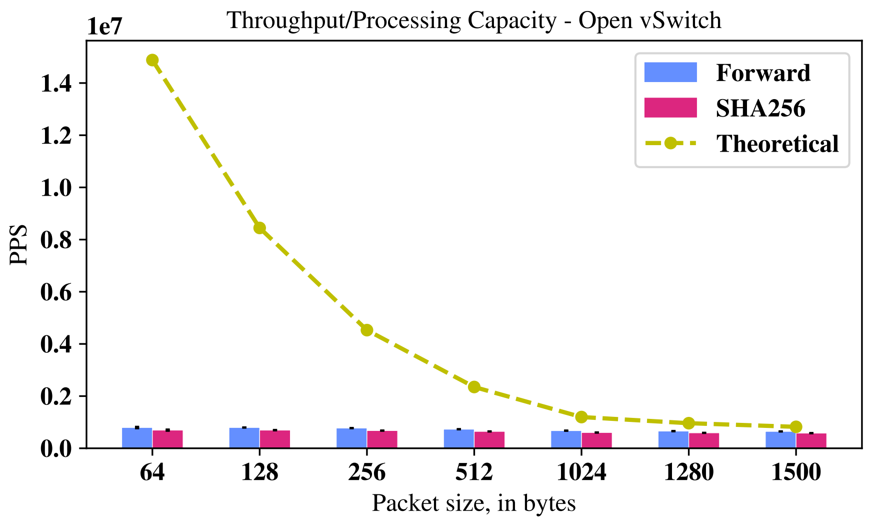


Figure 6.11: Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: Open vSwitch (OVS) software switch.

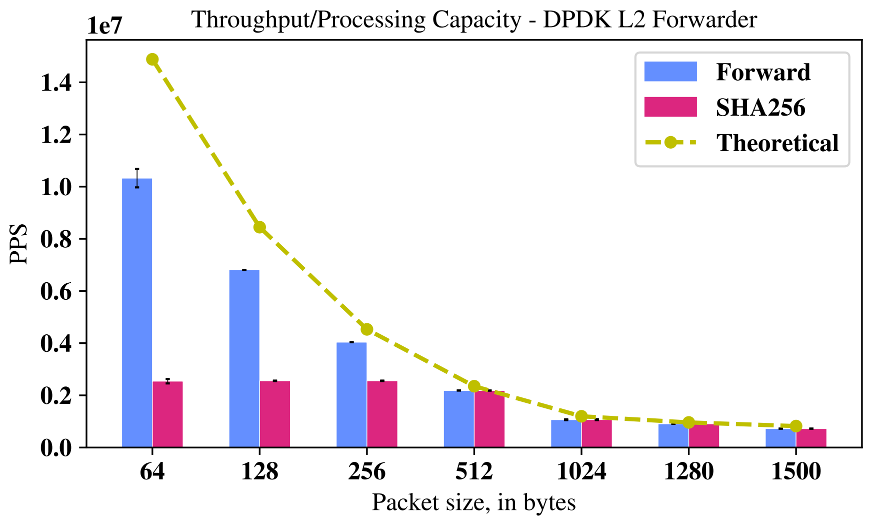


Figure 6.12: Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: DPDK application.

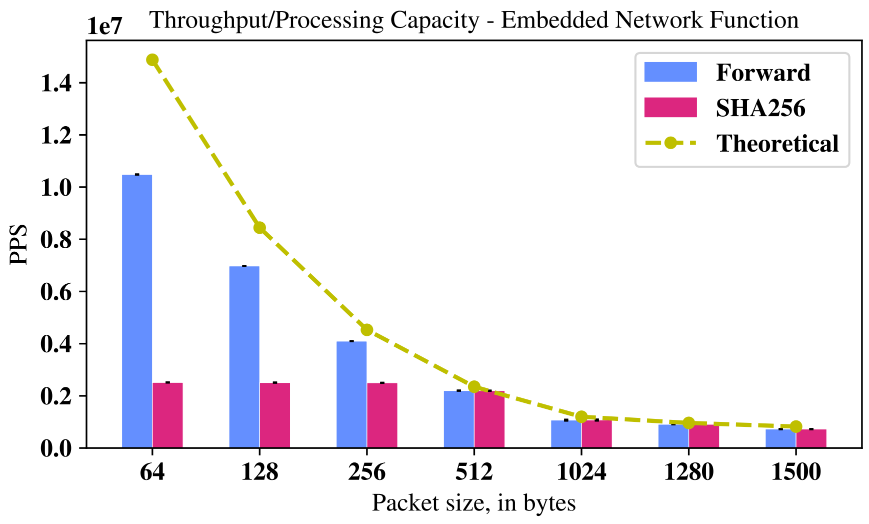


Figure 6.13: Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: embedded network function with P4 in NIC.

Figures 6.13, 6.12 and 6.11 depict the capacity of each virtualisation technique for processing packets using SHA-256 algorithm. In addition, a theoretical line showing the maximum packets per second (PPS) throughput according to the packet size using 10 Gbit/s speed, following the Ethernet frame MTU (64-bytes and 1500-bytes packets) of the experiment in order to check on the maximum and sustained processing rates when forwarding packets through each virtualisation technique.

It is important to highlight that the SmartNIC is not able to achieve line-rate speeds with small packets (<512 bytes), as confirmed in a set of benchmarks available on the manufacturer's document library[3]. However, the validity of our comparative results was not affected by this limitation once all experiments were performed using the same setup.

Starting with the Open vSwitch switch, we noted that the software switch cannot process and forward small to large packets (Fig. 6.11) alike, keeping the packets per second rate below a figure of $800K$, possibly due to the kernel/user modes and the OS stack, being unable to forward or process packets close to 10 Gbit/s. Meanwhile, DPDK has a more consistent result, showing that the polling mode and CPU resource reservation makes a difference in packet processing/forwarding actions. The eNF results are practically the same as those of the DPDK application, showing that despite the fact that packets are being processed in the network hardware, and consequently avoiding the OS stack and interrupts, the hardware is stressed when calculating and forwarding small packets at near line-rate. A tradeoff between cost and performance may lean the choice of the platform toward DPDK solutions when PPS is concerned.

Based on the results and low-latency application requirements in need of cryptographic hash functions, we can conclude that Open vSwitch, which is a well-known software switch intensively used by hypervisors, had an unsuitable performance

[3]https://www.netronome.com/document-library

to use as an enabler for low-latency applications. It takes more than $210\ \mu$s to compute the SHA-256 algorithm and forward a packet. Thus, the time to send the network packets from the "hardware plane" to the "software plane" is quite high using the Open vSwitch switch. It can be reduced when using the DPDK application. Considering that the time to forward a packet on SmartNIC hardware is lower than $1\ \mu$s, all values exposed previously basically correspond to the time to switch the context from hardware to software levels. Moreover, comparing each approach, the eNF is the best option to forward a packet, in the same way, to compute a small work on its hardware. In fact, the eNF can complete the SHA-256 hash function, and in sequence forward a packet, using $\approx 60\%$ of the time needed by the best VNF case (DPDK) and $\approx 19\ \%$ of the time needed by the worst case (OvS).

6.4 Chapter remarks

Cryptographic hash functions are essential for assuring the security and privacy of network communication obliterating data from malicious entities, avoiding attacks and establishing mutual or multi-party trust relationships. This paper investigated alternatives for offloading the processing of cryptographic hash algorithms from application-level hosts into in-network computing for SFC in NFV infrastructures.

We described architecture and prototype implementations integrating secure hashing algorithm 2 (SHA-2) in different virtualisation techniques to benchmark the P4 hardware target platform. Our design of a P4 data plane is able to run a secure hash algorithm using a mechanism for steering traffic between software and hardware levels, enabling the use of complex applications for generating hashes.

Results obtained for hashing using in-network computing demonstrate that it is feasible, and in some cases better than the traditional virtualisation techniques, including DPDK. Applications running on general-purpose CPUs can do offloading of its processing to the network device, freeing resources for tenants and improving latency and throughput.

Chapter 7

Low-latency passive optical networks (PON) through intercepting and modifying DBA frames

7.1 Overview

The next-generation networks will need to deal with a new set of groundbreaking applications such as intelligent transportation systems, tactile Internet, and cyber-physical systems (CPS) for Industry 4.0. Among the major service groups, ultra-reliable and low-latency communications (URLLC) aims to foster new applications with stringent demands on latency and reliability [Pocovi et al. 2018]. Nevertheless, notwithstanding their promising capabilities, those networks also expect to coexist with different resources and technologies, such as WiFi and LTE networks crossing optical layers.

Passive Optical Networks (PONs) are considered a cost-effective architecture for ubiquitous broadband delivery, due to their ability to share cost and capacity across endpoints. For this reason, they are being increasingly considered as a possible solution to connect small cells in 5G functional split architecture (i.e., supporting ORAN Remote Unit(RU)-Distributed Unit(DU) split as well as higher level splits).

One of the main drawbacks of the PON point-to-multipoint topology is upstream latency, which is higher compared to the simpler point-to-point solution, as the scheduling mechanism requires the exchange of reports and bandwidth map calculations that introduce an additional few hundreds of microsecond delay.

We can cite approaches such as Cooperative DBA [Tashiro et al. 2014], recently standardised as Cooperative Transport Interface (CTI) [O-RAN 2020], which have addressed this latency issue by providing coordination between mobile scheduling at the DU and OLT scheduling. However, CTI works because in Cloud-RAN the low latency issue is generated by a protocol mismatch between PON and RAN rather than by application-level requirements. For this reason, the CTI is able to fix the issue by sharing the advanced scheduling information from the DU with the OLT. This exchange of information is further facilitated by PON virtualisation mechanisms [Ruffini et al. 2020], which simplifies the integration between wireless and optical technologies.

In this chapter, we address a more challenging issue on PON networks, where the low latency requirement comes from the application. We argue that using a mechanism for intercepting and reacting to these low-latency requirements directly on a network programmable hardware – after they are forwarded to a virtual network function – may even unleash comprehensive, reliable and low-latency service delivery using micro-applications and In-Network Computing. We also experimentally demonstrate a concept of a *Fast Intercept Mechanism*, to achieve low latency on PON upstream frames, leveraging the Virtual DBA implementation.

7.2 PON use-case

This section describes the potential limitations, the design principles and the architecture of our following proposal for the *Fast Intercept Mechanism* in Passive Optical Networks using PIaFFE as an enabler. Thus, this chapter explains how to use the framework to develop and deploy applications into programmable network hardware based on the passive optical networks use-case and its upstream scheduling algorithm.

7.2.1 Experimentation design

The experimentation design of our proposed use case for experimenting and evolving our framework was motivated by the challenge to tackle low-latency applications using PON, once we have to work with a low-level network layer and protocols, such as PON GEM frames. This chapter includes the implementation of a VNF using PIaFFE which is capable of intercepting and changing control data inside PON GEM frames, providing low latency for applications that want it. In order to do this, the PIaFFE framework will cover the implementation of an eNF that acts in the data link layer, as shown in Figure 7.1.

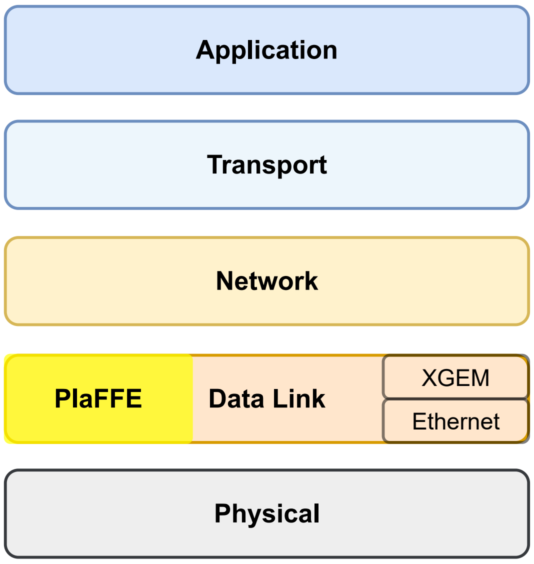


Figure 7.1: Network layers handled by PIaFFE in this Chapter.

To address the early-mentioned challenge, we have envisioned the design based on the following enablers:

i. a mechanism capable of intercepting and modifying PON frames based on the application requirements;

ii. split the upstream DBA scheduling to reach normal and low-latency bandwidth map (BWMAP);

iii. keeps the current implementation of the virtual DBA proposal, working transparently with the high-level applications.

7.2.2 Usage of the framework

Our proposal is to implement a mix of classical and modern approaches. The bulk of the scheduling calculations is executed as virtualised network functions on the host, however, some key calculations are performed on the network interface card. These key calculations usually pertain to scheduling routines which have strict response times.

Figure 7.2 depict the main objective of the use-case. According to 3GPP, the delay threshold for 5G front-haul traffic is set at 250 μs [Alliance 2015]. Thus, the focus here is to implement a low-latency solution for small cell virtualisation and C-RAN, which is defined to be less or equal to 10km of distance (50 μs), which will be the limit for providing a low-latency schedule mechanism improvement using in-network computing.

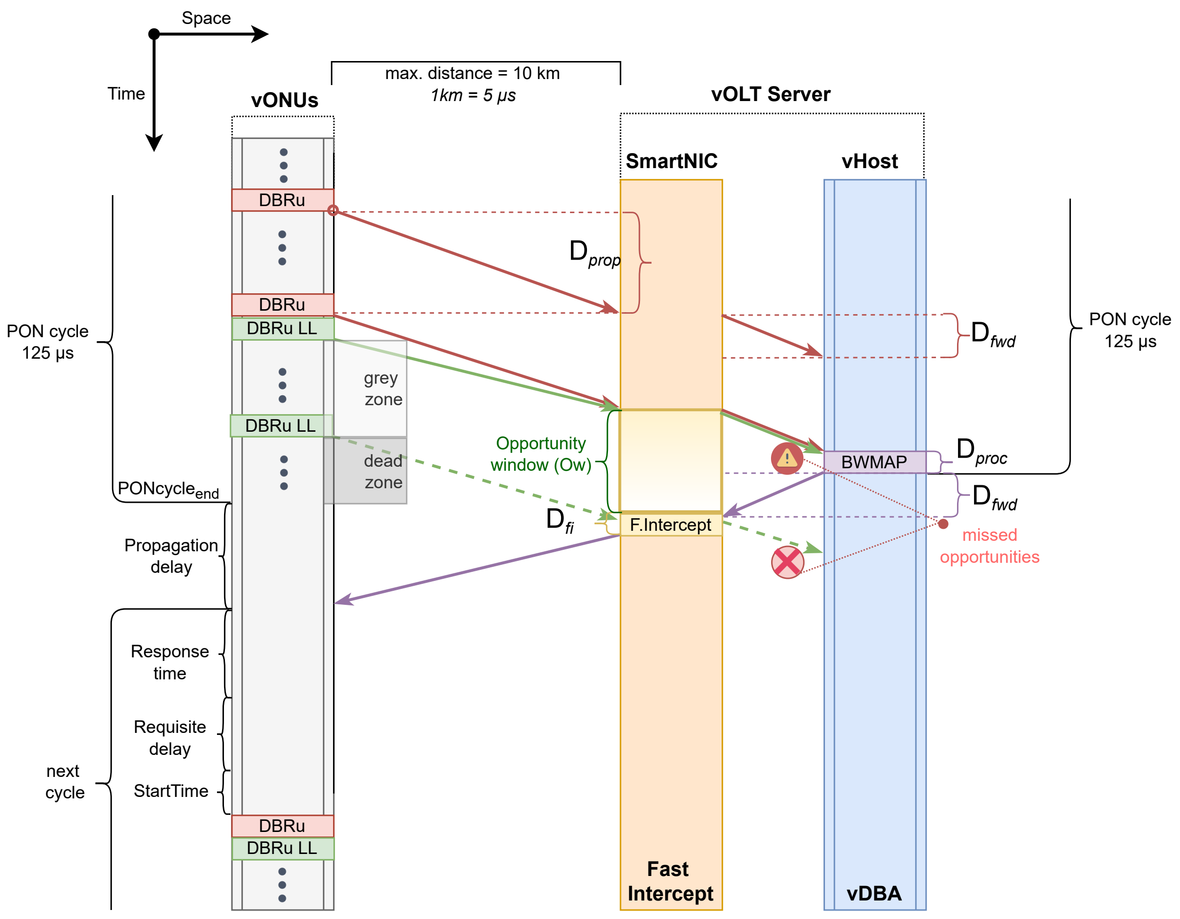


Figure 7.2: Fast Intercept concept.

According to Figure 7.2, the standard TDM-PON cycle is disclosed by 125 μs, which is synchronised between vOLT and vONUs. As shown in the diagram, we define as a *grey zone* ($T_{gz,begin}$ and $T_{gz,end}$) the interval in which a DBRu report will miss the opportunity to be included in the current BWMAP calculation cycle at vOLT perspective, however, will benefit from the Fast Intercept mechanism, being inserted into the current BWMAP cycle into the SmartNIC, interval which is called as *opportunity window* ($\Delta\ Ow$). In the same way, we use the term *dead zone* ($T_{dz,begin}$ and $T_{dz,end}$) to delimit the interval in which a DBRu report will definitively miss the opportunity to be included in the current BWMAP calculation cycle at vOLT side. Based on the diagram, we can find the values of each variable using the following equations:

$$T_{gz,begin} = [125 - (D_{prop} + D_{fwd})] + 1 \tag{7.1}$$

$$T_{gz,end} = (125 - D_{fwd} - D_{proc} - D_{fi}) \tag{7.2}$$

$$\Delta\ Ow = T_{gz,begin} - T_{gz,end} \tag{7.3}$$

$$T_{dz,begin} = T_{gz,end} + 1 \tag{7.4}$$

$$T_{dz},end = PONcycle_{end} \tag{7.5}$$

The range of ONU response time is a system-wide parameter that is chosen to give the ONU sufficient time to receive the downstream frame, including the upstream bandwidth map, perform downstream and upstream FEC as needed, and prepare an upstream response. All ONUs are required to have an ONU response time of 35 $\pm$ 1 μs. Further, the ONU is required to know its response time. The plus 1 μs included in the equations illustrated above are necessary due to avoid the overlapped begin/end zone times.

The general term $requisite\ delay$ refers to the total extra delay that an ONU may be required to apply to the upstream transmission beyond its regular response time. The purpose of the requisite delay is to compensate for variation of propagation and processing delays of individual ONUs, and to avoid or reduce the probability of collisions between upstream transmissions [ITU-T 2014]. During regular operation, the requisite delay is equal to the assigned equalisation delay (EqD_i). The equalisation delay of the ONU is found as:

$$EqD_i = T_{eqd} - RTD_i = T_{eqd} - \left(\Delta_i^{RNG} - \frac{StartTime}{R_{nom}}\right) \tag{7.6}$$

Here, T_{eqd} is the elapsed time between the start of the downstream PHY frame carrying a specific BWMAP and the upstream PHY frame implementing the BWMAP, RTD_i is the first round-trip delay measurement during serial number acquisition, R_{nom} is the nominal upstream line rate in words/μs, $StartTime$ is the start of the upstream PHY frame in ONU's view, Δ_i^{RNG} is the elapsed time between the downstream PHY frame containing the ranging grant and the upstream PHY burst containing the response Registration PLOAM. All details of each stage of communication between ONUs and the OLT are available in [ITU-T 2014] and will not be covered here due to being out of our scope.

Related to the traditional DBA mechanism, Figure 7.3 reports the different steps involved in a DBA process, together with typical latency times, from the moment a packet arrives at the ONU queue, until the moment that ONU is allowed to transmit the packet. Some of the latency times are typical of DBA implementation, while others are experimentally measured in our setup and further discussed in this chapter.

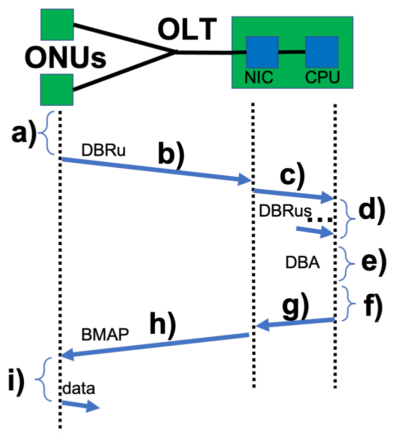


Figure 7.3: Summary of steps involved in the traditional DBA process.

(a) the ONU needs to wait for the opportunity to piggyback the DBRu to an upstream message (between 0 and 125 μs, for an average of 62.5 μs);

(b) the DBRu propagates trough fibre (assume 50 μs for a 10 km distance);

(c) the information travels between the physical card and the virtual process (this only occurs for virtual PON implementations, about 22 μs from our experimental data, i.e. half the round trip time of 41.96 μs);

(d) the DBA process waits for a given time window to receive DBRus from multiple ONUs (between 0 and 125 μs, for an average of 62.5 μs);

(e) the OLT runs the DBA algorithm to calculate the Bandwidth Map (assume DBA calculation time of 77 μs, according to results in figure 7.8 – the difference between second and third bars in the plot);

(f) the Bandwidth Map is included at the beginning of the next downstream frame (between 0 and 125 μs, for an average of 62.5 μs);

(g) the Bandwidth Map travels between the virtual function and the physical card (same consideration as c);

(h) the Bandwidth Map propagates trough fibre (same considerations as b);

(i) the ONU can transmit the data at its allocated time (considering we can schedule low latency allocations at the beginning of a frame, we assume between 0 and 20 μs, for an average of 10 μs).

Considering the calculations provided above, the minimum average time for low latency allocation through a classical DBA mechanism is of 374.5 μs, which is increased to 418.5 μs for a virtual implementation (i.e. considering the additional steps c) and g) above).

Our proposed approach is illustrated in Figure 7.4. The main difference here is that the grant calculations for the Fast Intercept mechanism occur in parallel in the P4 NIC while waiting for a BWMAP to arrive from the CPU VNF. We assume that the T-CONT ID is used to determine whether an allocation requires low-latency support.

As soon as it arrives, the Fast Intercept mechanism modifies the BWMAP to include the latest arrival low-latency grant requests.

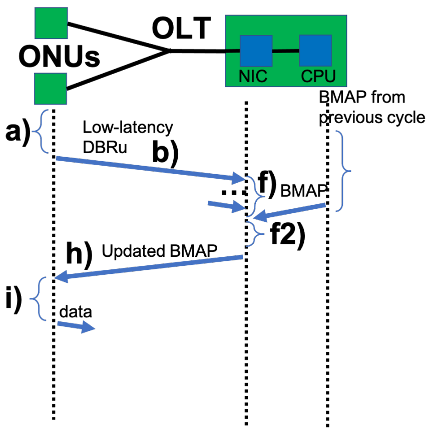


Figure 7.4: Proposed dual-DBA process with P4 in-NIC computing.

Thus, with respect to the stages above (we adopt the same step labels for ease of comparison), we have: a) The ONU needs to wait for the opportunity to piggyback the DBRu to an upstream message; b) the DBRu propagates through fibre; f) the next Bandwidth Map arrives at the NIC (between 0 and 125 μs, for an average of 62.5 μs), in parallel the NIC calculates the BWMAP update for the low latency grants (7.55 μs according to our results in section 3); f2) the NIC updates the BWMAP including the low latency grant allocations (2.5 μs according to our results in section 3); h) the BWMAP propagates through fibre; i) the ONU can transmit the data at its allocated time.

To reduce this part of latency, we have to short some steps, and that is the basis of our proposal of a Fast Intercept mechanism, that takes account of a programmable network device: a SmartNIC, which supports the high-level P4 language for deploying applications into this network card. Using it, we can build a mechanism that splits the upstream DBA scheduling into two parts, as shown in Figure 7.5. At the top of the diagram, using a standard DBA application running on a virtual machine, that operates according to standard DBA procedures, and a Fast Intercept mechanism on the bottom, that works independently from the first part and runs on the SmartNIC. Its basic functionality is: (i) to check the DBRu packets, storing their content for further updating the bandwidth maps; (ii) To select the low and normal latency data, diverting them to the high-level vDBA and update bandwidth maps, before sending them to ONUs.

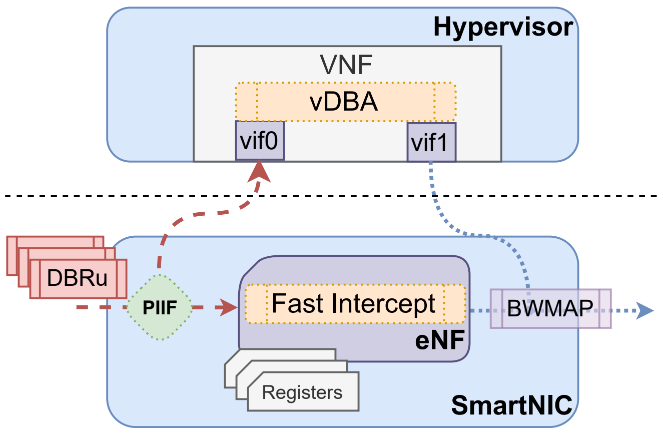


Figure 7.5: Multi-level DBA scheduling provided by In-Network Computing.

It is important to emphasise that the realisation of Multi-level DBA scheduling is enabled via **PIaFFE Intercepting and Forwarding** (PIIF) element, discussed in Chapter 3. In this use-case, it is responsible for a) intercepting, inspecting and directing the network traffic between the two levels: software and hardware, finding the packets with the low-latency tag; b) storing their DBRu requests into an internal buffer (registers) to be; c) further updated into grants inside the BWMAP, allowing the Fast Intercept mechanism to be implemented using an in-network processor using PIaFFE framework as an enabler.

7.2.3 Implementation details

Upstream data paths are typically Time Division Multiplexed requiring coordination of talk interval using a single centralised, remotely located, controller located upstream. We see this in the DBA of PONs, the mac scheduler of LTE radio networks. In classical implementations, these schedulers are embedded in hardware, making modification of the algorithm difficult or impossible to achieve. Classical DBA is calculated in proprietary hardware, collocated with OLT. Round trip transmission latency is around tens and hundreds of nanoseconds. In newer implementations that are seen in 5G networks, we see schedulers implemented as virtualised network functions running on host devices. While making the algorithms configurable and flexible, it does introduce latency.

The Fast Intercept mechanism implements a rescheduling mechanism, as depicted by Figure 7.6, which is capable of altering the original BWMAP coming from the vDBA host, reordering the allocation structures to insert the low-latency DBRu request into the first position of the grant. Here, we assume that the DBRu request is less or equal to the original pre-allocation, allowing the exchange of allocation structures positions and/or values without a complex calculation or reallocation from another allocation structure(s).

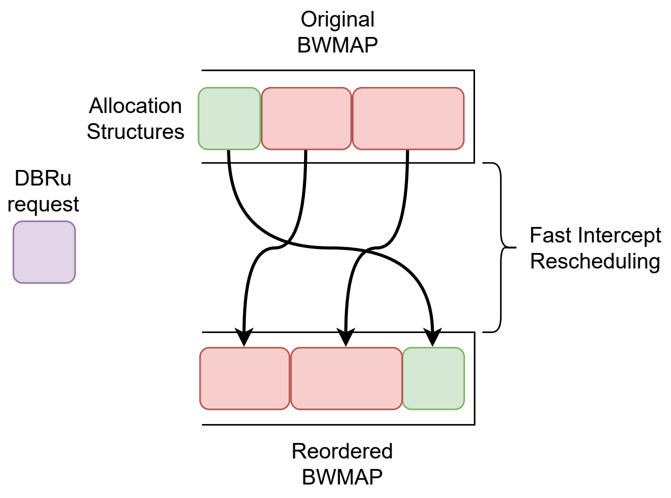


Figure 7.6: Fast Intercept rescheduling mechanism.

Thus, in this chapter, we propose a performance analysis of the use of the P4 language in a programmable NIC to implement complex algorithms, in particular, a partial DBA scheduler algorithm that relies on preempting bandwidth allocation for ONT/T-CONTs related to LTE traffic, modifying them directly on a P4-enabled hardware device, to overcome the total pathway budget of the network function virtualisation environment employed to deploy the Virtual DBA.

Based on our previous work of embedding VNFs on P4-enabled devices, we developed a P4 embedded network function (eNF) on a Netronome SmartNIC that processes the BWMAPs and DBRus data structures. Figure 7.7 shows the P4 Parser and Ingress pipelines on the SmartNIC. Our eNF classifies and timestamps the DBRUs and BWMAPs, and then stores their header structures in registers for later processing by the main DBA algorithm logic, in the Ingress pipeline.

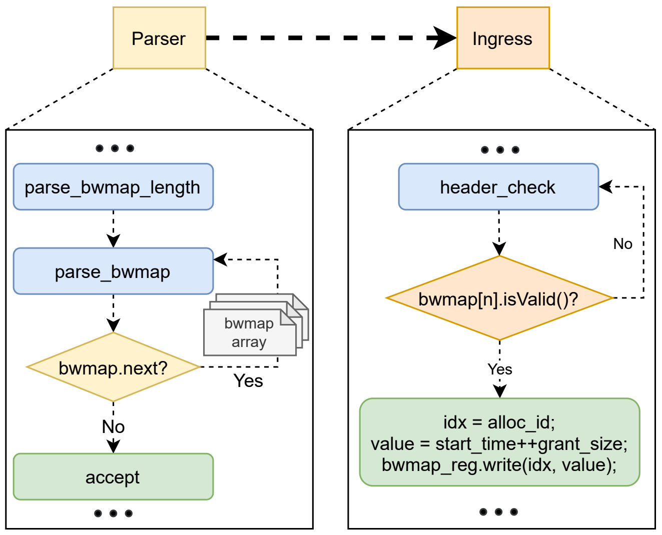


Figure 7.7: P4 Parser and Ingress pipelines

For each upstream burst arriving at the SmartNIC, the eNF checks the T-CONTs and stores their content (DBRus) in a data structure for subsequent checking in the downstream direction. Thus, in the downstream direction, the eNF analyses the data stored previously run a fast intercept rescheduling DBA algorithm on the network hardware and updates the upcoming BWMAP accordingly.

7.3 Experimental evaluation

Before going into further experimentation details, one consideration we want to make is whether allocating part of the BWMAP for low latency applications could be considered wasteful, in case there are not enough applications requiring it on any given frame. In our implementation we easily solve this issue: as all DBRu request always passes through the SmartNIC, the Fast Intercept mechanism can fill in spare allocations using requests from other lower priority grant requests (i.e., that do not have low latency constraints), to avoid wasting capacity.

7.3.1 Experimental results

Figure 7.8 reports both DBA computation time and transmission time between SmartNIC and CPU. These affect the steps c), e) and g) shown in Figure 7.3. The first bar in Figure 7.8 reports the time required to send DBRus from NIC to CPU, for the DPU to calculate the BWMAP and for this to be sent from the CPU to the NIC. This is a baseline scenario, where no specific optimisation is carried out, using Linux Netdev, and shows the longest time of about 393 μs. The second bar represents the same process, but when implemented through our optimised Intel DPDK solution for virtual PON. This bypasses the Linux network stack and runs in user space in poll mode, and it reduces the timing to 119.51 μs. The third bar shows the time required for a round trip time between NIC and CPU when using DPDK (41.96 μs). From the difference, we can infer the DBA processing time in the DPDK implementation of 77.55 μs.

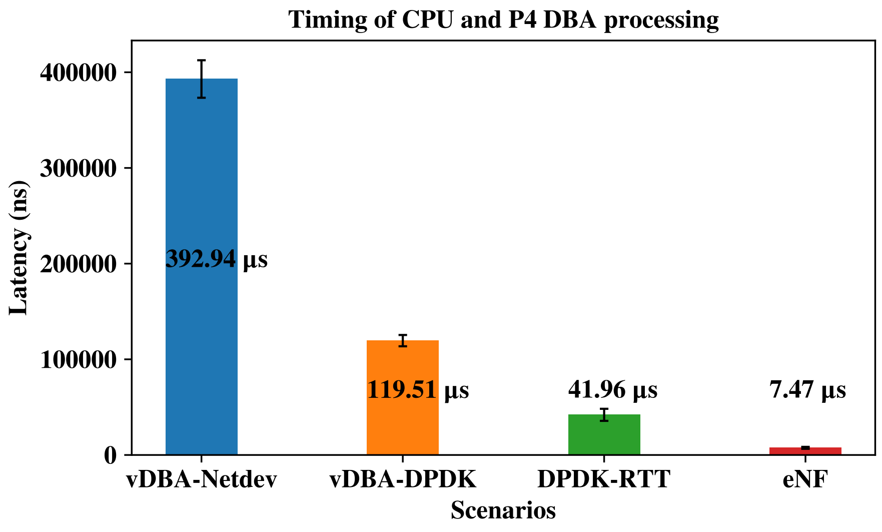


Figure 7.8: Comparison between multiple approaches to calculating the DBA algorithm.

The fourth bar finally, shows the time required by the NIC to operate the Fast Intercept mechanism, inclusive of grant calculation and update of the BWMAP. As this does not require any data transfer between NIC and CPU and allows fast P4 processing for the fast DBA, it can be computed to be only 7.47 μs. It should also be noted that since the PON is a synchronous TDM technology, the eNF knows exactly when the BWMAP will be received from the CPU. Thus it can initiate the fast DBA calculation 8 μs before it receives the BWMAP. In this case, the only additional time will be that required to modify the BWMAP. This is reported in Figure 7.9, which shows the breakdown of the eNF computation time.

As discussed before, the algorithm modifies the $start\_time$ and $grant\_size$ in the BWMAP, rescheduling the allocation structures and reordering the BWMAP into the network hardware. The process then follows the following 4 steps, as depicted by Figure 7.9:

1. The eNF identifies the DBRus in transit into a P4 register, which is a fast memory on the network hardware.
2. If the DBRu **does not** include a low latency grant request, the DBRu packets continue to their destination (the DBA in the CPU), without any modification.
3. If the DBRu **does** include a low latency grant request, the eNF prepares the allocation that will be used to modify the incoming BWMAP, applying any required modification to the grant_size fields.
4. When the next BWMAP arrives to the NIC from the CPU, the P4 process modifies this accordingly to include the low latency allocations, before forwarding it to the ONUs.

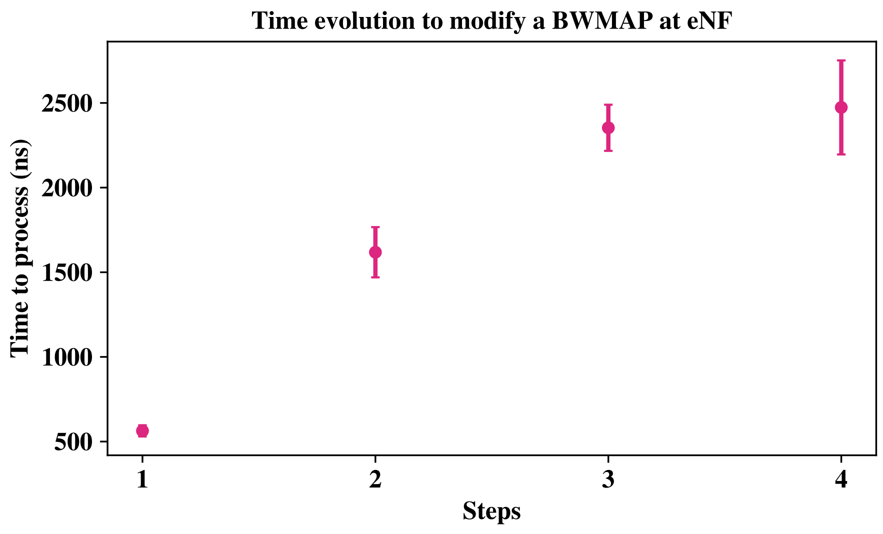


Figure 7.9: Dissecting the algorithm: times of each step.

Summarising the experimental results in Figure 7.8 and 7.9, re-iterating the overall DBA calculation times reported in the previous section and Figure 7.3 and 7.4, we have calculated that a standard DBA mechanism (assuming the DBA is calculated for each service interval), would have an average latency of 374.5 μs and 418.5 μs, respectively for an OEM and virtual implementation of a PON DBA. On the other hand, our proposed mechanism, split into a CPU and SmartNIC implementation can provide an average latency of 237.5 μs. This is a significant reduction of 37 % and 43 %, for upstream PON latency, which can enhance the PON support for low latency applications.

7.4 Chapter remarks

C-RAN transport service on an eCPRI interface requires latency below 250 μs for variable rate traffic. Traditional DBAs provide a high-latency report/grant process, which can be reduced using an eNF to process the low-latency related data, granting the vDBA the ability to tackle the eCPRI requirements.

PIaFFE framework proves that can be used to port a solution using in-network processing, leveraging the packet processing for Passive Optical Networks in lower network layers using the capabilities of the network hardware exposed by the P4 language to build complex functions inside a narrow programmable network card.

We have trade-offs using a NIC to process a high volume of data, due to their hardware processing limitations (CPU, memory), as well as the restrictions about float point operations and lack of loops related to the P4 language plus SmartNIC architecture, restricting the development of more complex calculations using an in-network processor. However, the overall processing is improved, taking advantage of the Fast Intercept mechanism which builds an opportunity window inside the in-network processor, allowing the update of grants for ONUs during the same TDM-PON cycle, improving the latency and shorting the path to process network data directly into the programmable network interface card.

8. Conclusion and Future Works

In this concluding chapter, we summarise this thesis and the work hereby presented to discuss the implications of our research. We then give our view concerning possible research directions that may emerge from our work and how other techniques can extend the network programmability using the PIaFFE Framework. Finally, we finish the thesis with our concluding remarks.

8.1 Thesis Summary

The thesis' more general implications just scratch the surface of the debate that aims to refute the widely held belief that network programmability should be considered as an enabler for offloading data the edge. Given that, we have explored various aspects of hardware and software co-design for developing and deploying network and non-network applications onto smart network interface cards, also known as SmartNICs, which led us to tackle the following questions: "How to exploit co-design, i.e., between programmable network devices and end-host processing, opened by SmartNICs to accommodate diverse logic and their requirements considering different applications, including malicious ones, within the edge computing paradigm?"

In order to fully meet the co-design capability for the deployment of network applications, we claim that the development process must be supported by a functional decomposition approach, aligning with **Hypothesis H1**. Therefore, in Chapter 4 explore experimentally this affirmative and also the questions mentioned above. The merit of our approach is in creating a mechanism for steering traffic data between the hardware and software levels (SmartNIC and VNF), offloading network computation when providing a technique to select the desired data to be processed in a run-time fashion.

Extending the programmability level, we develop Chapter 5, now envisioning the creation of a network function able to intercept and interact with the cloud robotics use-case, matching with **Hypothesis H2** and showing the vulnerabilities of a programmable data plane using a Robot Operating System (ROS), which is currently a widely used framework for robot development, casting light in how attacks that are well described in the literature can be refactored to be launched from the network itself. Moreover, in Chapter 6 we discuss a possible solution for ensuring data integrity by implementing a cryptographic hash function based on Secure Hash Algorithm 2 (SHA-2), complementing the security constraints raised in the previous chapter.

The fragment of the work presented in Chapter 7 explores the development and deployment of an application using in-network computing and Passive Optical Networks using the Virtual DBA [Ruffini et al. 2020] as the basis, tackling the application's low latency requirements – named as Fast Intercept mechanism – after they are forwarded to a virtual network function, reacting to these requests directly on a network programmable hardware based on packet inspection to determine when these low latency requirements are needed. The Chapter also ensures the **Hypothesis H2**, once a new complex and critical application took advantage of such mechanism for intercepting and interacting with raw packets.

Each use-case, listed in the thesis in chapter format, served to improve the prototyping techniques of our PIaFFE framework for embedding applications into SmartNICs and performing with traditional applications running on the server side. Starting with the NFV approach for creating or supporting traditional network applications, moving to security use-cases over cloud robotics, exposing the concerns when using a programmable network device in such scenarios, as well as proposing the offloading of cryptographic hash functions into the data plane. Finishing, the work shows the evolution towards non-network applications, such as the PON schedule mechanism, using the capabilities of the programmable data plane to process, modify and generate network data in a transparent manner for the legacy applications running inside commodity servers.

8.2 Future Works

Notwithstanding our efforts to limit the scope of this work, our research activities have inevitably touched on a diversity of other topics. Hence, future works involve exploring programmable data planes to enable a new range of modern and innovative applications using the PIaFFE Framework, such as:

- **Cryptographic applications**, as shown by Chapter 6, we have room for exploring cryptographic algorithms using in-network computing to alleviate the packet processing by the host, as this kind of algorithm is considered compute-intensive. We can build a cryptographic application that can be used to encrypt/decrypt data directly on the data plane, porting existing cryptographic libraries and methods for running inside this hardware. Thus, using PIaFFE for prototyping and embedding cryptographic hash functions we can effectively maximise the network resource utilisation and alleviate the host side execution by offloading packet processing to a SmartNIC, co-executing the application in a transparent fashion.

- **Data security and authenticity**, which brings the tampering of the headers or even the data from the network as one of the challenges for developing in-network applications. To solve this problem, it is possible to explore well-known techniques for checking and encrypting data over the network, creating public and private keys to be verified between data planes, and avoiding the host's CPU usage when computing these data or creating a co-execution environment between hardware and software, optimising the utilisation of both.

- **Passive Optical Networks**, improving the Fast Intercept mechanism to afford a more realistic scenario should be a good starting point, as we can explore new targets for the development and deployment of applications using PIaFFE Framework, such as programmable switches, which fits better with PON network operators that employ these network devices in the "real world". This step will bring new challenges, and also, new opportunities for embedding applications into a network programmable hardware, extending the PIaFFE Framework definition to new SDK tools from specific targets and moving forward the extensibility and applicability of the framework for developing new embedded network functions outside of a commodity host.

- **In-band network telemetry for cloud robotics**, despite the possibility of collecting network data and bundling with the network packets in the P4 language using the PIaFFE Framework, we can explore more ways to employ it, providing per-packet latency/jitter performance and generic robot fleet information, such as embedded sensor data and feedback control. Since the packet header will carry on such information, we can avoid the OS' layers when processing data inside the data plane, improving latency when reacting to some robot events which require them.

8.3 Final Remarks

From a broader perspective, there is a fascinating list of research questions around our proposals. For example, one great challenge of modern networks is selecting the best place to execute a network application in the function of the nature of the traffic data flowing through the stacks of OS and hardware. Whereas, in recent years, emerging networking programming languages and architectures, such as

P4 and PISA, have created unprecedented opportunities for rapidly prototyping disruptive solutions in programmable data planes. In addition, the potential impacts of providing a framework aligned with multiple different targets are far-reaching the principle of separating control and data planes, providing tools and methods to develop and deploy applications across multiple platforms, including programmable switches.

Therefore, even after all these years in which we have conducted this work, in-network computing and data plane customisation are still exciting research fields. However, various properties are missing to be explored in the PIaFFE Framework. It was shown as a viable paradigm to enable and empower a wide range of network applications, which was not the case when we started with this line of research. Finally, we conclude by reaffirming our belief that the work present in this thesis provides solid guidelines and interesting insight to foster expressive and flexible research and experimentation using programmable network devices.

Bibliography

[Ahmadi 2019] Ahmadi, S. (2019). *5G NR: Architecture, technology, implementation, and operation of 3GPP new radio standards*. Academic Press. viii, 21

[Alliance 2015] Alliance, N. (2015). Further study on critical c-ran technologies. *Next Generation Mobile Networks*. 102

[Alvarez et al. 2016] Alvarez, P., Marchetti, N., and Ruffini, M. (2016). Evaluating dynamic bandwidth allocation of virtualized passive optical networks over mobile traffic traces. *Journal of Optical Communications and Networking*, 8(3):129–136. 20

[Bosshart et al. 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, page 87–95. 84

[Canada et al. 2017] Canada, B. et al. (2017). Network Functions Virtualisation (2017). http://portal.etsi.org/NFV/NFV_White_Paper.pdf. ix, 58, 64

[CAVIUM 2019] CAVIUM (2019). Cavium LiquidIO SmartNICs. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/. 10, 25

[Cesen et al. 2020] Cesen, F. E. R., Csikor, L., Recalde, C., Rothenberg, C. E., and Pongrácz, G. (2020). Towards low latency industrial robot control in programmable data planes. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 165–169. 72

[Chanclou et al. 2013] Chanclou, P., Pizzinat, A., Le Clech, F., Reedeker, T.-L., Lagadec, Y., Saliou, F., Le Guyader, B., Guillo, L., Deniel, Q., Gosselin, S., Le, S. D., Diallo, T., Brenot, R., Lelarge, F., Marazzi, L., Parolari, P., Martinelli, M., O'Dull, S., Gebrewold, S. A., Hillerkuss, D., Leuthold, J., Gavioli, G., and Galli, P. (2013). Optical fiber solution for mobile fronthaul to achieve cloud radio access network. In *2013 Future Network Mobile Summit*, pages 1–11. 20

[Cheng et al. 2014] Cheng, N., Gao, J., Xu, C., Gao, B., Liu, D., Wang, L., Wu, X., Zhou, X., Lin, H., and Effenberger, F. (2014). Flexible twdm pon system with pluggable optical transceiver modules. *Opt. Express*, 22(2):2078–2091. 20

[Choi et al. 2019] Choi, S., Shahbaz, M., Prabhakar, B., and Rosenblum, M. (2019). Lambda-nic: Interactive serverless compute on programmable smartnics. *arXiv preprint arXiv:1909.11958*. viii, 34

[Clark et al. 2017] Clark, G. W., Doran, M. V., and Andel, T. R. (2017). Cybersecurity issues in robotics. *2017 IEEE Conference on Cognitive and Computational Aspects of Situation Management, CogSIMA 2017*. 75

[Cloud 2016] Cloud, C. G. (2016). Cisco Global Cloud Index: Forecast and Methodology. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html. 10

[Corporation 2020] Corporation, I. (2020). Intel Accelerator Functional Unit (AFU) developer's guide. https://www.intel.com/content/www/us/en/docs/programmable/683129/1-2-and-2-0-1/about-this-document.html. 35

[CVE 2020] CVE (2020). CVE-2020-9115. Available from MITRE, CVE-ID CVE-2020-9115. 75

[da Silva et al. 2018] da Silva, M. V. B., Jacobs, A. S., Pfitscher, R. J., and Granville, L. Z. (2018). Ideafix: Identifying elephant flows in p4-based ixp networks. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE. 84

[Dang et al. 2015] Dang, Q. H. et al. (2015). Secure hash standard (shs), standard fips 180-4. *National Institute of Standards and Technology*. 25

[Dargahi et al. 2017] Dargahi, T., Caponi, A., Ambrosin, M., Bianchi, G., and Conti, M. (2017). A Survey on the Security of Stateful SDN Data Planes. *IEEE Communications Surveys and Tutorials*, 19(3):1701–1725. 75

[Das and Ruffini 2020] Das, S. and Ruffini, M. (2020). Pon virtualisation with east-west communications for low-latency converged multi-access edge computing (mec). In *2020 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE. 20

[Das et al. 2020] Das, S., Slyne, F., Kaszubowska, A., and Ruffini, M. (2020). Virtualized EAST-WEST PON architecture supporting low-latency communication for mobile functional split based on multiaccess edge computing. *Journal of Optical Communications and Networking*, 12(10):D109–D119. 10, 21

[DC 2015] DC, C. (2015). The New Need for Speed in the Datacenter Network. https://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-734328.pdf. 10

[Dearle 2007] Dearle, A. (2007). Software deployment, past, present and future. In *FoSE 2007: Future of Software Engineering*, pages 269–284. 55

[Dieber et al. 2020] Dieber, B., White, R., Taurer, S., Breiling, B., Caiazza, G., Christensen, H., and Cortesi, A. (2020). *Penetration Testing ROS*, pages 183–225. Springer International Publishing, Cham. 77

[DiLuoffo et al. 2018] DiLuoffo, V., Michalson, W. R., and Sunar, B. (2018). Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15(3):1729881418770011. 77

[Eran et al. 2019] Eran, H., Zeno, L., Tork, M., Malka, G., and Silberstein, M. (2019). NICA: An Infrastructure for Inline Acceleration of Network Applications. *ATC*. viii, 35

[ETSI NFV ISG 2014] ETSI NFV ISG (2014). NFV 002 V1.2.1. Network Functions Virtualisation (NFV); Architectural Framework. 23, 40

[Firestone 2017] Firestone, D. (2017). Hardware-Accelerated Networks at Scale in the Cloud. https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf. 10

[Firestone et al. 2018] Firestone, D., Putnam, A., et al. (2018). Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. viii, 31, 32

[Glebke et al. 2019] Glebke, R., Krude, J., Kunze, I., Rüth, J., Senger, F., and Wehrle, K. (2019). Towards executing computer vision functionality on programmable network devices. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, page 15–20, New York, NY, USA. Association for Computing Machinery. 72

[Grant et al. 2020] Grant, S., Yelam, A., Bland, M., and Snoeren, A. C. (2020). SmartNIC Performance Isolation with FairNIC : Programmable Networking for the Cloud. *Sigcomm '20*. viii, 38

[Grigoryan et al. 2019] Grigoryan, G., Liu, Y., and Kwon, M. (2019). Iload: In-network load balancing with programmable data plane. In *Proceedings of the*

15th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT ’19 Companion, page 17–19, New York, NY, USA. Association for Computing Machinery. 84

[Han et al. 2015] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97. viii, 23, 24

[ITU-T 2014] ITU-T (2014). G.987.3:10-Gigabit-capable passive optical networks (XG-PON): Transmission convergence (TC) layer specification. *Itu-T G-Series Recommendations*, 2.0:1–146. viii, 21, 104

[Jin et al. 2017] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, page 121–136, New York, NY, USA. Association for Computing Machinery. 84

[Johnson 1997] Johnson, R. E. (1997). Components, frameworks, patterns. *Proceedings of the 1997 Symposium on Software Reusability, SSR 1997*, pages 10–17. 46

[Karrakchou et al. 2021] Karrakchou, O., Samaan, N., and Karmouch, A. (2021). Ep4: An application-aware network architecture with a customizable data plane. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. viii, 40

[Kim et al. 2018] Kim, J., Smereka, J. M., Cheung, C., Nepal, S., and Grobler, M. (2018). Security and performance considerations in ROS 2: A balancing act. 77

[Kirschgens et al. 2018] Kirschgens, L. A., Ugarte, I. Z., Gil-Uriarte, E., Rosas, A. M., and Vilches, V. M. (2018). Robot hazards: from safety to security. *CoRR*, abs/1806.06681. 75

[Kohler et al. 2000] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297. 30

[Kun et al. 2016] Kun, B. L., Layong, T., Luo, L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. (2016). ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. *Sigcomm*. viii, 30

[Kunze et al. 2021] Kunze, I., Glebke, R., Scheiper, J., Bodenbenner, M., Schmitt, R., and Wehrle, K. (2021). Investigating the applicability of in-network computing to industrial scenarios. In *Proceedings of the 4th IEEE International Conference on Industrial Cyber-Physical Systems*. IEEE. 72

[Kurose and Ross 2016] Kurose, J. F. and Ross, K. W. (2016). *Computer Networking: A Top-Down Approach*. Pearson, Boston, MA, 7 edition. 75

[Le et al. 2016] Le, N. T., Hossain, M. A., Islam, A., Kim, D. Y., Choi, Y. J., and Jang, Y. M. (2016). Survey of promising technologies for 5g networks. *Mobile Information Systems*, 2016. 19

[Liu et al. 2019] Liu, M. et al. (2019). Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM ’19, page 318–333, New York, NY, USA. Association for Computing Machinery. viii, 35, 36

[Mafioletti et al. 2020] Mafioletti, D. R., Dominicini, C. K., Martinello, M., Ribeiro, M. R. N., and Villaça, R. d. S. (2020). Piaffe: A place-as-you-go in-network framework for flexible embedding of vnfs. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–6. 70, 71, 74, 87

[Martino and Cilardo 2020] Martino, R. and Cilardo, A. (2020). SHA-2 Acceleration Meeting the Needs of Emerging Applications: A Comparative Survey. *IEEE Access*, 8:28415–28436. 25

[Martins et al. 2014] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. (2014). ClickOS and the Art of Network Function Virtualization. In *NSDI*, NSDI’14, pages 459–473, Berkeley, CA, USA. USENIX Association. 23

[Mell et al. 2011] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing. `http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf`. 22

[MELLANOX 2019] MELLANOX (2019). Mellanox Innova Ethernet Adapter. `http://www.mellanox.com/products/smartnic/`. 10

[Mello et al. 2019] Mello, R. C., Jimenez, M. F., Ribeiro, M. R. N., Laiola Guimarães, R., and Frizera-Neto, A. (2019). On Human-in-the-Loop CPS in Healthcare: A Cloud-Enabled Mobility Assistance Service. *Robotica*, pages 1–17. 10, 71

[Mirjalili and Lenstra 2008] Mirjalili, S. H. and Lenstra, A. K. (2008). Security observance throughout the life-cycle of embedded systems. *Proceedings of the 2008 International Conference on Embedded Systems and Applications*, pages 186–192. 75

[Nencioni et al. 2018] Nencioni, G., Garroppo, R. G., Gonzalez, A. J., Helvik, B. E., and Procissi, G. (2018). Orchestration and Control in Software-Defined 5G Networks: Research Challenges. *Wireless Communications and Mobile Computing*, 2018:1–18. 19

[NETRONOME 2019] NETRONOME (2019). Netronome Agilio SmartNIC. https://www.netronome.com/products/agilio-cx/. 10, 25, 64, 85, 91

[Nour et al. 2020] Nour, B., Mastorakis, S., and Mtibaa, A. (2020). Compute-less networking: Perspectives, challenges, and opportunities. *IEEE Network*, 34(6):259–265. 10, 71

[O-RAN 2020] O-RAN (2020). O-RAN Fronthaul Cooperative Transport Interface Transport Control Plane Specification 1.0 - April 2020 (O-RAN.WG4.CTI-TCP.0-v01.00). https://www.o-ran.org/specifications/. 100

[Osinski et al. 2019] Osinski, T., Tarasiuk, H., Rajewski, L., and Kowalczyk, E. (2019). DPPx: A P4-based Data Plane Programmability and Exposure framework to enhance NFV services. *Proceedings of the 2019 IEEE Conference on Network Softwarization: Unleashing the Power of Network Softwarization, NetSoft 2019*, pages 296–300. viii, 40, 41

[Panicucci et al. 2020] Panicucci, S., Nikolakis, N., Cerquitelli, T., Ventura, F., Proto, S., Macii, E., Makris, S., Bowden, D., Becker, P., O'Mahony, N., Morabito, L., Napione, C., Marguglio, A., Coppo, G., and Andolina, S. (2020). A cloud-to-edge approach to support predictive analytics in robotics industry. *Electronics*, 9(3). 10

[Phothilimthana et al. 2018] Phothilimthana, P. M. et al. (2018). Floem: a programming system for nic-accelerated network applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 663–679. viii, 33

[Pocovi et al. 2018] Pocovi, G., Shariatmadari, H., Berardinelli, G., Pedersen, K., Steiner, J., and Li, Z. (2018). Achieving ultra-reliable low-latency communications: Challenges and envisioned system enhancements. *IEEE Network*, 32(2):8–15. 100

[Pontarelli et al. 2019] Pontarelli, S., Bifulco, R., Bonola, M., Cascone, C., Spaziani, M., Bruschi, V., Sanvito, D., Siracusano, G., Capone, A., Honda, M., et al. (2019). {FlowBlaze}: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548. viii, 37

[Quigley et al. 2009] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. (2009). Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3. 75, 76

[Rahman Chowdhury et al. 2019] Rahman Chowdhury, S., Salahuddin, M. A., Limam, N., and Boutaba, R. (2019). Re-Architecting NFV Ecosystem with Microservices: State-of-the-Art and Research Challenges. *IEEE Network*, pages 1–9. ix, 12, 26, 60, 61

[Richardson 2018] Richardson, C. (2018). *Microservices patterns: with examples in Java*. Simon and Schuster. 48, 87

[Rivera et al. 2019] Rivera, S., Lagraa, S., Nita-Rotaru, C., Becker, S., and State, R. (2019). Ros-defender: Sdn-based security policy enforcement for robotic applications. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 114–119. 76

[Ruffini et al. 2020] Ruffini, M., Ahmad, A., Zeb, S., Afraz, N., and Slyne, F. (2020). Virtual dba: virtualizing passive optical networks to enable multi-service operation in true multi-tenant environments. *J. Opt. Commun. Netw.*, 12(4):B63–B73. 100, 113

[Rüth et al. 2018] Rüth, J. et al. (2018). Towards in-network industrial feedback control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, page 14–19, New York, NY, USA. Association for Computing Machinery. 10

[Sabella et al. 2019] Sabella, D., Sukhomlinov, V., Trang, L., Kekki, S., Paglierani, P., Rossbach, R., Li, X., Fang, Y., Druta, D., Giust, F., et al. (2019). Developing software for multi-access edge computing. *ETSI white paper*, 20:1–38. 19

[Saha and Dasgupta 2018] Saha, O. and Dasgupta, P. (2018). A Comprehensive Survey of Recent Trends in Cloud Robotics Architectures and Applications. *Robotics*, 7(3):47. 71

[Satyanarayanan 2017] Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, 50(1):30–39. 19

[Sherry et al. 2012] Sherry, J. et al. (2012). Making middleboxes someone else's problem. *ACM SIGCOMM Computer Communication Review*, 42(4):13. 23, 58

[Shi and Dustdar 2016] Shi, W. and Dustdar, S. (2016). The Promise of Edge Computing. *Computer*, 49(5):78–81. viii, 19

[Sivaraman et al. 2017] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017). Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 164–176, New York, NY, USA. Association for Computing Machinery. 84

[Sobti and Geetha 2012] Sobti, R. and Geetha, G. (2012). Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*, 9(2):461. 25

[Song et al. 2005] Song, H., Dharmapurikar, S., Turner, J., and Lockwood, J. (2005). Fast hash table lookup using extended bloom filter: An aid to network processing. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, page 181–192, New York, NY, USA. Association for Computing Machinery. 50

[Tarkoma et al. 2012] Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. (2012). Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155. 50

[Tashiro et al. 2014] Tashiro, T., Kuwano, S., Terada, J., Kawamura, T., Tanaka, N., Shigematsu, S., and Yoshimoto, N. (2014). A novel dba scheme for tdm-pon based mobile fronthaul. In *OFC 2014*, pages 1–3. 100

[The P4 Language Consortium 2021] The P4 Language Consortium (2021). P4 Language Specification - May 2021. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html. viii, 27, 28, 29

[The P4.org API Working Group 2020] The P4.org API Working Group (2020). P4Runtime Specification - December 2020. https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html. viii, 39

[The P4.org Architecture Working Group 2020] The P4.org Architecture Working Group (2020). P4-16 Portable Switch Architecture (PSA). https://opennetworking.org/wp-content/uploads/2020/10/P416-Portable-Switch-Architecture-PSA.html. viii, 27, 28

[Turner 2008] Turner, J. M. (2008). The keyed-hash message authentication code (hmac). *Federal Information Processing Standards Publication*, 198(1). 84

[Wang et al. 2020] Wang, J., Cai, R., and Liu, S. (2020). Research on the protection mechanism of cisco IOS exploit. *Journal of Physics: Conference Series*, 1584:012045. 75

[White et al. 2016] White, R., Christensen, D. H. I., and Quigley, D. M. (2016). Sros: Securing ros over the wire, in the graph, and through the kernel. 76

[Xie et al. 2019] Xie, Y., Guo, Y., Mi, Z., Yang, Y., and Obaidat, M. S. (2019). Loosely Coupled Cloud Robotic Framework for QoS-Driven Resource Allocation-Based Web Service Composition. *IEEE Systems Journal*, pages 1–12. 10

[Ángel Manuel Guerrero-Higueras et al. 2018] Ángel Manuel Guerrero-Higueras, DeCastro-García, N., and Matellán, V. (2018). Detection of cyber-attacks to indoor real time localization systems for autonomous robots. *Robotics and Autonomous Systems*, 99:75–83. 75