POLITECNICO DI TORINO

SCUOLA DI DOTTORATO

PhD Course in Computer and Control Engineering – XXVIII cycle

PhD Dissertation

High Performance Network Function Virtualization for User-Oriented Services

Ivano Cerrato
student ID: 199847

Tutor	**Course Coordinator**
dr. Fulvio Risso	prof. Matteo Sonza Reorda

May 2016

Acknowledgements

I would like to thank my supervisor Fulvio Risso, who really helped me during all the three years of my PhD and collaborated on the realization of the work presented in this dissertation.

A special thank is also for my wife Sara, who always supported me, and for all the other guys who worked with me in Politecnico, especially Matteo, Serena, Roberto, Amedeo, Fabio and Francesco.

Summary

The Network Function Virtualization (NFV) paradigm is changing the way in which network services are delivered, as it allows them to experiment the same degree of flexibility and agility already available in the cloud computing world. In fact, NFV proposes to transform those network functions today running on dedicated and often closed appliances (e.g., firewall, wan accelerator) into pure software images, called Virtual Network Functions (VNFs), which can be consolidated and executed on high-volume standard servers.

NFV is mainly seen as a technology targeting network operators, which can exploit the power of IT virtualization to deliver network services with unprecedented agility while achieving a reduction of OPEX and CAPEX. However, also the end users (e.g., xDSL customers) can benefit from NFV, as this would enable them to customize the set of services that are active on their Internet connection; in other words, NFV would enable end users to personalize their Internet experience. Furthermore, by instantiating applications (e.g., parental control) in the network instead then on a specific device, end users can get exactly the same service regardless of the terminal device they are currently using.

This dissertation focuses on the possibility to enable each single end user (and not only network operators) to set up network services by means of NFV. This goal requires to address flexibility and performance issues. Regarding to the former, it is important: *(i)* to support services including both network (e.g., firewall) and cloud (e.g., storage server) applications; *(ii)* to hide to the user defining the service low level details that are not of interest of the user itself. Instead, with respect to performance, services deployed as chains of VNFs should not significantly affect throughput and latency of the Internet connection.

Flexibility aspects are considered in Part I of the dissertation. Particularly, it defines a multilayer architecture that, leveraging different levels of abstraction, deploys generic network services on the computing resources available in the operator network, ranging from the home gateways installed in the customer premises to the data center servers. This part also introduces new models used to describe the service to be instantiated, each one modeling the service with a different level of detail and exploited by a specific layer of the above mentioned architecture. Then, we present three different software architectures of the infrastructure nodes (e.g.,

servers, customer premise equipments) that actually execute the VNFs required by the service, originated by different design principles and exploited to validate the idea in different contexts. Notably, one of such architectures aims at scaling when a huge number (eight thousands) of end users run together their own VNFs on the node; the prototype demonstrates that this objective is feasible and the resulting architecture is scalable.

Part II of the dissertation addresses instead performance problems of NFV, by focusing on new mechanisms to efficiently interconnect VNFs instantiated on the same infrastructure node. Particularly, we consider the two following directions. In the former, we focus on solutions to improve the efficiency of the packet exchange between the virtual switch and each VNF, especially when a massive number of (tiny) VNFs are executed on the same infrastructure node. In the latter, instead, we directly connect VNFs among each other, leaving the virtual switch forwarding plane out of the picture in case some specific conditions are satisfied on the service to be implemented. To conclude, the proposed solutions aim at scaling with the number of VNFs executed concurrently on the same infrastructure node; then, they are well suited to be exploited in the use case presented above, in which several end users deploys (many) VNFs just operating on their own traffic.

Contents

Summary V

List of Tables XI

List of Figures XII

I Delivering user-oriented network services in an NFV scenario 1

1 A scalable and massively multi-tenant platform for user-oriented network services 5

- 1.1 Introduction 5
- 1.2 Related Work 7
- 1.3 Overview 8
- 1.4 FROG design 10
  - 1.4.1 Network tiles 10
  - 1.4.2 Service order 11
  - 1.4.3 Private EXecution environment 14
  - 1.4.4 Data plane, control plane and management server 15
- 1.5 Implementation 16
  - 1.5.1 Virtual switch 16
  - 1.5.2 Packet buffers 18
  - 1.5.3 Private execution environment 20
  - 1.5.4 Packet metadata 22
  - 1.5.5 Distributed architecture 22
- 1.6 FROG and the NFV model 23
- 1.7 Experimental results 24
  - 1.7.1 Single server 25
  - 1.7.2 Multi server 26
  - 1.7.3 The DPI VNF 27
  - 1.7.4 VNFs and execution environments 28

1.7.5 FROG vs NetVM and ClickOS 29
1.8 Conclusion 30

2 Moving applications from the host to the network: experiences, challenges and findings 31
2.1 Introduction 31
2.2 Related Work 32
2.3 The Flexible and pROGrammable edge device 33
2.3.1 Storage Service 34
2.3.2 Remote Execution Environment 35
2.3.3 Communication and authentication 36
2.4 Programming the PEX 37
2.4.1 Callbacks 37
2.4.2 PEX Runtime 37
2.4.3 Packet Dispatcher 38
2.4.4 Storage Interface 38
2.4.5 Remote Application Interface 38
2.4.6 Management Interface 39
2.5 Parental control service 39
2.6 Validation 40
2.6.1 Starting the PEX 40
2.6.2 Accessing the RESTO 41
2.6.3 Exploiting the REX 42
2.7 Conclusion 42

3 Toward dynamic and virtualized network services in telecom operator networks 45
3.1 Introduction 45
3.2 Related work 47
3.3 General architecture 48
3.3.1 Service layer 48
3.3.2 Orchestration layer 50
3.3.3 Infrastructure layer 51
3.4 Data models 51
3.4.1 Service graph 52
3.4.2 Forwarding graph and lowering process 55
3.4.3 Infrastructure graph and reconciliation process 57
3.4.4 Network function template 60
3.5 The validation use case: user-defined network services 62
3.6 Prototype implementation 63
3.6.1 The service layer 64
3.6.2 Global orchestrator 66

3.6.3 The Universal Node . . . 67
3.6.4 The OpenStack-based node . . . 69
3.6.5 Discussion: Openstack-based Node vs. Universal Node . . . 72
3.7 Prototype validation . . . 73
3.7.1 Service overview . . . 73
3.7.2 Performance evaluation . . . 74
3.8 Conclusion and future works . . . 77

II Optimizing packets movement between Virtual Network Functions 79

4 Supporting fine-grained virtual network functions through Intel DPDK 83
4.1 Introduction . . . 83
4.2 DPDK overview . . . 84
4.3 General architecture . . . 85
4.4 Implementations . . . 86
4.4.1 Double buffer . . . 86
4.4.2 Double buffer + semaphore . . . 87
4.4.3 Single buffer towards the vSwitch + semaphore . . . 87
4.4.4 Double buffer + FDIR . . . 88
4.4.5 Isolated buffers + semaphore . . . 89
4.5 Performance evaluation . . . 89
4.5.1 Double buffer . . . 90
4.5.2 Double buffer + semaphore . . . 92
4.5.3 Double buffer + FDIR . . . 93
4.5.4 Isolated buffers + semaphore . . . 93
4.6 Related work . . . 93
4.7 Conclusion . . . 94

5 Efficient data exchange algorithm for chained virtual network functions 97
5.1 Introduction . . . 97
5.2 Related Work . . . 99
5.3 The data exchange architecture . . . 101
5.3.1 Operating context . . . 101
5.3.2 Architecture Overview . . . 101
5.3.3 Execution model . . . 102
5.3.4 Basic algorithm: handling pass-through data . . . 103
5.3.5 Extended algorithm: handling Worker-generated data . . . 109
5.4 Formal verification . . . 112

5.4.1 Properties specification . . . 113
5.4.2 Model details . . . 114
5.4.3 Verification results . . . 115
5.5 Implementation . . . 116
5.6 Experimental results . . . 118
5.6.1 Single chain - Throughput . . . 119
5.6.2 Single chain - Latency . . . 121
5.6.3 Single chain - Comparison with other approaches . . . 121
5.6.4 Single chain - Other tests . . . 123
5.6.5 Multiple chains . . . 124
5.6.6 Network tests . . . 125
5.7 Conclusion . . . 126

6 Transparent optimization of inter-virtual network function communication in Open vSwitch 129
6.1 Introduction . . . 129
6.2 Background . . . 131
6.3 Architecture . . . 132
6.3.1 Detecting p-2-p links . . . 133
6.3.2 Handling the new `ivshmem` device . . . 133
6.3.3 Remapping process . . . 134
6.3.4 Port statistics . . . 136
6.4 Experimental validation . . . 136
6.4.1 Throughput with internal traffic . . . 137
6.4.2 Throughput with physical NICs . . . 138
6.4.3 Latency . . . 138
6.4.4 Establishment time . . . 139
6.5 Related work . . . 140
6.6 Conclusion . . . 140

Conclusions 143

Bibliography 145

Appendix 153

List of Tables

1.1 NFV vs FROG 23
1.2 Protocols and corresponding regex. 28
1.3 “One VNF per PEX” vs “Multiple VNFs per PEX”. 29
2.1 Latency in accessing the RESTO. 42
3.1 Challenges of the considered use case and related solutions. 62
3.2 Universal Node vs OpenStack-based Node. 72
5.1 Algorithm verification. 115

List of Figures

1.1 The FROG operating context: a network edge node and its end users. 9
1.2 High level view of a FROG node. 10
1.3 Possible journey of a packet. 12
1.4 Possible journey of a broadcast/multicast packet. 14
1.5 Data and control plane. 16
1.6 FROG software architecture. 17
1.7 Packet exchange between the fvSwitch and PEX. 19
1.8 Single vs. multi-server architecture. 23
1.9 Throughput with a growing number of PEX. 26
1.10 Latency introduced by the node with a growing number of native PEX. 26
1.11 PEX memory consumption. 27
1.12 Single vs. multi server. 27
1.13 Latency in case of 5 VNFs running: *(i)* in the same PEX, *(ii)* in 5 different PEX. 29
2.1 Overview of the entire system. 33
2.2 Internal view of the Remote Storage Service. 35
2.3 Exploded view of a REX. 36
2.4 Exploded view of a PEX hosting two VNFs. 37
2.5 Starting a PEX with four VNFs. 41
3.1 Deployment of virtual network functions on the telecom operator network. 46
3.2 Overall view of the system, including the two implementations of the infrastructure layer. 49
3.3 Service graph: basic elements and example. 52
3.4 Cascading SGs. 55
3.5 From the SG to the FG: the *lowering process*. 58
3.6 Excerpt of a forwarding graph. 59
3.7 Example of the output of the reconciliation process when mapping a L2 switch functionality in case of two different types of infrastructure nodes. 60
3.8 Example of a VNF template. 61
3.9 Authentication SG and FG. 66

3.10 Logical architecture of the Universal Node. 68
3.11 OpenStack-based Node. 70
3.12 Use case scenario. 74
3.13 Memory consumption. 75
3.14 Performance of the infrastructure layer: (a) ping; (b) file transfer. . . 76
4.1 High-level view of a server with a vSwitch and several VNFs. 86
4.2 Implementation based on a (different) pair of rings shared between the vSwitch and each VNF. 87
4.3 Implementation that exploits the `FDIR` feature. 88
4.4 Throughput with a growing number of VNFs. 91
4.5 Latency introduced by the framework. 91
5.1 Function chains deployed in a server. 98
5.2 Deployment of the algorithm within a server. 102
5.3 Run-time behavior and indexes of the algorithm. 109
5.4 Binding primary buffer - auxiliary buffer. 112
5.5 Throughput of a single function chain with the algorithm presented in this chapter. 120
5.6 Internal throughput of the function chain, with real Workers and a 1M packets in memory. 121
5.7 Latency introduced by the function chain with a growing number of cascading Workers. 122
5.8 Throughput of a single function chain when other data exchange algorithms are used. 122
5.9 Results with a growing number of function chains running in parallel, each one spotting two Workers in cascade. 125
5.10 Results with a function chain of growing length, with the Master accessing to the network. 126
6.1 Traffic crossing several VNFs: (a) the “abstract” service graph; (b) its implementation on a server. 130
6.2 Sharing DPDK data structures between OvS and VMs. 132
6.3 Different implementations for the `dpdkr` port. 132
6.4 `ivshmem` device for port remapping. 134
6.5 Test setup. 137
6.6 Memory-only traffic. 137
6.7 Traffic using physical NICs. 137
6.8 # of cores required/assigned during the tests. 138
6.9 Latency when physical NICs are involved. 139
6.10 Time required to establish a direct connection. 139

Part I

Delivering user-oriented network services in an NFV scenario

Overview

The way network services are delivered is going to dramatically change in the next few years thanks to the Network Function Virtualization (NFV) [43] paradigm, which allows the network services to experiment the same degree of flexibility and agility already available in the cloud computing world. In fact, NFV proposes to transform network functions that today run on dedicated appliances (e.g., firewall, WAN accelerator) into a set of software images that can be consolidated into high-volume standard servers, hence replacing dedicated middleboxes with virtual machines implementing those Virtual Network Functions (VNFs).

NFV is mainly seen as a technology targeting network operators, which can exploit the power of the IT virtualization (e.g., cloud and datacenters) to deliver network services with unprecedented agility and efficiency and at the same time achieve a reduction of OPEX and CAPEX. However, also the *end users* (e.g., xDSL customers) can benefit from NFV, as this would enable them to customize the set of services that are active on their Internet connection; in other words, NFV would enable end user to personalize their Internet experience.

The possibility to enable multiple players, including the end users, to deploy network services by means of the NFV paradigm is explored in this part of the dissertation.

Chapter 1 presents FROG, a software architecture allowing both the *end users* and other players such as service providers to install and operate their own VNFs on a network edge node. Particularly, end users connected to the node can create their customized network services that are then applied to their own traffic independently from the physical terminal in use (e.g., smarthphone, tablet, ect.). Numbers from ISPs indicate that, on the same Broadband Remote Access Server (BRAS) (which may be a deployment scenario for FROG), the termination of 20K ADSL lines is not uncommon. Then, the proposed architecture needs to guarantee an adequate level of performance even when a huge number of end users run together their own VNFs on the node. The prototype demonstrates that this objective is feasible and the resulting architecture is scalable; in addition, such an architecture guarantees traffic isolation among VNFs deployed by different players.

Chapter 2 validates the FROG platform from the point of view of the service developer; in particular, the chapter presents the API exported to the developers, as well as a complex parental control service built on top of such an API.

Chapter 3 extends the FROG concept by proposing a multilayer architecture that, leveraging different levels of abstraction, can orchestrate and deploy generic network services on the whole network of a telecom operator, ranging from the home gateway installed in the customer premises to the data center servers. This architecture is then exploited to deliver generic services in presence of multiple concurrent players (e.g., network operators, *end users*), leveraging a new simple data model.

Particularly, the chapter proposes a description-based approach allowing the deployment of agile, implementation-independent and high-level network services over a distributed set of resources. The resulting data model can abstract generic services, including both middlebox-based (e.g., firewalls, NATs, etc.) and traditional LAN-based ones (e.g., a BitTorrent client). Finally, two distinct prototypes, originated by different design principles, are implemented in order to validate the proposal with the aim of demonstrating the adaptability of the approach in different contexts.

Chapter 1

A scalable and massively multi-tenant platform for user-oriented network services

1.1 Introduction

Thanks to the arising of Network Function Virtualization (NFV) [43] it is possible, perhaps for the first time, to influence network operations through software applications developed by third parties; however, at the best of our knowledge, currently a few players, namely network manufacturers and network operators seem to be allowed to create, install and operate Virtual Network Functions (VNFs) on the network nodes.

Going against this trend, this chapter[1] presents **FROG**, a **Flexible and pROGrammable edge device** that offers to different players, including *end users*, the possibility to customize the behavior of the device itself through the deployment of their own virtual network functions. In fact the end users, with their imagination, are the ones that drove the innovation in the PC and smartphone markets with the creation of many unexpected applications, and we expect them to be the ones that will contribute most to network evolution. In this respect, we envision for Network Service Providers (NSPs) the possibility to evolve in *infrastructure providers* (a sort of new *Network IaaS*), offering to multiple players a pipe that transports bit (the network) and a programmable platform where those bits can be processed and even modified in transit.

FROG classifies the players enabled to deploy VNFs into two categories: *(i)* *end users*, i.e., the ADSL customers who connect to the Internet through a FROG

[1]The work of this chapter is partially published in [79] and partially described in the master thesis of Luca Capano, who collaborated in the development of the prototype.

node and that can deploy VNFs only on this particular FROG instance; *(ii)* other entities such as Internet Service Providers (ISPs) and content providers, which are not directly connected to FROG, but that are anyway enabled to deploy VNFs on the node itself. This partitioning represents one of the key points of our framework, and it is exploited to decide the order in which a packet belonging to multiple players is processed by the VNFs they have deployed on the FROG node.

It is worth noting that players such as ISPs may be interested to exploit FROG to run, as software images, those applications traditionally executed in dedicated appliances or proprietary boxes, such as NAT and WAN accelerators. Instead the end users may be more interested in moving on FROG (and then into the network) those applications today executed in their many devices (e.g., personal firewall, parental control), in order to obtain exactly the same service regardless of the device they are currently using.

The fact that several players are enabled to deploy their own VNFs on FROG may result in a very huge number (even thousands) of VNFs executed simultaneously on the node. The support of this massive number of VNFs is not trivial, and it requires to address several issues.

First of all, *VNFs cannot be executed in full fledged virtual machines*, because they would be too expensive in terms of hardware requirements, namely memory and processing power. Then FROG defines a lightweight container, called **Private EXecution environment (PEX)** (see Figure 1.2), which is in charge of executing *all the VNFs belonging to the same player*; in the remainder of this chapter we will demonstrate that a single FROG node implemented on general purpose hardware can execute thousands of VNFs at the same time (eight thousands in our setup). Moreover, to support even more VNFs[2], we designed a *distributed version of the framework*, in which a single FROG deployment actually consists of a cluster of servers. It is worth noting that the multi-tenancy of FROG imposes that the framework guarantees isolation among VNFs, so that a malicious application installed by user *A* cannot interfere with applications belonging to user *B*. Then, to guarantee isolation among players, FROG *(i)* executes each PEX into a different Docker container [3], and *(ii)* exploits a packet dispatching mechanism not based on zero-copy techniques, which would improve performance but that would not ensure the required level of isolation.

Obviously, a huge number of VNFs cannot come at the expense of performance; in fact, our prototype demonstrates that this platform can support thousands of users, each one running its own VNFs at reasonable speed, and that its overall performance, when FROG is installed on commodity hardware, is excellent (several Gbps on a

[2] In fact, numbers from ISPs indicate that in a deployment scenario such as a Broadband Remote Access Server (BRAS), the termination of 20K ADSL lines (and not users!) on the same network box is not uncommon.

single machine). The achievement of this goal required a careful implementation of the component that classifies packets and provides them to the proper VNFs (the **FROG vSwitch** in Figure 1.2), while the multi-server version further increases the performance in demanding environments.

This chapter is structured as follows. Section 1.2 analyses the related work, while Section 1.3 provides an overall view of the FROG platform, which is then detailed in Section 1.4. The implementation of the framework is instead described in Section 1.5. Section 1.6 compares FROG with the NFV model; experimental results are then shown in Section 1.7, while Section 1.8 finally concludes the chapter.

1.2 Related Work

Several works in literature address the possibility of consolidating many network functions on the same physical hardware. However, FROG defines a service model in which the end users connected to the node play a central role (Section 1.4); at the best of our knowledge, no other platform gives such an importance to the end users, although some of them (ClickOS [64] and NetVM [55]) do not prevent end users to deploy VNFs.

Particularly, **ClickOS** [64] bases on XEN [23] to create a multi-tenant software middlebox consisting of many ClickOS instances, i.e., virtual machines (VMs) with Click [71] running on top of a minimal operating system. Unlike ClickOS, FROG defines a service model in which the VNFs deployed by end users who are the source and the destination of network traffic are respectively the first and the last VNFs to operate on such a traffic; hence, FROG has been explicitly designed and optimized for this type of service. Moreover, results provided in [64] do not show how ClickOS scales with the number of VNFs, while the support of thousands of active players (and then VNFs) at the same time is one of the goals of the FROG platform.

Similar differences exist also between FROG and **NetVM** [55], a platform built on top of KVM and the DPDK [56] framework, designed to efficiently provide traffic to VNFs deployed as different VMs. Notably, NetVM guarantees traffic isolation among “untrusted” VMs, while packets are moved in a zero-copy fashion among “trusted” VMs.

CoMb [86] is a software middlebox that optimizes the resource usage of VNFs running on the same server. Particularly, a centralized controller selects the CoMb node in which deploy a VNF, using information such as the resources required by this VNF, the resources available on the nodes, and the traffic on which the VNF must operate. However, aspects such as the multi-tenancy, the traffic isolation among VNFs of different tenants, and the execution of thousands of VNFs on the same server are not mentioned in [86].

SHG [95] proposes an home gateway that can host both data plane functions (e.g., firewall) and cloud-based services (e.g., remote terminal, remote storage, etc.), but that does not consider the end user as players that can reprogram the box. The home gateway is also the target of [96], which proposes to partition the node in multiple slices, each one assigned to a different provider. Each slice is completely orthogonal to the others and has its own reserved resources (i.e., bandwidth, entries in the forwarding table, CPU) and control logic. However, in this case there is no provision to customize the packet processing.

Another work that we can cite is **Click** [71], which is one of the first proposals of a framework to customize the packet processing in the network. Compared to FROG, it presents several limitations. For instance, its processing path is rather static and cannot be changed at runtime; furthermore, it does not allows to dynamically load/unload network services and the multitenancy.

xOMB [22] is a software architecture for building programmable middleboxes; however, while FROG supports VNFs operating at any layer of the network stack, xOMB is particularly oriented to application-layer functions (e.g., HTTP load balancer). **APLOMB** [87] proposes to outsource enterprise middlebox functions to the cloud, which may be appropriate only in some cases as it could pose non trivial problems of traffic tromboning and latency.

ServerSwitch [61] is a programmable platform for datacenters that integrates an ASIC switching chip in a commodity server. It allows the implementation of new forwarding algorithms executed in the switching chip, which can be programmed by control plane applications running on the CPU. However, ServerSwitch seems to support only some particular VNFs (e.g., bridging). **PacketShader** [49], instead, is mainly a software router, although some other applications (e.g., IPSec gateway) are possible, but only one at a time. Notably, it exploits Graphical Processing Units (GPU) to accelerate some processing tasks.

As a final remark, it is worth pointing out that the work described in this chapter has been done in 2013, when the literature on NFV was still quite poor. More recent works on this research area are then discussed in Chapter 3, which describes a more recent architecture to deploy generic network services in the whole network of the telecom operator.

1.3 Overview

As shown in Figure 1.1, FROG operates in the context of a network edge node directly connected to the final users. Its main idea is to offer to a massive number of users the possibility to execute VNFs operating on (a portions of) the network traffic flowing through the edge device, according to a service model that in principle resembles to NFV. The FROG service model is based on two ingredients: *(i)* the capability to associate a portion of traffic, identified by a set of rules operating mostly on protocol field values, to a sort of network partition called *network tile*, and *(ii)* an execution environment that hosts the VNFs that have to process the traffic corresponding to a specific tile, called *Private EXecution environment* (PEX).

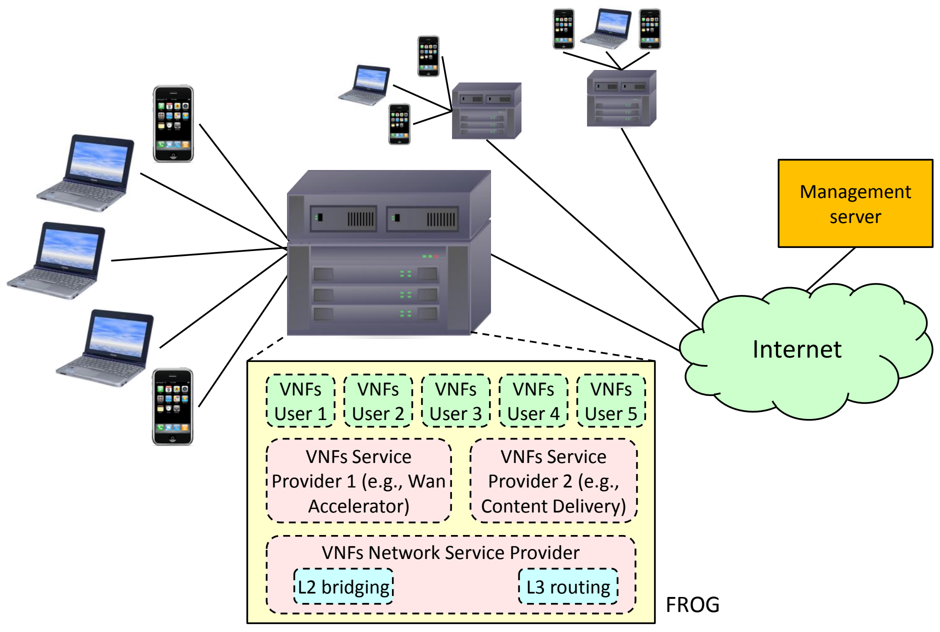

Management server
Internet
VNFs User 1
VNFs User 2
VNFs User 3
VNFs User 4
VNFs User 5
VNFs Service Provider 1 (e.g., Wan Accelerator)
VNFs Service Provider 2 (e.g., Content Delivery)
VNFs Network Service Provider
L2 bridging
L3 routing
FROG

Figure 1.1: The FROG operating context: a network edge node and its end users.

According to Figure 1.2, each PEX is associated with a specific player (i.e., an end user or other entities such as a content provider or a network operator), who has full control over the applications running in it, being able to choose which VNFs have to be executed and the order in which they are called. In addition, FROG defines a set of *permissions* granted to the VNFs running in each PEX, such as the possibility to modify and/or drop packets.

The capability to define tiles and permissions allows FROG to support multi-tenancy. Depending on the rules used to define network tiles, we may have either the possibility to partition the traffic into orthogonal or overlapped tiles. A packet that matches multiple tiles has to be delivered to several PEX, and hence we need to define a proper precedence level to determine in which order each PEX should receive the traffic.

Traffic coming from a new end user is redirected to a captive portal, which implements the user authentication. If the authentication is successful, a new PEX is created that receives all the packets belonging to the user's tile, defined by the traffic generated by/directed to the new end user's device. This PEX hosts a set of staked VNFs, installed by the user himself, which will be called sequentially on each user's packet. Finally, other PEX are created ahead of time by customers such as ISPs and content providers, and that can operate on traffic belonging to multiple end users.

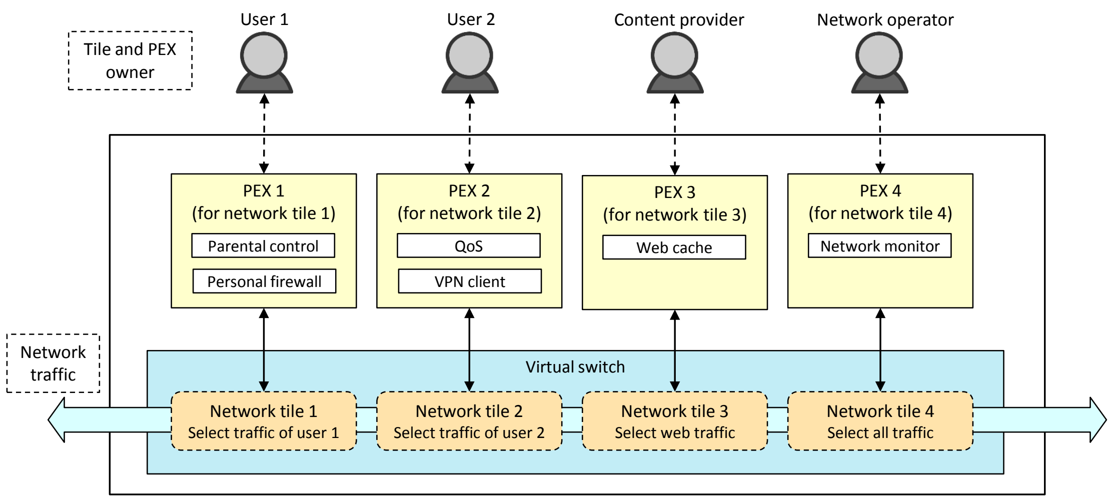

Tile and PEX owner
User 1
User 2
Content provider
Network operator
PEX 1 (for network tile 1)
Parental control
Personal firewall
PEX 2 (for network tile 2)
QoS
VPN client
PEX 3 (for network tile 3)
Web cache
PEX 4 (for network tile 4)
Network monitor
Network traffic
Virtual switch
Network tile 1 Select traffic of user 1
Network tile 2 Select traffic of user 2
Network tile 3 Select web traffic
Network tile 4 Select all traffic

Figure 1.2: High level view of a FROG node.

1.4 FROG design

This section presents the key aspects of FROG, which include the network tiles, the PEX and the permissions that can be associated with it, and the order in which a packet matching several tiles is provided to all the proper PEX. Moreover, it also presents the FROG data and control planes, and the management server.

1.4.1 Network tiles

Traffic entering in FROG must be classified to determine the tile(s) it belongs to, then packets need to be sent to the proper set of PEX for the processing. For the classification, FROG supports a set of rules operating mainly on protocol fields, and that are similar to the ones used in OpenFlow [68]. As tiles can be defined by multiple rules as shown in Figure 1.3, a packet belongs to a tile if it matches at least one of the rules defining the tile itself.

The capability to create partitions over the network and to allow different players to operate on their traffic does not represent a novelty [88]. However, those partitions were always oriented to guarantee network isolation, and then they do not overlap. As our objective is not to provide network isolation, FROG defines two types of partitions, or tiles in the FROG terminology, namely the *vertical* tiles and the *horizontal* tiles.

A **vertical tile** includes all the *unicast traffic* of a specific end user connected to FROG, and it is defined by two symmetrical rules expressed on the MAC address of the user terminal[3], namely $\{MAC_{user} \rightarrow *, * \rightarrow MAC_{user}\}$; these rules respectively represent *all the packets sent by that end user*, and *all the unicast traffic destined to the end user* himself. It is worth pointing out that, taking into account the direction of a packet (i.e., **from** the end user, or **towards** the end user), vertical tiles are not overlapping, since a (unicast) packet will match at most one vertical tile in each direction (the tile of the sender and the tile of the receiver)[4]. The *Default vertical tile* is a particular tile that is in charge of all the unicast traffic coming from an access port (i.e., from a port connected to the end users) and that does not match any other vertical tile. In practice, it handles the traffic belonging to a new end user connected to FROG, who still does not have his vertical tile configured (and his PEX running).

Unlike vertical tiles, **horizontal tiles** are defined with an arbitrary set of rules, not necessarily symmetric with respect to the network traffic, operating on any supported field. In this case we may have network tiles with overlapping rules, which means that a packet can belong to several horizontal tiles at the same time.

Figure 1.3 shows a possible configuration of FROG with rules defining both vertical and horizontal tiles. As evident, a PEX under the control of an end user is allowed to operate only on the traffic belonging to that user, i.e., on the packets to/from the user himself. Vice versa, horizontal tiles are appropriate to implement traditional network middlebox functions working on traffic aggregates (e.g., a web cache); hence, they appear more appropriate for hosting VNFs of players such as service providers or network operators.

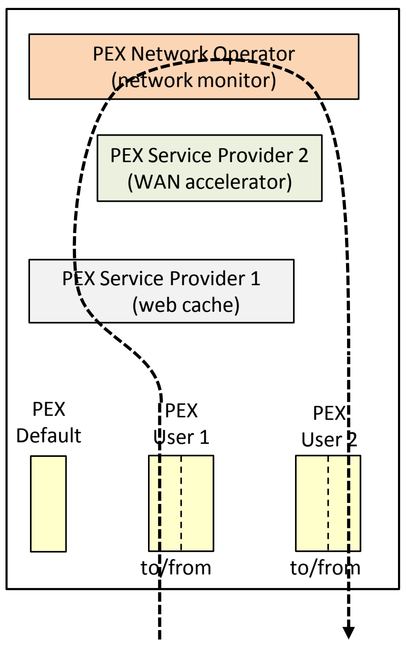
1.4.2 Service order

A packet that enters in FROG through an access port is immediately checked against the rules defining the vertical tiles, in order to identify the tile associated with the end user who is sending the traffic. Given the structure of the rules defining vertical tiles, at most one of them is matched, and the packet is then provided to the PEX associated with that tile, so that it can be processed by the VNFs running in that PEX. A packet sent by an unauthenticated user matches the Default vertical tile, and hence it is processed in the *Default vertical PEX*. This way, the user can be authenticated, and the vertical tile identifying his traffic, as well as the PEX running his VNFs, are created.

After that the packet has been processed in the PEX associated with the sender user, we move to horizontal tiles. The matching against horizontal tiles is more

[3]Note that this parameter may be different in other implementations (e.g., ATM VPI/VCI), provided that it allows to select univocally the traffic of a single user.

[4]Packets coming from (directed to) the Internet match *a single* vertical tile, since their source (destination) is not an end user connected to this FROG node.



Vertical tiles

Network tuples	PEX	Permissions

`MAC1 → *`
`* → MAC1`
	`User 1`	`read/ write`

`MAC2 → *`
`* → MAC2`
	`User 2`	`read/ write`

`* - {MAC1, MAC2} → *`
`* → * - {MAC1, MAC2}`
	`Default`	`read/ write`

Horizontal tiles

Network tuples	Priority	PEX	Permissions

`* → tcp port 80`
`tcp port 80 → *`
	`40`	`SP1`	`read`

`* → net 8.8.8.0/24`
`net 8.8.8.0/24 → *`
	`30`	`SP2`	`read/write`

`* → *`	`20`	`NET OP`	`read/write/ forward`

Figure 1.3: Possible journey of a packet.

complicated because their rules can overlap, hence a packet can match multiple horizontal tiles. For this reason, they are associated with a *priority*, whose value is unique within FROG. Hence, the packet will be checked against the horizontal tile with the highest priority and delivered to the corresponding PEX in case of positive matching, then the same operation is repeated with the tile with the second highest priority, and so on.

Finally, before exiting, a unicast packet is checked again against the rules defining the vertical tiles, and then delivered to the PEX running the VNFs deployed by the destination end user.

Given the above rules for the service order, a PEX associated with a vertical tile looks like a portion of the network stack of the user terminal that is moved in the network. In fact, it is the first PEX that receives the traffic sent by the device as soon as it enters in FROG, as well as it is the last PEX that processes the packets before they leave FROG towards the user terminal. Hence, for example, a personal firewall running in the user terminal is functionally equivalent to the same application running in the PEX associated with that user[5].

[5]For the sake of precision, some differences still exist. The most important refers to the visibility over encrypted traffic, which can be obtained more easily by an application running in the user terminal. This problem has not been addressed in work presented in this chapter, and it is left for future work.

In the example shown in Figure 1.3, a packet enters in FROG and is first delivered to the vertical tile that matches its source MAC address, then it hits a variable number of horizontal tiles traversed in an order defined by their priority, and finally it concludes its journey by hitting the vertical tile that matches its destination MAC address. Note that this processing path is valid independently from the VNFs running in each PEX.

The fact that VNFs can (almost) arbitrarily change the packet content may lead to a situation in which a packet that, in the next processing step, should have been delivered to tile T_1, is modified so that the next matching tile becomes T_2. This may create a cyclic workflow, in which PEX A modifies the packet so that it matches the tile of PEX B, which modifies the packet to match the tile of PEX A, endlessly. To avoid this problem, a packet exiting from the vertical tile of the sender user will be matched against horizontal tiles, no matter which modifications have been done to the packet content. Similarly, if a packet matched an horizontal tile with priority Φ, it will be checked against all the tiles having priority $< \Phi$. Finally, after that a packet has been processed in the PEX of the destination end user, it is sent out of FROG. This way, the PEX calling order is always a direct acyclic graph, hence guaranteeing that a packet cannot enter into a processing loop involving different PEX.

Broadcast and multicast traffic

A broadcast/multicast packet entering in FROG is first delivered to the vertical PEX of the end user who is sending that packet; then, it is checked against horizontal tiles in an order defined by their priority, as happens in case of unicast traffic. Finally, when the packet has to be delivered to the vertical PEX associated with the end user who is the destination of the packet itself, the time has finally come to duplicate this traffic.

In principle, FROG should duplicate the packet as many times as the number of access ports on which the packet has to be sent, and then each copy should be delivered to the vertical PEX of the end user who is connected to that port. Finally, if not dropped by any VNF, the packet is delivered to the user terminal. This is the case of `User4` in Figure 1.4, where VNFs running in User4's PEX receive the packet and (if permissions allow) can even drop it; in this case, the above mentioned packet will never reach `User4`.

Vice versa, if many user devices are connected through the same access port (such as `User1` and `User2` that share the same WiFi interface in Figure 1.4), the behavior is different. In this case, the packet is further duplicated $N+1$ times, where N is the number of terminals sharing that interface. A first copy of the packet is delivered directly to the interface, hence reaching immediately the user devices. The other N copies will be sent, marked with a special flag, to the corresponding PEX of those end users, giving to the VNFs the possibility to process the packet (e.g., a network monitor can update its internal counters). However, this packet will be dropped by FROG as soon as it exits from the PEX, hence losing any modification that may have been occurred to the packet. The rational for this behavior is to offer to the VNFs running in the PEX the possibility to receive exactly the same traffic as they would be executed in the user terminal; however, in case of shared interfaces we cannot give them the possibility to modify the packet, as a single copy of each packet must be sent across the shared port and we are unable to handle possible conflicts when the different PEX touch the packet (modify/drop) in an incoherent way.

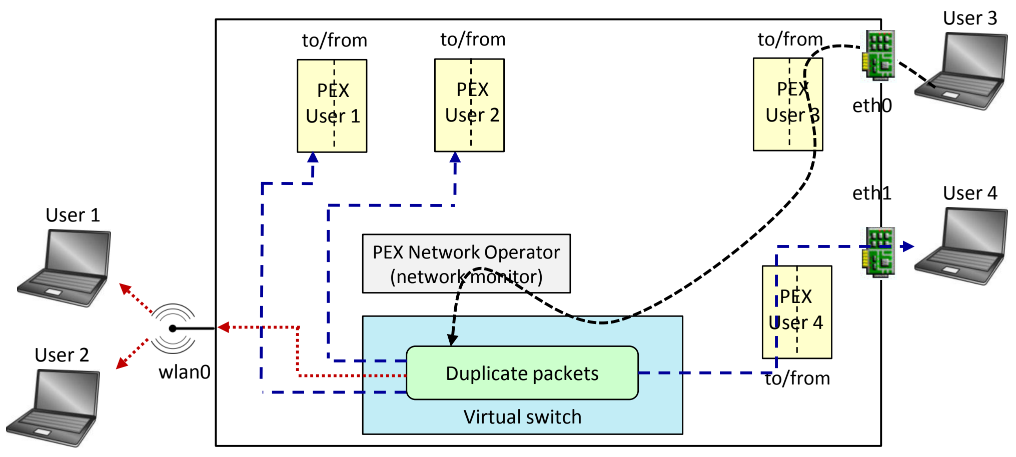


Figure 1.4: Possible journey of a broadcast/multicast packet.

1.4.3 Private EXecution environment

As previously mentioned, the PEX is the execution environment running all the VNFs installed by a single player. In particular, it receives (all) the traffic matching the network tile to which it belongs to, and in turns it provides these packets to the VNFs it is running. A packet entering in a PEX will traverse all the applications executed in that PEX in an order decided by the PEX owner and that, in the current prototype, has to be strictly sequential. In case a VNF creates a new packet, this will traverse all the applications that follow, while the ones that appear earlier in the sequence will have no visibility on that packet.

Some network privileges can be associated with the PEX. For instance, end users are usually enabled to do whatever they want on their traffic, including generating and/or modifying packets traversing their PEX. A PEX of an entity in charge of network monitoring may have instead a “read mode” privilege. Further, other PEX could have also access to network parameters, and influence the forwarding process of the FROG node, such as determining the output interface of a given packet. Then the PEX is also in charge of enforcing these permissions, and then of preventing that VNFs perform illegal operations.

The possibility granted to VNFs running in a PEX with the *packet modification* permission, to potentially change arbitrarily the packet content, including the possibility to create new packets, may lead to packets that do no longer belong to the tile associated with the current PEX, which we feel may not be acceptable. For this reason, the PEX allows modifications to the packet content as long as the modified/new packet *still belongs to the tile associated with that PEX*. For instance, a packet that is under processing in the `SP1` PEX in Figure 1.3 cannot be modified to become `tcp port 1000` → `tcp port 8080`, as in this case it would no longer belong to that tile. However, the enforcement of these constraints introduces some additional overhead in the FROG processing, as packets exiting from a PEX are checked for tile conformance and, if the control fails, a shadow copy is sent to the next processing component instead of the original packet[6].

As a final remark, memory spaces among different PEX are disjoint, so that VNFs installed by a user cannot intercept/corrupt the traffic belonging to another PEX. Instead, this property is not guaranteed to the VNFs running in the same PEX, which share the same address space for performance reasons. Therefore we can expect that a misbehaving VNF could affect the execution of the other applications running in the same PEX, although this may be considered reasonable since they all belong to the same player.

1.4.4 Data plane, control plane and management server

As depicted in Figure 1.5, FROG includes both a data plane and a control plane portion.

The data plane consists of the PEX and the VNFs deployed in these PEX, which operate on a portion of the traffic flowing through the node, i.e., on a network tile.

The control plane portion, instead, is connected to the rest of FROG through a virtual port, named `tap0` in the picture. The `tap0` interface is visible from the TCP/IP stack of the operating system (while all the other ports of FROG are hidden), hence all the traditional TCP/IP applications can be executed on that interface (e.g., the captive portal to authenticate new users, a DHCP server, etc.).

Finally, the entire set of FROG nodes is coordinated by an external management server, as shown in Figure 1.1. It contains the user database, the permissions, the list of VNFs, and more. Furthermore, it stores the VNFs associated with each user, which are in fact copied from this server to the proper FROG node each time a new

[6]In fact, this algorithm has been optimized and a complete shadow copy of the packet is created only when PEX has read-only privileges over the packet content. In case of read-write privileges, only the fields that concur to determine the network tile are copied.

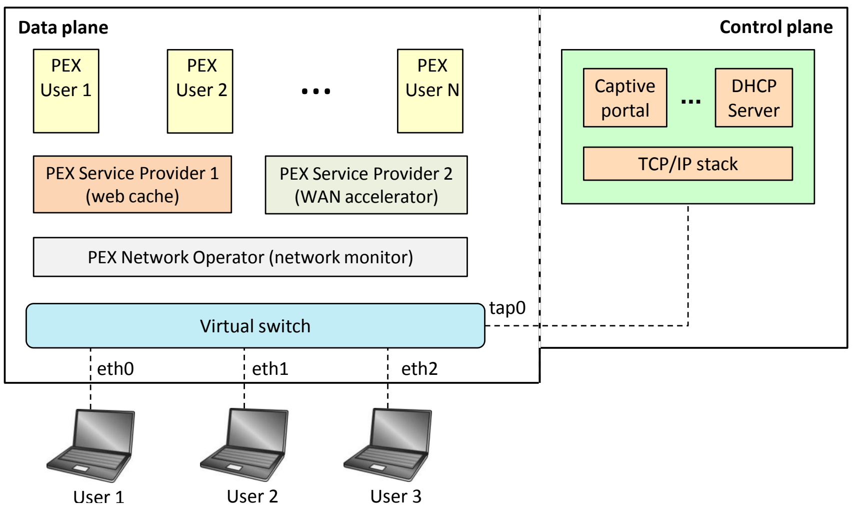


Figure 1.5: Data and control plane.

PEX has to be activated, e.g., each time an end user logs in.

1.5 Implementation

This section presents the FROG software architecture that includes, in addition to the several PEX associated with the users, the FROG virtual switch (which is the component in charge of exchanging packets with the network and implementing the network tiles) and the exchange buffers. The entire architecture, which has been designed to efficiently scale with the number of VNFs, is depicted in Figure 1.6.

1.5.1 Virtual switch

The **FROG virtual switch (fvSwitch)** is the component in charge of implementing the network tiles, then of providing the packets to the proper PEX according to the service order defined in Section 1.4.2. Hence, this module cyclically repeats the following main operations: *(i)* read packets from the physical network ports, classify the traffic (according to the network tiles) and deliver it to the PEX associated with the first matched tile; *(ii)* read packets from each running PEX, classify and forward them to the proper next PEX, or send the packets on the network through the proper physical network port(s).

The fvSwitch relies extensively on batch processing; in fact, phase (*i*) is repeated multiple times by reading several packets from each port before moving to phase (*ii*), where several packets per PEX are processed in a row before going to the next PEX. This allows the fvSwitch to execute code that has an high degree of locality and to concentrate memory accesses to nearby locations (e.g., reading several packets in a

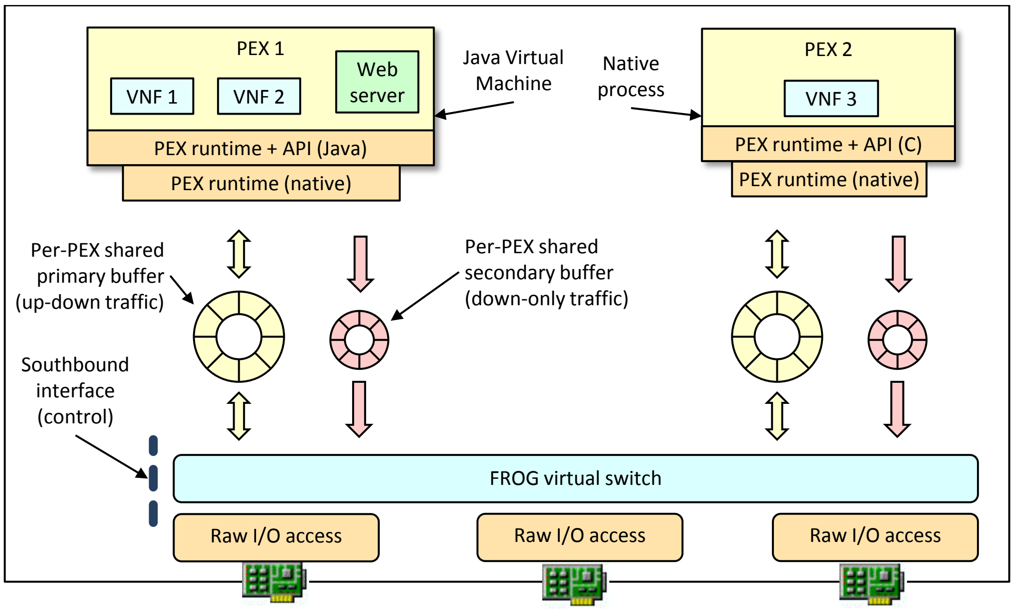


Figure 1.6: FROG software architecture.

row), which have an important impact on the performance of the system because of the capability to exploit cache (code and data) locality.

Since the fvSwitch is supposed to be traversed by a huge amount of traffic (all the packets flowing through the node, each one multiple times), this module operates in polling mode. In fact, if interrupts are used to notify the packets arrival, the throughput can drop dramatically with high packet rates [70]. Moreover, to avoid expensive context switches [60] and limit the L1/L2 cache pollution, a CPU core is statically allocated to the fvSwitch. Note that this module has been designed to use a *single* core, as we would like to allocate all the others to the PEX.

The fvSwitch has raw access to network ports through accelerated NIC drivers, which enable "direct" I/O with the hardware without involving the operating system. Currently we support DNA [44], although a fallback mode exploiting `libpcap` has been implemented in order to allow FROG to operate on NICs that do not support accelerated drivers.

Finally, the fvSwitch is also responsible to duplicate broadcast and multicast packets according to the rules described in Section 1.4.2.

Why yet another virtual switch

In principle, existing virtual switches such as Open vSwitch (OvS) [76], whose forwarding table can be configured by means of SDN protocols [80] (e.g., Openflow), can be used in FROG to move packets among PEX and the physical network interfaces. However, we decided to implement the fvSwitch from scratch, for a number of reasons.

First, as described in Section 1.4.2, in FROG we have a well defined service model (the packet is first matched against rules defining vertical tiles, then it is sent to the PEX associated with the matched horizontal tiles, and finally it is again matched against rules defining the vertical tiles), then the fvSwitch forwarding table and matching logic have been designed and optimized to implement such a service model. Second, as detailed in the next section, the fvSwitch exchanges packets with a PEX using a couple of memory buffers shared with the PEX itself, and optimized for the NFV case in which almost all the packets provided by the virtual switch to a VNF will eventually come back to the vSwitch itself.

OvS, as any other Openflow-based virtual switch, is instead designed to be generic, and then it does not do any assumption on the order in which rules have to be matched, as well as it is not optimized for the case in which packets sent through a port (e.g., toward a VNF) will likely come back through the same port.

1.5.2 Packet buffers

While designing the internal packet exchange mechanisms between the fvSwitch and the PEX, we had to consider two opposite requirements: *performance*, which suggests to use a single buffer shared among all the components in order to exploit a zero-copy algorithm, and *isolation*, which requires each PEX to have its own dedicated buffer in order to guarantee that a VNF can only have access to packets that belong to its tile. The resulting mechanism is a mixture of those requirements: each PEX has its own buffer shared with the fvSwitch, hence a packet traversing N PEX has to be copied by the fvSwitch N times. However, each packet is copied only once in each buffer, for both the up and down trips. In fact, even if this communication channel can be modeled with two FIFO queues, one bringing packets from the fvSwitch to the PEX and the other for the opposite direction, we exploit an algorithm (extensively detailed in Chapter 5) that moves a packet from the fvSwitch to the PEX, and then back to the fvSwitch without any copy of the packet within the PEX itself.

As shown in Figure 1.6, each PEX shares two circular buffers with the fvSwitch: the *primary buffer* is used by the fvSwitch to send packets to the PEX and to receive them back once their processing has been completed, while the *secondary buffer* is used only for the traffic that is generated by VNFs, as the primary buffer does not accept insertions of new packets from the PEX.

The algorithm that manages the primary buffer, shown in Figure 1.7, uses four indexes, which are respectively a pointer to the last packet written by the fvSwitch in the buffer (`fvSwitch.write()`), a pointer to the last data that has actually been read (and processed) by the PEX (`PEX.read()`), the oldest packet that has been processed by all the VNFs in the PEX (`PEX.write()`) and hence is ready to be

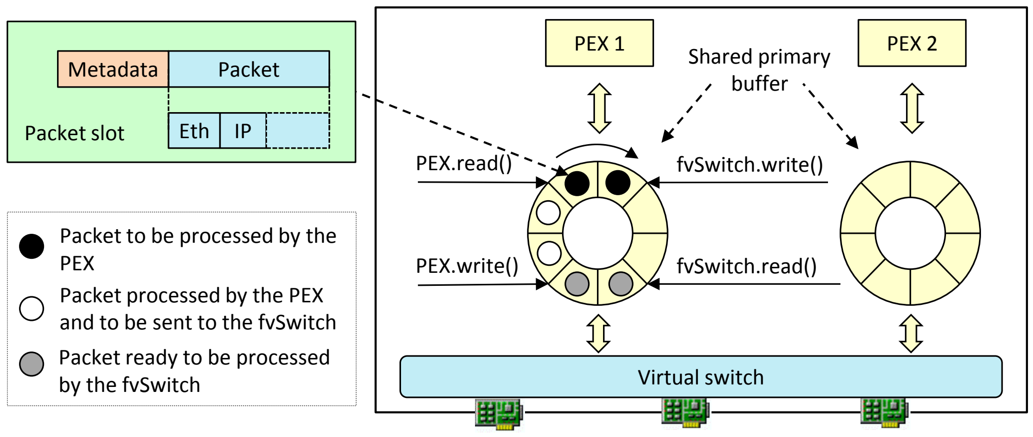

Metadata
Packet
Packet slot
Eth
IP
● Packet to be processed by the PEX
○ Packet processed by the PEX and to be sent to the fvSwitch
● Packet ready to be processed by the fvSwitch
PEX 1
Shared primary buffer
PEX 2
PEX.read()
fvSwitch.write()
PEX.write()
fvSwitch.read()
Virtual switch

Figure 1.7: Packet exchange between the fvSwitch and PEX.

delivered to the next PEX in the chain, and finally the next packet that will be drained by the fvSwitch (`fvSwitch.read()`) and sent to the next PEX. Note that this buffer is lock-free, hence very efficient; moreover, in the implementation we took care not to access too often to the shared indexes (we use shadow copies instead) in order to minimize the cache synchronization cost among processes running in different CPU cores.

In addition, the primary buffer is operated through a batching mechanism; in fact, the fvSwitch writes several packets before signaling the PEX over a shared semaphore, waking it up[7]. This allows the PEX to be scheduled (hence starting its processing) only when a reasonable amount of packets is present in the buffer, hence limiting the number of context switches in the system and exploiting at best memory locality as in the fvSwitch. Obviously, a timeout has been implemented as well in order to avoid packets starving in the buffer in case the PEX receives limited amount of traffic over time. The PEX will suspend itself only when no packets waiting to be processed are present in the buffer. To facilitate batch processing in the fvSwitch, the `PEX.write()` pointer is updated only when the amount of packets processed in the PEX exceeds a threshold, or when there are no more packets to be processed.

The primary buffer is well suited for packets that enter in the PEX, traverse all the local VNFs, and return back to the fvSwitch. However, it may happen that: *(i)* a packet is dropped by a VNF, and then it cannot continue its journey; *(ii)* a VNF modifies a packet so that it exceeds the MTU, hence requiring to be split in multiple fragments; *(iii)* a VNF generates new packets.

The first point is addressed by setting a special flag in the packet metadata (Section 1.5.4) that informs the fvSwitch to drop that packet as soon as it is received

[7]The fvSwitch is aware of the status of the PEX, i.e., running or waiting for packets, thanks to a `status` variable shared with the PEX itself.

back in the buffer. Instead, points *(ii)* and *(iii)* require the *secondary buffer*, which is used by the PEX to send its own generated packets to the fvSwitch. This buffer is a traditional (circular) FIFO queue: a special flag in the packet metadata of the primary buffer informs the fvSwitch that, after that packet, the following N packets have to be read from the secondary buffer, before returning to drain the traffic from the primary buffer.

Currently, both buffers have slots with a fixed length, whose size is equal to the maximum packet size of the network[8]. Moreover, they are allocated in memory using huge pages, in order to reduce the pressure over the Translation Lookaside Buffer (TLB).

1.5.3 Private execution environment

Each PEX is implemented as a different *process* running all the VNFs belonging to a specific player. In particular, we defined (and implemented) two flavors of PEX: one privileges the features offered to the users, while the other is more oriented to the achievement of high performance.

The former type of PEX is a Java Virtual Machine (JVM) running Java VNFs and that, as shown in Figure 1.6, is enriched with different components. First, a native library written in C and based on the Java Native Interface (JNI) takes care of the interface with the rest of the system (e.g., accessing to the shared buffers). Second, a set of Java classes that handle the communication among the VNFs deployed in the PEX, implement the dynamic loading/unloading of the VNFs[9], and export a rich set of API to VNFs developers (described in Chapter 2). Third, a web server that is used for management (e.g., to handle the command to load/unload applications) and exported to VNFs for their own purposes (e.g., configuration, visualization of internal data, etc.).

The second type of PEX is instead a pure native process that supports VNFs written in C/C++, which have to be linked with another native library, written in C, that implements the interface with the rest of the system. The number of features are reduced in this case (no dynamic load/unload of VNFs, simpler API exposed to VNFs developers, no web server) but it guarantees higher performance with reduced requirements in terms of both CPU and memory. In fact, the JNI layer used in the Java PEX is known to be rather inefficient, hence increasing considerably the processing cost per packet; furthemore, the memory footprint of the JVM with the required pieces (JNI, web server, etc.) reaches about 18MB, which may represent a

[8]In this respect we disabled the TCP Large Receive Offload on the NIC, as this function merges multiple TCP segments creating packets up to several tens of kilobytes.

[9]In a first phase we used OSGi for this purpose, but we found its memory requirements incompatible with our targets.

limitation when a massive number of PEX have to be executed concurrently on the same machine.

As mentioned in Section 1.5.2, PEX operate according to a blocking I/O model, hence freeing CPU resources when no packets have to be processed. This allows to execute a number of PEX that is far beyond the number of CPU cores available on the machine, as we expect that, in average, each PEX stays idle most of the time (it just processes the traffic of a single player). In fact, the implemented techniques for efficiently exploiting CPU cycles (memory locality, a few context switches) allow to potentially execute thousands of PEX on a single physical server.

To prevent that VNFs consume too many resources (and hence to increase the number of users supported), a PEX is executed on a single CPU core; however, this may limit the throughput of the system in case few of them are installed, as we are unable to exploit all the available cores. An extension that allows a PEX to exploit multiple cores is left to our future work.

Finally, it is worth nothing that the permissions, which are one of the peculiarities of FROG , introduce a noticeable overhead in the PEX processing. In fact, they must be enforced in each PEX in order to guarantee that VNFs do not perform forbidden operations, such as the shadow copies mentioned in Section 1.4.1. Although we took care of implementing the permission checking very efficiently, we found that this part of the code could slow down the system throughput up to 10%.

Resource isolation

As the FROG owner does not have any control on the VNFs installed in the PEX, the framework should limit the effects of malicious applications. Unfortunately, the solution to run each PEX into a different virtual machine is not compatible with our scenario in which a huge number of end users are connected to the same FROG, due to the huge amount of resources required to execute each virtual machine.

Then, in order to guarantee that a VNF cannot access to resources belonging to other PEX/users, as well as to guarantee that it does not consume all the hardware resources available in the machine where FROG is running (i.e., RAM and CPU), each PEX is executed in a different Docker container [3]. Containers are in fact a lightweight virtualization mechanism that, unlike virtual machines, does not run a complete operating system; particularly, all the containers share the same kernel. Docker containers limit the resources visible by a user space process through the `cgroups` [4] feature of the Linux kernel, while isolation is provided through the Linux `namespaces` [10], which give to the process running in the container a limited view of the process tree, networking, file system, and more.

As described in Section 1.5.2, the fvSwitch and each PEX share two buffers and a semaphore. Unfortunately, Docker by default uses the Inter Process Communication (IPC) namespace, forbidding shared memory and semaphores between a process running in the container and the processes executed outside such a container (the fvSwitch in our case). Then, to bypass these limitations, we had to slightly modify the Docker behavior, as well as to change the implementation of the `sem_open` system call.

To conclude, traffic isolation is guaranteed with the creation of distinct exchange buffers per each PEX, which makes impossible, to VNFs running in a PEX, to access the buffers (and hence packets) of another PEX.

1.5.4 Packet metadata

As shown in Figure 1.7, each packet is associated with some metadata during its journey within a FROG node, which are used by the fvSwitch to dispatch packets among the PEX, and that can be exploited by the VNFs. In particular, metadata can either be read or written by VNFs, depending on their permissions.

Among the information kept in the metadata we can cite a shadow copy of the fields used to identify the network tiles, some flags (e.g., the one used to inform the fvSwitch to drop a packet), the input/output ports. Particularly, the shadow copy of the fields used to identify network tiles cannot be modified by VNFs, and it is used by the PEX to restore the original value in the packet, in case the VNF modified such a packet so that it no longer belongs to the current slice (Section 1.4.3).

1.5.5 Distributed architecture

Given the huge number of users (some thousands, as presented in Section 1.1), hence PEX, that we may expect to be handled by FROG, the framework has been engineered to support a distributed architecture.

The idea is to distribute the PEX across multiple servers and to transform the fvSwitch into a distributed module, although, from an external view, the system still appears as an unique network device, as shown in Figure 1.8(b) (for instance, VNFs do not notice any difference with respect to the case in which a FROG node is actually a single server). In particular, the system is made up of a cluster of servers connected through an high speed network (currently a 10Gbps switched network) with no external access, namely the *FROG crossbar*. Each physical port of the server can then be marked either as *external* or *crossbar*. The former is a port visible from the outside world, and can be either *access* (i.e., connected to the end users) or *core* (i.e., connected to the Internet); the latter is instead a port used to connect the server with the rest of the cluster.

Each PEX has a number that identifies univocally the server where it is executed, hence each fvSwitch knows exactly the server responsible for the next processing step. To facilitate the processing in the next server traversed by a packet, the identifier of the next PEX is added to the metadata; then, both packet and metadata

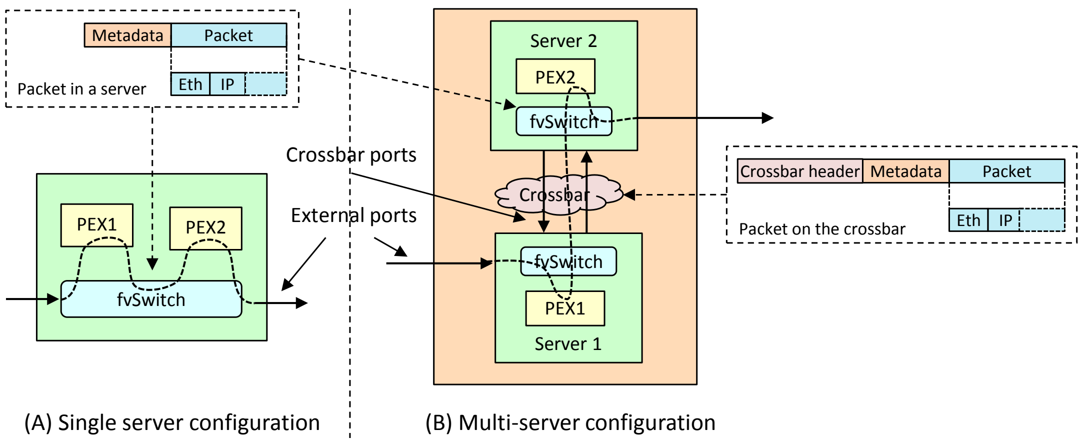


Figure 1.8: Single vs. multi-server architecture.

are encapsulated into an additional Ethernet header[10] and sent on the crossbar, which will deliver the packet to the server where the target PEX is running. A packet that is received through a crossbar port is not classified again by the fvSwitch, which simply copies it in the buffer associated with the target PEX, as written in the metadata.

Furthermore, the output port selected for a given packet could be physically installed in another server. In this case the packet (enriched with the proper metadata and encapsulated in the crossbar header) is sent to the target server, which will send it on the network through the correct output port.

1.6 FROG and the NFV model

Table 1.1: NFV vs FROG

	NFV	FROG
Execution model	One VNF per VM, multiple VMs per player	Multiple VNF per PEX, one PEX per player
Virtualization environment	Full fledged VM	PEX executed inside Docker container
Network interfaces towards VNFs	Virtualized or paravirtualized network interface cards	Designed and optimized for the NFV environment
Service order	No assumptions	Src end user → Generic players → Dst end user

Although FROG looks like a possible instantiation of the NFV paradigm [43], there are some important differences between the two proposals, which are summarized in Table 1.1 and discussed in the remainder of this section.

First, NFV defines a different virtualization environment (i.e., virtual machine) per VNF, while FROG runs all the VNFs belonging to the same player into a single

[10]The MTU on the crossbar interfaces has been configured appropriately to exceed the traditional MTU of the Ethernet network.

execution environment called PEX. This choice originates from performance reasons and it is intended to reduce the pressure that the high number of expected PEX (and hence players) poses to our system, which could increase even further in case each VNF would require its own PEX. For instance, while the zero-copy mechanism cannot be implemented to move packets between PEX, in order to guarantee traffic isolation among players, it is considered reasonable within the same PEX, which executes VNFs that belong to the same players.

Second, as discussed in Section 1.5.3, in order to support thousands of (active) players at the same time, a PEX is not a full fledged virtual machine (as indicated by NFV), whose requirements in terms of memory and CPU would be too onerous. It is instead a process executed inside a Docker container, which is a lightweight form of virtualization that exploits some features of the Linux kernel to provide resources isolation and limitation.

Another important difference is the nature of the interface between the PEX and the underlying virtual switch. In fact, PEX features a dedicated communication primitive toward the fvSwitch that defines a *single* (and very optimized) communication channel to send/receive all the traffic, while VNFs executed in virtual machines access to network packets through virtualized or paravirtualized network interfaces cards.

Finally, NFV does not make any assumption on which is the next VNF that has to process a packet. Instead FROG has a well defined service order, and then the fvSwitch is optimized for this specific case; as shown in Section 1.7, this results in reasonable performance of FROG even with a huge number of concurrent VNFs.

1.7 Experimental results

A prototype of FROG was installed on a workstation with 32 GiB of memory, CPU Intel i7-3770 @ 3.40 GHz (four cores plus hyperthreading), Ubuntu 12.10, kernel 3.5.0-17-generic (64 bits), which was equipped with a Silicom dual port 10Gbps Ethernet NIC, based on the the Intel X540 controller and managed by the DNA driver. We also implemented a distributed version of FROG, made up of two of the above machines. In all tests, an entire core was dedicated to the fvSwitch, while PEX have been distributed among the remaining cores. Each test lasted 100 seconds and was repeated 10 times, then results were averaged.

Graphs representing the maximum throughput are provided with a bars view that reports the throughput in millions of packets per second, and a points-based representation that reports the throughput in Gigabit per second. Instead, latency measurements are based on the `gettimeofday` Unix system call and include only the time spent by packets in the FROG node, without the time needed to send/receive data on the network.

1.7.1 Single server

In order to provide a concrete demonstration of the scalability of our framework , this section evaluates the maximum throughput that can be obtained with a single FROG node executing a growing number of PEX, and the latency introduced by the node itself in the same conditions.

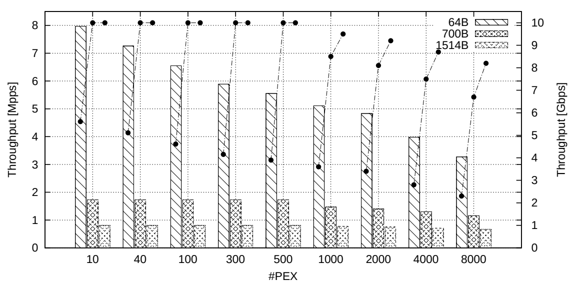
During the tests, FROG has one physical port connected to a traffic generator and another one to a traffic receiver, handling unidirectional traffic such as in Figure 1.8(a). Each packet traverses two PEX, each one running a simple VNF that calculates a signature across the first 64B of the packets[11]. We repeated the test with both our PEX implementations in order to compare the performance of the Java PEX with that of the native PEX. Network tiles are defined by MAC addresses. The traffic generator sends packets in a way so that the first one belongs to the tiles associated with PEX 1 and PEX N, the second packet hits PEX 2 and PEX (N-1), and so on. This pattern stresses the fvSwitch that never receives from the network two consecutive packets to be delivered to the same PEX, with a dramatic impact on memory access patterns. Finally, in each test condition, we selected the minimum size of the shared buffers that allowed the system to work without losses in the communications between the fvSwitch and PEX.

As expected (Figre 1.9), the throughput decreases when increasing the number of PEX in the system. This is mainly due to the loosely localized memory access patterns, as the fvSwitch has to handle packets located in different exchange buffers (hence poor cache locality), which causes also an increase of the CPU TLB misses. However, FROG reaches an impressive result of 6.8Gbps with 700 bytes packets when 8000 PEX are executed. Furthermore, the available bandwidth (i.e., 10Gbps) is saturated in many cases with packets of 700 and 1514 bytes. By comparing Figure 1.9(a) with Figure 1.9(b), it is evident that the native PEX is more efficient than the Java PEX; however, in our opinion the latter performed rather well and its throughput never fell 20% below its competitor.

Figure 1.10 plots the latency introduced by FROG with a growing number of native PEX. As evident, it tends to increase considerably with the number of PEX, reaching an average value of 59.22ms in case of 8000 PEX. The latency introduced by the Java PEX (not reported) is higher in each test case, reaching a worsening of 13% in average in case of 1000 PEX.

While the difference between native and Java PEX in terms of performance and latency is limited, the main difference consists in the maximum number of PEX instances concurrently running in our system. In fact, each native PEX uses approximately 1.5MB of memory (the memory pages of the executable are shared

[11]This workload is rather realistic, as it emulates the fact that most network applications operate only on the first few bytes (i.e., the headers) of the packet.


(a) Native PEX.

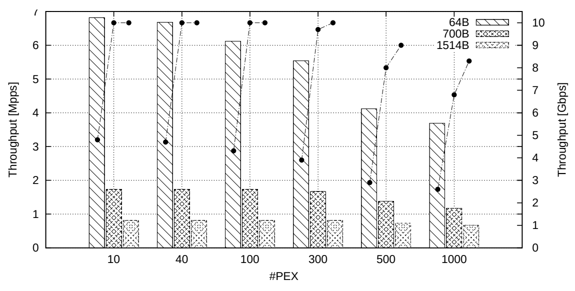

(b) Java PEX.

Figure 1.9: Throughput with a growing number of PEX.

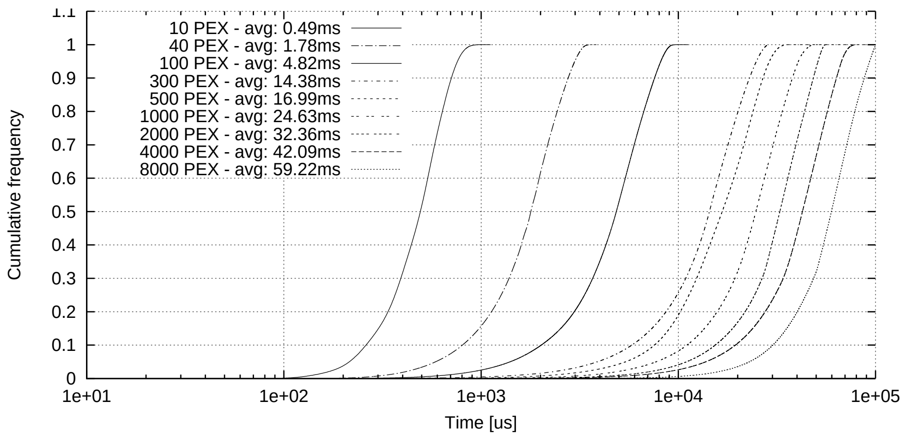


Figure 1.10: Latency introduced by the node with a growing number of native PEX.

among the many instances of the program), while the Java version requires about 18MB, plus the memory allocated for the shared buffers, with little possibilities to share the memory pages of the executable. This means that, while we were able to squeeze 8000 native PEX in our machine, we were forced to stop at 1000 Java PEX.

1.7.2 Multi server

This test measures the throughput reachable with FROG in a dual server configuration, as shown in Figure 1.8(b). During the test, half of the packets enters from `server1` and leaves from `server2`, while the others traverse FROG in the opposite direction. Also in this test, two consecutive packets coming from the network never belong to the same tile; moreover, each packet is first processed by a PEX running on the first server, and then by a PEX executed on the second server, so that it always crosses the crossbar once. Each server runs 4000 native PEX and results are depicted in Figure 1.12.

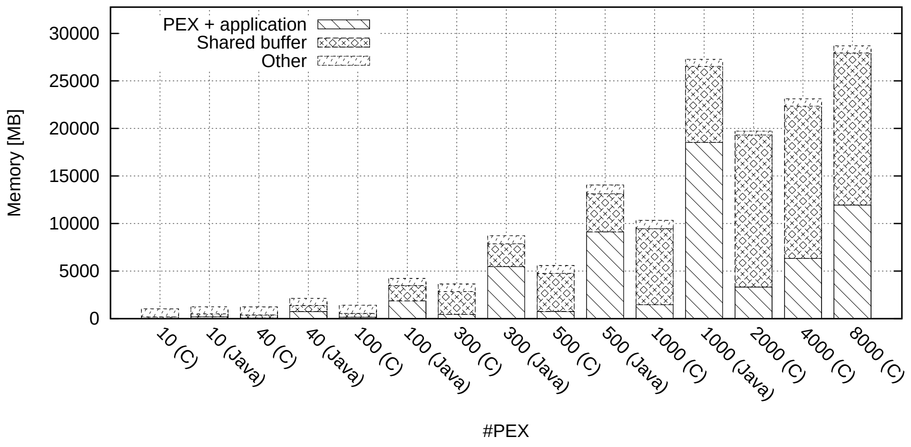


Figure 1.11: PEX memory consumption.

The graph shows that the throughput (with 64B packets) increases from 4Mpps in case of 4000 PEX on a single server, to 5Mpps with twice the number (i.e., 8000) of PEX partitioned between two servers. This means that the *average* throughput per server is even higher in case of the multi-server setup because of the more efficient memory access patterns, which looks promising for our future work focusing on extending the system to support even more users.

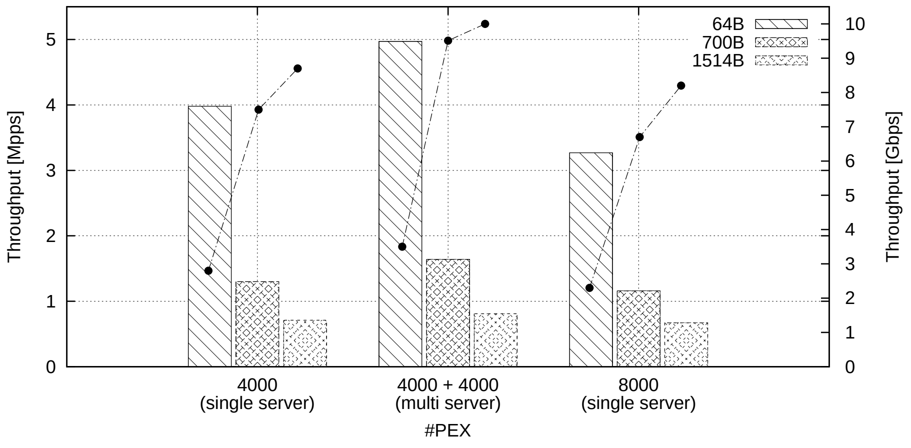


Figure 1.12: Single vs. multi server.

1.7.3 The DPI VNF

To validate FROG in critical conditions, we wrote a DPI on the top of the native PEX API, which exploits the `PCRE` library to classify the four protocols listed Table 1.2. Then, we measured the throughput with a growing number of PEX, each one running an instance of the DPI. Network tiles are again defined by MAC addresses, and the packet generator sends traffic so that two consecutive packets belong to different tiles.

Table 1.2: Protocols and corresponding regex.

Protocol	**Regular expression**
HTTP	`http/(0\.9|1\.0|1\.1) [1-5][0-9][0-9]|(connect|post|get|head|propfind|mkcol|delete|put|copy|move|lock|unlock)[\x09-\x0d -~]* http/[01]\.[019]`
FTP	`^220.*ftp |^220.*(\x0d\x0a)$`
TELNET	`^\xff[\xfb-\xfe].\xff[\xfb-\xfe].\xff[\xfb-\xfe]`
SSH	`^ssh-[12]\.[0-9]`

The throughput measured with `HTTP GET` packets of 480B ranges between the 0.44Gbps in case of 10 PEX, and the 0.38Gbps achieved when running 4000 PEX. During the test, all the 8 CPU cores were completely loaded, regardless of the number of PEX executed.

In this test, the DPI becomes the bottleneck of the system, since it executes complex tasks in order the check the regular expressions in each packet. This results in a throughput that mainly depends on the amount of offered load, irrespective on the number of PEX running, as evident from the small performance worsening when moving from 10 to 4000 PEX. As a consequence, the effects of the fvSwitch memory access patterns, which were the main cause of the performance reduction in tests discussed in Section 1.7.1, are now limited.

1.7.4 VNFs and execution environments

To validate our choice of running all the VNFs deployed by the same player within a single PEX, we measured the throughput and the latency in two different conditions: *(i)* the traffic is processed in 5 VNFs, all deployed in the same PEX; *(ii)* the traffic is processed in 5 VNFs, running in 5 different PEX (as suggested by the NFV model). The throughput achieved is shown in Table 1.3; as evident, in the former case performance are definitely better, regardless of the packets size. Figure 1.13 shows instead the latency introduced by FROG in case of 64B packets; also in this case, it is evident the advantage of running several VNFs (i.e., those belonging to the same player) within the same PEX.

Table 1.3: "One VNF per PEX" vs "Multiple VNFs per PEX".

	64B	**700B**	**1514B**
1 PEX - 5 VNF	7.5Gbps	10Gbps	10Gbps
5 PEX - 5 VNF	2.6Gbps	4.6Gbps	4.9Gbps

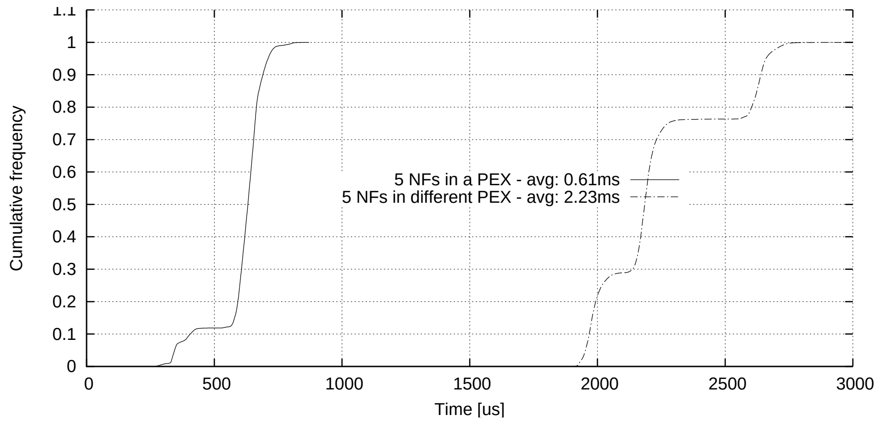


Figure 1.13: Latency in case of 5 VNFs running: *(i)* in the same PEX, *(ii)* in 5 different PEX.

1.7.5 FROG vs NetVM and ClickOS

This section tries to compare FROG with ClickOS and NetVM, which are all software architectures that allow the deployment of generic VNFs on the same hardware.

According to [55], NetVM achieves a throughput of about 7Gbps when moving (in a zero-copy fashion) 64B packets through a chain of 5 VNFs. In similar conditions, FROG provides a throughput of 7.5Gbps (first line of Table 1.3)[12]. Although these numbers are similar, our framework definitely requires few hardware resources than NetVM. For instance, NetVM exploits 2 cores to manage the network interfaces, 4 cores to move traffic between VNFs, while each VNF uses two cores. Instead, by design the fvSwith and each PEX are single thread, in order to support thousands of concurrent players. Moreover, NetVM deploys VNFs as full fledged VMs, while PEX are just processes executed in lightweight Docker containers.

Unfortunately, results obtained with FROG cannot be compared with those achieved with ClickOS. In fact, [64] only shows the throughput in case of many packet generator/receiver VNFs, or with packets traversing a chain of VNFs that

[12]Remember that FROG moves packets with zero-copy among VNFs instantiated in the same PEX.

does not involve the network (in this case the traffic source and generator are VNFs as well, executed on the same physical server together with the other VNFs of the chain). We did not execute tests in these conditions because, in real deployments, VNFs are passthrough applications operating on traffic coming from the network, and that goes again on the network after its processing.

1.8 Conclusion

This chapter presents FROG, the software architecture of a network edge node that gives to multiple players the possibility to execute their virtual network functions on a portion of the network traffic (called tile). Particularly, end users connected to FROG can create their customized network services that are then applied to their own traffic independently from the physical terminal in use (e.g., smarthphone, tablet, ect.).

On FROG, each player is provided with a lightweight execution environment (the PEX), which runs all the VNFs installed by the player himself and that just operates on the traffic matching a specific tile. Isolation among PEX is provided through Docker containers; this way, malicious VNFs installed by a player cannot "damage" the VNFs and traffic of other players. Furthermore, PEX can have different permissions on the packets on which they operate. Notably, since the PEX of an end user connected to the Internet through a FROG node is the first one processing the traffic coming from the end user device, and the last one that handles the traffic towards his terminal, such a PEX can be seen as an extension of the TCP/IP stack of the user device moved in the network.

The chapter describes the general architecture of the node, a prototype implementation, and the experimental evaluation of the system. Based on our results, a model that allocates a PEX to each player, and which supports until eight thousand of active PEX on the same physical machine, proves to be feasible.

Among the possible future works, we can envision the following directions. First, more efficient lightweight execution environments are advisable: in fact the Java PEX, unless the many features exported, may not seem completely adequate, particularly with respect to the memory consumption. Second, the latency introduced by both the PEX implementations should be reduced as well.

Chapter 2

Moving applications from the host to the network: experiences, challenges and findings

2.1 Introduction

Chapter 1 presented FROG, the software architecture of a network edge node that enables several players (e.g., end users, network service providers) to install their own virtual network functions (VNFs) operating on a portion of the packets traversing the node itself. This chapter[1] aims instead at validating the potentialities of FROG, and in particular of the Private EXecution environment (PEX), from the point of view of the VNF developer, and it presents our experience in developing a complex application-layer service.

First, the chapter describes the programming architecture of FROG, showing the main functions offered to programmers and a brief overview of the API. Second, it presents our experience in implementing a complex service based on such an API, namely a parental control, and shows a preliminary characterization of its performance. The parental control was chosen because it is a service that can take many advantages from the possibility to be executed in the network instead of in user devices. In fact, by being executed in the network, the parental control is able to inspect all the traffic to/from the users that need to be protected (e.g., kids), regardless of the device their are using to connect to the Internet, as well as their access network (e.g., domestic WLAN, 4G, etc.). Furthermore, our prototype allows to safely share the same physical device (e.g., a tablet) among many users (e.g., kids, parents), as the network is able to recognize the users and install the proper VNFs operating on their traffic.

[1]The content of this chapter has already been published in [31].

This chapter is structured as follows. Section 2.2 presents the related work, focusing on the existing parental control services and highlighting their limitations with respect to our proposal, in which the parental control is deployed as a VNF inside a box at the edge of the network. Section 2.3 summarizes the main concepts of FROG and it presents the new extensions that were needed to implement the service we had in mind. Section 2.4 presents the programming architecture and an overview of the exported API, while Section 2.5 describes our parental control service. Finally, Section 2.6 shows some numbers that come from the deployment of the parental control service and Section 2.7 draws some conclusive remarks.

2.2 Related Work

To validate FROG from the point of view of the VNF developer, we chose to develop a parental control service; even if many implementations of this application already exist (installed on user terminals, but also deployed within the network, i.e., on routers and proxy servers), they all suffer of several limitations. Solutions such as *Net Nanny* [36], *safeeyes* [67] and *Davide.it* [2] are installed on the end users terminals, and they may not be able to properly protect kids because of the many different devices owned by each person. In fact, parental controls could operate in different ways and/or offer different degrees of protection on different devices and, in some cases, they may not even exist on some platforms. Among the router-based solutions, we can cite manufacturers like Cisco [1] and Netgear [74], which offer parental control services running on their devices. However, these applications are poorly customizable and implement only the features decided by the manufacturers themselves. Users cannot upgrade the service with new features and have to wait for the manufacturer to implement them. Other solutions are based on a proxy server, such as *DansGuardian* [24]. In this case the web browser on the user terminal must be configured to send all requests to the proxy where the parental control application is running, rather than directly to the destination web server. However, the configuration on the client terminal can be easily bypassed by reconfiguring the application in order to access the Internet directly. Furthermore, this solution is limited to the protection from threats coming from web traffic. Finally, we can cite services that filter DNS requests (e.g., *OpenDNS* [12]), which cannot block applications that do not use the DNS such as instant messaging and others.

A parental control service installed as a VNF on a network edge node like FROG has many advantages compared to the existing approaches. First, it is able to protect kids regardless of the physical device they use to connect to the Internet, as well as to recognize the user connected to the network and act differently according to his profile. In fact, FROG requires the user to go through an authentication phase before being able to connect to the network, which allows the system to detect the user identity and to install exactly the VNFs associated with him. Second, it has the potential to protect kids independently from the location they connect to the Internet. Third, it enables a fine tuning of the service by allowing end users to install the applications they want, e.g., with additional or more advanced capabilities. Finally, it is able to protect children independently from the applications they use, as it operates on all the network traffic.

2.3 The Flexible and pROGrammable edge device

This section summarizes the main concepts of the Flexible and pROGrammable edge device (FROG) (Chapter 1), and presents the new extensions related to the remote execution environment and the storage service.

The logical architecture of FROG is depicted in Figure 2.1, and includes the following main components. The **Private EXecution environment (PEX)** executes the VNFs belonging to a single user; in addition, it enforces the privileges granted to the VNFs executed into the PEX itself, such as the possibility to modify and/or drop the traffic in transit. The **FROG virtual switch (fvSwitch)** is instead the module in charge of partitioning the network in tiles, and then redirect the incoming traffic to the PEX associated with all the tiles the packet belongs to.

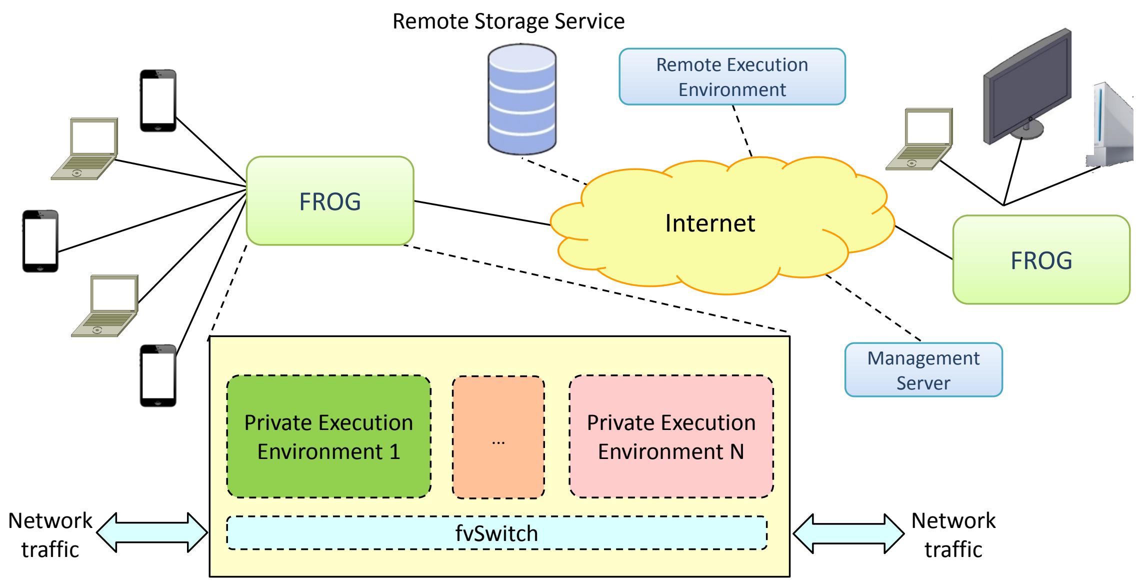


Figure 2.1: Overview of the entire system.

Moreover, a **Management Server** exists that is in charge of keeping the user database and handling a part of the authentication process. In general, this entity coordinates the entire set of FROG nodes located at the edge of the network; in fact, it contains also the list of VNFs to be installed, together with other information needed to manage the system.

When we started writing some more complete applications for FROG, we recognized that the architecture presented so far was missing some important features. For instance, we felt we needed a *remote storage service* and a *remote execution environment*, which are detailed in the following.

2.3.1 Storage Service

One of the modules we developed for our parental control service is *"DNSFilter"* (detailed in Section 2.5), which basically filters DNS requests and rejects those that refer to names included in a forbidden list. While at the beginning we included the blacklist in the application itself, we recognized that it was much better to keep that information on a remote storage. In general, VNFs may need to store persistent data, such as state information or configuration parameters that must be preserved across multiple execution of the same service. In the current system, VNFs running on FROG have only the "volatile" storage provided by the variables that are defined in the application itself.

As a consequence, we added a **Remote Storage Service** (RESTO), whose internal architecture is depicted in Figure 2.2. The RESTO includes the **Authentication Manager**, i.e., the component that authenticates the couple user/VNF (more details in Section 2.3.3) and the **Resource Manager**, which is responsible for reading/writing data upon requests from VNFs, as well as for deleting information when such an application is uninstalled. The RESTO service organizes the data in a tree based on the *VNF* that generated it and on the *user* who is executing that application in his own PEX. Vice versa, multiple instances of the same VNF running on different PEX associated with the same user (e.g., a user connected to the network through a smartphone and a laptop) share the same data; the implementation of different storage areas for those instances are under the responsibility of the programmer. The "remote" characteristic of the service enables VNFs to access their persistent data independently from the FROG node they are running on, and it enables that data to be accessed also from other parties (e.g., other services residing in the cloud).

The RESTO module is deeply integrated with the rest of the platform and it allows programmers to read/write data with simple primitives, while other issues (such as the authentication process, which guarantees that a user can write/read only his data) are under the responsibility of the system and are transparent to the programmer. The system takes also care of cleaning up the data associated with a given VNF/user when that VNF is removed from the user and hence does not longer belong to the PEX associated with his slice. It is worth noting that we do not force programmers to exploit the RESTO; in fact, they are still enabled to save their data

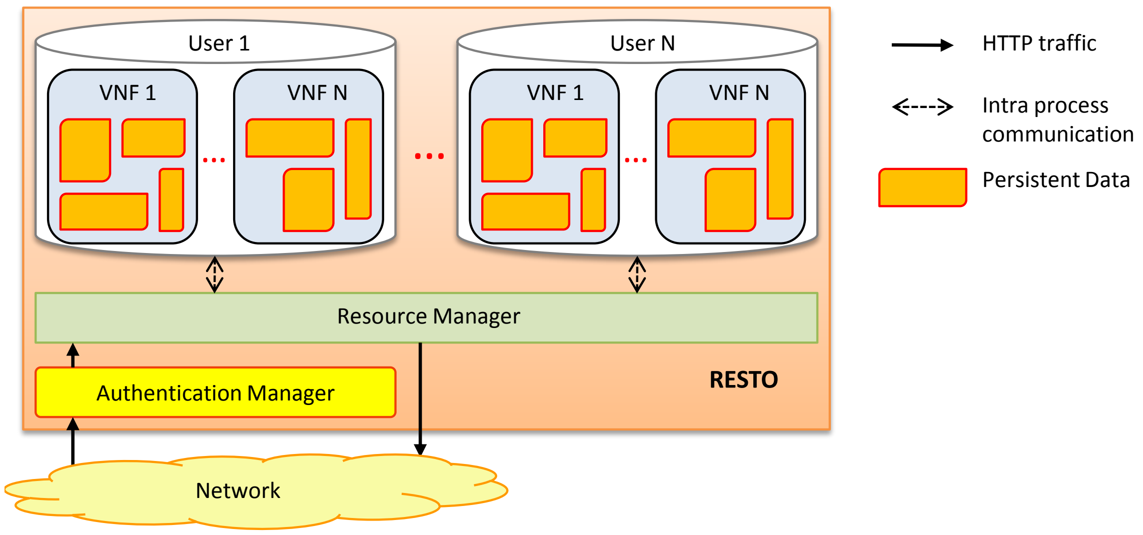


Figure 2.2: Internal view of the Remote Storage Service.

wherever they want. This way, however, they must address by themselves all the issues already solved by our platform.

2.3.2 Remote Execution Environment

The *DNSFilter* module mentioned before showed another problem. As the list of forbidden sites was rather large and was consuming a huge amount of memory on the edge node, it would be better to split the application into a FROG part, with a short list, and a server part, which keeps the full list and that is invoked upon demand. This suggested us that there may be a class of VNFs that can be split in multiple portions hosted on different locations, such as a part running on a FROG node and another running on a remote server.

While, in line of principle, the programmer can implement this splitting by setting up a remote service and modifying the VNF running on FROG in order to access to that service, we decided to offer him another possibility that looks more integrated with our solution. Then, we defined a **Remote Execution Environment** (REX), a module that can host the "server" part of VNFs and that can offer some standard functions to programmers. This module, whose architecture is shown in Figure 2.3, provides a Java-based execution environment very similar to the PEX, and also the exported API has similarities with the one present in that module. For instance, obviously no primitives are available for reading/modifying network packets, but others (e.g., the API for accessing the RESTO service) are the same in both the REX and PEX environments.

Finally, the REX programming environment hides both the communication between the code executed in the PEX and the one running in the remote environment, and all the authentication/authorization issues.

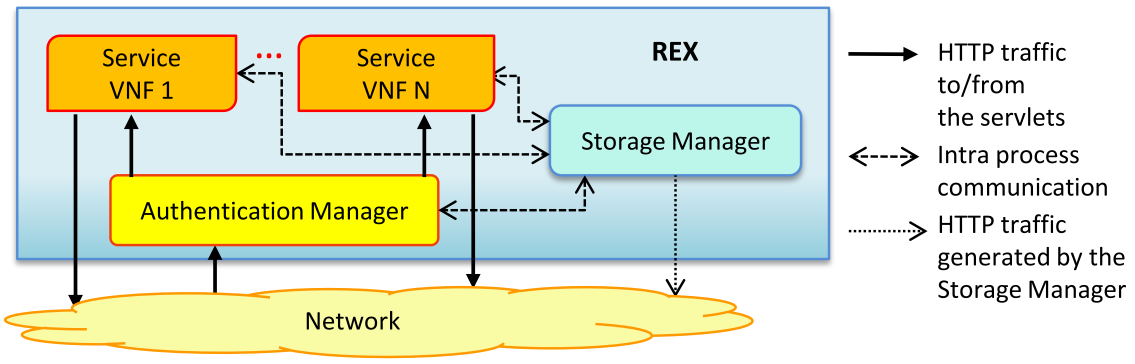

Service VNF 1
Service VNF N
REX
Storage Manager
Authentication Manager
Network
HTTP traffic to/from the servlets
Intra process communication
HTTP traffic generated by the Storage Manager

Figure 2.3: Exploded view of a REX.

2.3.3 Communication and authentication

Programmers access the REX and the RESTO through an API that hides both the details of the communication (which occurs through HTTP) and the authentication process. With respect to the authentication, the Management Server randomly generates a secret key when the user logs in, which is shared between the user's PEX, RESTO and REX services. This secret allows the remote components to identify the user and the VNF when they receive a request from the PEX.

In fact, we insert in the HTTP requests information such as the *username* of the user who is running the VNF that requested the service, the *VNF name* identifying the VNF that makes the request, a PEX identifier that uniquely identifies the PEX in which the application is running, the *FROG identifier* that uniquely identifies the VNF as being executed on a given FROG node. Finally, part of those information and the secret key mentioned before are given as input to the `sha-256` algorithm in order to generate an unique signature that will be used by the remote party as authentication key.

It is worth noting that all these parameters are under the control of the PEX and are not visible by the VNF. As we suppose that the PEX is trusted (while the user-provided code running in it may not), we can safely assume that those parameters are enough to guarantee the proper interaction with the remote services, at least in our prototype. In fact, when an HTTP message reaches the remote service, the authentication module is able to recognize the user / VNF / instance of VNF that asked for the service. Then, it forwards the request to the proper service handler, being it a storage module or a service in the REX.

2.4 Programming the PEX

This section describes the programming architecture of the (Java) PEX, by detailing the various components depicted in Figure 2.4 and by providing an overview of the exported API.

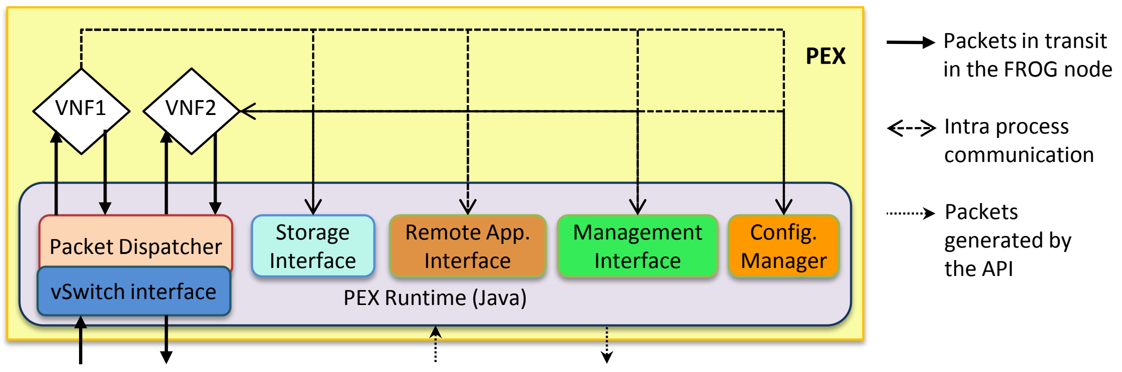


Figure 2.4: Exploded view of a PEX hosting two VNFs.

2.4.1 Callbacks

As the PEX exports an event-driven programming model, a VNF is requested to implement a set of callbacks that are called when specific events occur. In particular, `OnStartUp` and `OnShutDown` must contain the code that has to be executed when the VNF is started/stopped as a consequence of user's commands. `OnReceivedPacket` is instead called when a new packet is available in the system and needs to be processed. In this case, the VNF receives the packet and a set of metadata such as the physical port of the FROG node on which that packet was received. This method must return `DROP` or `CONTINUE`, depending on whether the packet must be discarded or it can be forwarded to the next recipient, which can be either the VNF that follows in the same PEX or the fvSwitch, in case that the VNF was the last one in that PEX.

2.4.2 PEX Runtime

The PEX runtime creates the (Java) environment on which VNFs are executed, and it includes several modules that can be exploited by VNFs through the proper API (e.g., the interface toward the REX/RESTO services), the packet dispatcher and the interface toward the fvSwitch. Particularly, the latter exchanges packets with the fvSwitch by means of highly optimized buffers, as described in Chapter 1. Finally, the **Configuration Manager** implements a set of REST services that enable users to *(i)* install/uninstall VNFs, *(ii)* change their calling order and *(iii)* start/stop VNFs already installed.

2.4.3 Packet Dispatcher

The **Packet Dispatcher** is the component in charge of delivering packets to the VNFs by calling their `OnReceivePacket` handler each time that a new packet reaches the PEX. As PEX, and then VNFs, can be associated with different privileges (e.g., packets can be received in read-only mode), the `GetPrivileges` method allows VNFs to know their privileges and act accordingly. Since a VNF may ignore this information and perform illegal actions, this module also implements techniques to ensure that privileges cannot be violated. Finally, the Packet Dispatcher exports the `RegisterFilter` and `UnregisterFilter` methods, which enable VNFs to receive only the packets matching a given filter (e.g., HTTP traffic); however, an efficient implementation of this function is left to future work.

2.4.4 Storage Interface

The **Storage Interface** implements the communication towards the RESTO mainly through the intuitive `SaveData` and `ReadData` methods. Data to be stored must be Java objects implementing the interface `Serializable`, or also Java primitive data types such as `int` and `float`; this way, the system is able to manage any kind of data, even objects defined by programmers. Optionally, data can be stored with some additional metadata such as the timestamp in which the data was modified, and the responsible of that change (in terms of PEX identifier and FROG identifier). In order to manage the concurrent access to data, the Storage Manager also exports the `LockResource` and `UnlockResource` methods, which can be used to implement atomic modification on that data even in presence of multiple running VNFs associated with that user, e.g., in case multiple terminals associated with the same user are connected to the network. Finally, this interface implements all the mechanisms required for remotely authenticate the user on the RESTO, as explained in Section 2.3.3.

2.4.5 Remote Application Interface

Similarly to the previous component, the **Remote Application Interface** handles the interaction with services hosted in the REX. This module exports the simple `Get`, `Post`, `Put` and `Delete` methods, which derive from the HTTP methods defined in a REST interface. Those methods create the appropriate HTTP request for the resource specified as a parameter and return to the application the HTTP response coming from the remote service. The remote URL is partially created automatically by the PEX (e.g., the application name), while other information are set by the VNF (e.g., the part identifying the resource and the additional parameters that may be needed). Also in this case this component hides the entire authentication process to the VNFs.

2.4.6 Management Interface

The **Management Interface** enables each VNF to implement the primitives that can be used to configure or monitor the service from the external world. In fact, each VNF can be reached by the PEX owner by typing the standard URL

```
http://config.ctrl/vnf_name/
```

The system will check the request for permissions, then redirects it to the REST services exported by VNF. It is worth noting that both the semantic and the syntax of the data exchange is completely application-dependent.

2.5 Parental control service

The parental control service developed on top of the FROG API consists of the following modules.

GSafe exploits the *Google safe search* [47] feature of the Google search engine to filter harmful contents. When active, GSafe enables the safe search by changing the URL in all the HTTP `GET` messages towards Google and related to a search. In particular, the URL is extended with `safe=active` or `safe=strict`, depending on the selected level of protection specified in the configuration parameters. Currently, the implementation can operate only on packets that (after the modification) do not exceed the MTU. We plan to introduce a stream reassembly function in the API in the future, in order to enable programmers to avoid this issue and then to allow them to operate also on *messages* in addition to *packets*.

DNSFilter prevents children from reaching disturbing websites whose URLs are included into a blacklist. As URLs are organized by topic (e.g., porn, drug, etc.), a configuration parameter (visible to parents) can be used to enable/disable one or more sections. We implement this application both in a *monolithic* version, entirely running as a VNF in the PEX, and in a *split* version in which DNS packets are received by the VNF on the FROG node, checked against a small cache, and in case of miss the request is redirected to a remote service on the REX. Based on the result of the check, the VNF can drop the DNS message, or let it go on its way. In addition, DNSFilter gathers all the DNS names translated, in order to allow parents, through the web interface, to inspect which sites were accessed by their children.

TimePeriod can block the access to the Internet during a given time slot, as well as it can limit the amount of time for which kids can be logged in into the system. For instance, a child could be enabled to navigate only from 2 pm to 9 pm;

in addition, he could be allowed to spend no more than two hours per day on the Internet. Similarly to the previous VNFs, TimePeriod exports a web interface that enables parents to configure the application itself.

Finally, **SkypeBlocker** is able to identify and discard the Skype traffic according to the signature defined in `nDPI` [19].

All these components of the parental control service exploit the RESTO in order to store the configuration parameters selected by the parents.

2.6 Validation

This section validates the FROG platform and the parental control service through different categories of test. The test set up consists in a Fast Ethernet network that includes a user laptop directly connected a FROG node; a set of servers implementing the DNS server, RESTO, REX and the management server are directly connected to FROG. All the servers are workstations with an Intel Core2 processor (Q8400 at 2.66MHz), FROG runs an Intel i5 3450S at 2.8GHz, while the laptop is a Intel Core2 P8700 at 2.53 GHz. All the machines have 4GB RAM and a 7200rpm hard disk in the range 250-320 GBytes; moreover, they were preloaded with the Linux Debian 7 operating system running at 64 bits.

2.6.1 Starting the PEX

This test measures the time required to activate a PEX upon the receipt of a successful login from a new user. When the system authenticates the user through the captive portal, it starts a new PEX on that FROG node and it activates all the VNFs associated to that user. This process includes several steps summarized in the top part of Figure 2.5, such as the time required to check the user credential in the management server, the time needed to configure the environment to host a new PEX, the time needed to start a new PEX with all the requested VNFs, and finally the time needed to create the tile for the user and map this to the newly created PEX. Although we agree that our implementation can be improved, everything completes in less than 5 seconds in our operating conditions. Particularly, the time required to start each VNF (which requires downloading it from the management server, injecting it in the existing PEX, and starting it) is negligible compared to that needed to completely activate the execution environment. In fact, the worst time we get refers to the DNSFilter application, which completes this process in 195 ms. Furthermore, many tasks such as installing VNFs are launched in parallel, contributing to keep the overall duration low.

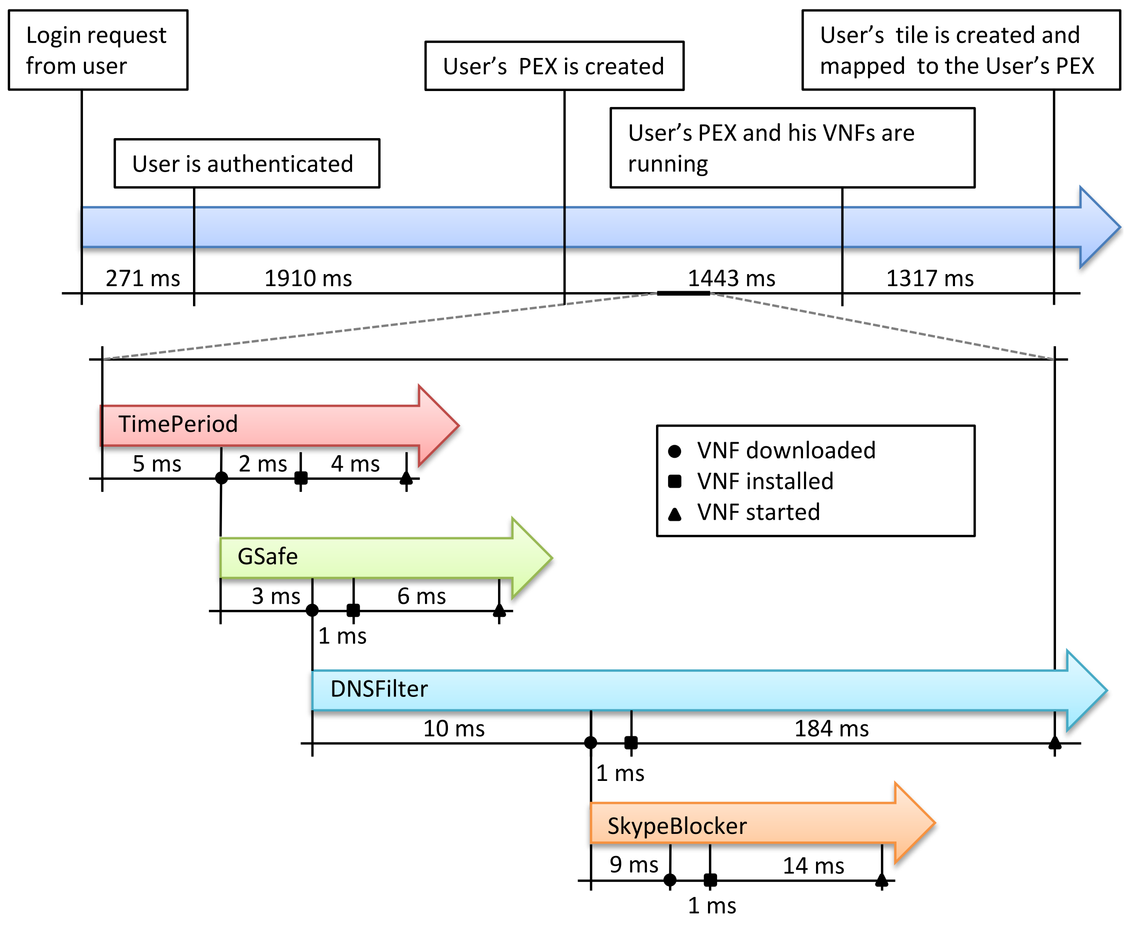


Figure 2.5: Starting a PEX with four VNFs.

2.6.2 Accessing the RESTO

This test shows the latency introduced when a VNF uses the RESTO. In order to obtain this time, we wrote a simple program that saves and reads resources of different sizes to/from the storage service each time a packet is received. Gathered numbers take into account the worst condition as we read resources that were not in the cache of the RESTO.

The application was repeated thousand times and the numbers were averaged in order to obtain the results shown in Table 2.1. As expected, readings are always slower than writings, and the latency grows with the size of the managed resource. Numbers confirm also that the RESTO service is most appropriate for VNFs that need occasional access (e.g., to store/load configuration parameters), while it may not be appropriate for a VNF that requires an access to this service each time a new packet is received.

Table 2.1: Latency in accessing the RESTO.

	10B	**100B**	**1KB**	**10KB**	**100KB**
write [ms]	3.49	4.27	6.15	17.25	129.61
read [ms]	2,29	3,24	4,29	14,24	115,59

2.6.3 Exploiting the REX

This test evaluates the impact of the REX in terms of memory saving on the FROG node and of latency. For this purpose, we run both the monolithic and the split flavors of the DNSFilter.

Since the split VNF does no longer use the blacklist within the PEX, the memory consumption at the edge node decreases considerably: 141MB with the monolithic version, against 24MB with the split VNF. On the other hand, time needed to serve the DNS request increased from 1.4ms to 7.8ms, due to the additional steps (e.g., creating and sending the HTTP request, etc.) required to obtain the answer. These results were obtained by averaging numbers coming from thousand queries toward the local DNS server set up in our network, which was configured to answer to all the queries without forwarding them to the Internet, hence avoiding any issue not under our control.

2.7 Conclusion

This chapter presents our experience in developing a parental control service for the FROG framework presented in Chapter 1. This work allowed to test, with the eyes of the VNF developer, the validity of our platform. In fact, we felt the necessity to extend the programmable environment in order to accommodate some additional requirements of the selected parental control service, which have not been foreseen in our original prototype. In particular, we added the *Remote Storage Service* (RESTO), which enables VNFs to save their persistent data, i.e., information that must be maintained among different executions of the VNFs themselves. Experimental results show that an occasional access to the RESTO (e.g., reading/writing configuration parameters) does not significantly reduce the performance of VNFs. We also defined the *Remote Execution Environment (REX)*, which enables VNFs to exploit a remote service (not executed in a PEX running on a FROG node) for their purposes. Thanks to the REX, we were able to partition VNFs into an “edge” and a “cloud” portion, hence reducing the hardware requirements on the FROG node (e.g., in terms of memory consumption), although at the cost of introducing some additional latency in the VNFs.

Currently the PEX, and then the VNFs, receive traffic *packet by packet*, resulting in a huge amount of work when the payload has to be modified (e.g., adding some bytes to an HTTP request). In fact low level tasks such as TCP reassembly and session tracking have to be implemented from scratch in each VNF. Then, we plan to add these functionalities inside the PEX, and enrich the API it exports to VNFs so that VNF developers can concentrate on the logic of theirs applications, without taking care of these (annoying but needed) tasks.

Chapter 3

Toward dynamic and virtualized network services in telecom operator networks

3.1 Introduction

Network Function Virtualization (NFV) [43] is bringing a new breath of fresh air in the networking field. In fact, thanks to their software-based nature, Virtual Network Functions (VNFs) could be potentially deployed on any node with computing capabilities located everywhere in the network, ranging from the home gateway installed in the customer premises till to the data center servers [42].

Although NFV is mainly seen as a technology targeting network operators, which can deliver network services with unprecedented agility and efficiency while reducing OPEX and CAPEX, end users (e.g., xDSL customers) can benefit from NFV as well, as this would enable them to customize the set of services that are active on their Internet connection. But, while NFV currently focuses mostly on middlebox-based applications (e.g., NAT, firewall), end users are probably more oriented to services based on traditional network facilities (e.g., L2 broadcast domains), which receive less consideration in the NFV world.

Motivated by the growing interest, e.g., of telecom operators, in extending the functionalities of Customer Premise Equipments (CPEs) in order to deliver new and improved services to the end users [25] [90] [85] [29], this chapter[1] presents a solution that is oriented to deliver *generic* network services that can be selected by *multiple players*. Particularly, our proposal enables the *dynamic* instantiation

[1]The content of this chapter has already been published in [35] and partially in [33]. This work is also partially described in the master thesis of Fabio Mignini and Matteo Tiengo, who collaborated in the development of the prototype.

of *per-user* network services on the large infrastructure of the telecom operators, possibly starting from the home gateway till the data center, as depicted in Figure 3.1. Our solution enables several players (e.g., telecom operator, end users, etc.) to cooperatively define the network services; moreover, it is general enough to support both traditional middlebox functions as well as traditional host-based network services. For example, a customer can define its own network service by asking for a transparent firewall and a Bittorrent client, while the network operator complements those applications by instantiating a DHCP and a NAT service[2].

In our solution, the entire network infrastructure is controlled by a service logic that performs the identification of the user that is connecting to the network itself, following the approach proposed in Chapter 1. Upon a successful identification, the proper set of network functions chosen by the user is instantiated in one of the nodes (possibly, even the home gateway) available on the telecom operator network, and the physical infrastructure is configured to deliver the user traffic to the above set of VNFs.

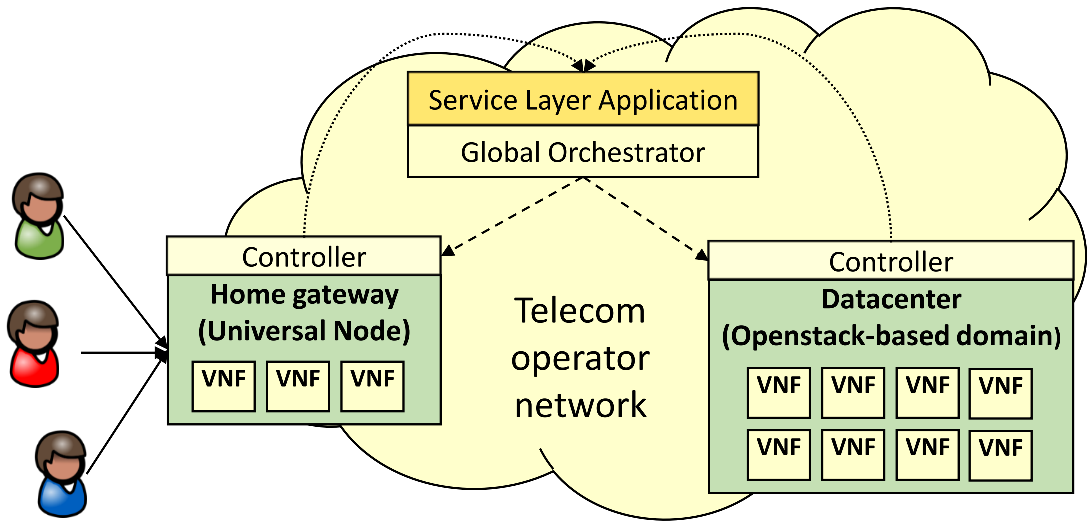


Figure 3.1: Deployment of virtual network functions on the telecom operator network.

The chapter describes the service-oriented layered architecture to achieve those objectives, modeled after the one proposed by the Unify project [89, 37], and a possible set of data models that are used to describe and instantiate the requested network services starting from an high-level and user-friendly view of the service.

[2]In this chapter we assume that end users are only enabled to select VNFs *trusted* by the operator. The case in which they can deploy *untrusted* VNFs (e.g., implemented by the end users themselves) would in fact open security issues that are beyond the scope of this work.

The high-level description is then converted into a set of primitives (e.g., virtual machines, virtual links) that are actually used to instantiate the service on the physical infrastructure. Moreover, it presents two possible implementations of the nodes of the infrastructure layer on which the service is actually deployed. Particularly, we explored two solutions that are based on different technologies, with different requirements in terms of hardware resources. The first is based on the OpenStack open-source framework and it is more appropriate to be integrated in (existing) cloud environments; the second exploits mostly dedicated software and it is more oriented to the domestic/embedded segment (e.g., resource-constrained CPEs).

The remainder of this chapter is structured as follows. Section 3.2 provides an overview of the related works, while Section 3.3 introduces an architecture to deploy generic network services across the whole network under the control of the telecom operator (as shown in Figure 3.1). Section 3.4 details some formalisms expressing the service to be deployed, which are then exploited to solve the challenges arising from our use case, as discussed in Section 3.5. Section 3.6 details the preliminary implementation of the architecture, which is then validated in Section 3.7, both in terms of functionalities and performance. Finally, Section 3.8 concludes the chapter and provides some plans for the future.

3.2 Related work

Three FP7 EU-funded projects focusing on the integration of the NFV and SDN [80] concepts (UNIFY [18], T-NOVA [17] and SECURED [16]) started recently. Particularly, the first one aims at delivering an end-to-end service that can be deployed everywhere in the telecom operator network, starting from the points of presence at the edge of the network, till to the datacenter, by exploiting SDN and NFV technologies. Similarly, T-NOVA proposes an equivalent approach that puts more emphasis on the target of providing a uniform platform for third-party VNF developers, while SECURED aims at offloading personal security applications into a programmable device at the edge of the network. An SDN-based service-oriented architecture has been proposed also in [83, 27], which enables to deliver middlebox-based services to user devices, leveraging Service Function Chaining and SDN concepts.

From the industry side, the IETF Service Function Chaining (SFC) [57] working group aims at defining a model to describe and instantiate network services, which includes an abstract set of VNFs, their connections and ordering relations, provided with a set of QoS constraints. Similarly, the European Telecommunications Standards Institute (ETSI) started the Industry Specification Group for NFV [41], which aims at developing the required standards and sharing their experiences of NFV development and early implementation. Based on the ETSI proposal, the Open Platform for NFV (OPNFV) project [75] aims at accelerating the evolution of NFV by defining an open source reference platform for Virtual Network Functions.

The problem of CPE virtualization, which represents one of the possible use cases of our architecture, is investigated in several papers (e.g., [25, 90, 85, 29]); however they have a more limited scope as do not foresee the possibility to instantiate the service across an highly distributed infrastructure and focus on more technological-oriented aspects.

Finally, the OpenStack [13] community is aware of the possibility to use that framework to deliver NFV services as well, as shown by the new requirements targeting traffic steering and SFC primitives [21, 20]; in fact, we rely on some preliminary implementation of those functions [62] in order to build our prototype.

3.3 General architecture

Our reference architecture to deliver network services across the telecom operator network, which follows closely the one proposed by the ETSI NFV working group [41], was defined in the FP7 UNIFY project [89, 37] and it is shown in Figure 3.2. As evident from the picture, it allows the deployment of network services through three main portions, namely the *service layer*, the *orchestration layer* and the *infrastructure layer*.

3.3.1 Service layer

The *service layer* represents the upper level component of our system and enables different players to define their own network services. Although our service layer includes some use-case specific functions such as user identification (detailed in Section 3.6.1), we introduce here the general concept of a generic service description expressed in an high level formalism called *service graph* (Section 3.4.1), which enables the definition of generic services (expressed independently by each player) and their potential interactions.

The service graph could be provided accompanied with several non-functional parameters. Particularly, we envision a set of *Key Quality Indicators (KQIs)* that specify requirements such as the maximum latency allowed between two VNFs, or the maximum latency that can be introduced by the entire service. We also foresee the definition of a list of high-level policies to be taken into account during the deployment of the service. An example of such policies could be the requirement of deploying the service in a specific country because of legal reasons.

Given the above inputs, possibly facilitated by some graphical tools that allow different players to select and build the desired service, the service layer should be able to translate the service graph specification into an orchestration-oriented formalism, namely the *forwarding graph* (Section 3.4.2). This new representation provides a more precise view of the service to be deployed, both in terms of computing and network resources, namely VNFs and interconnections among them, always

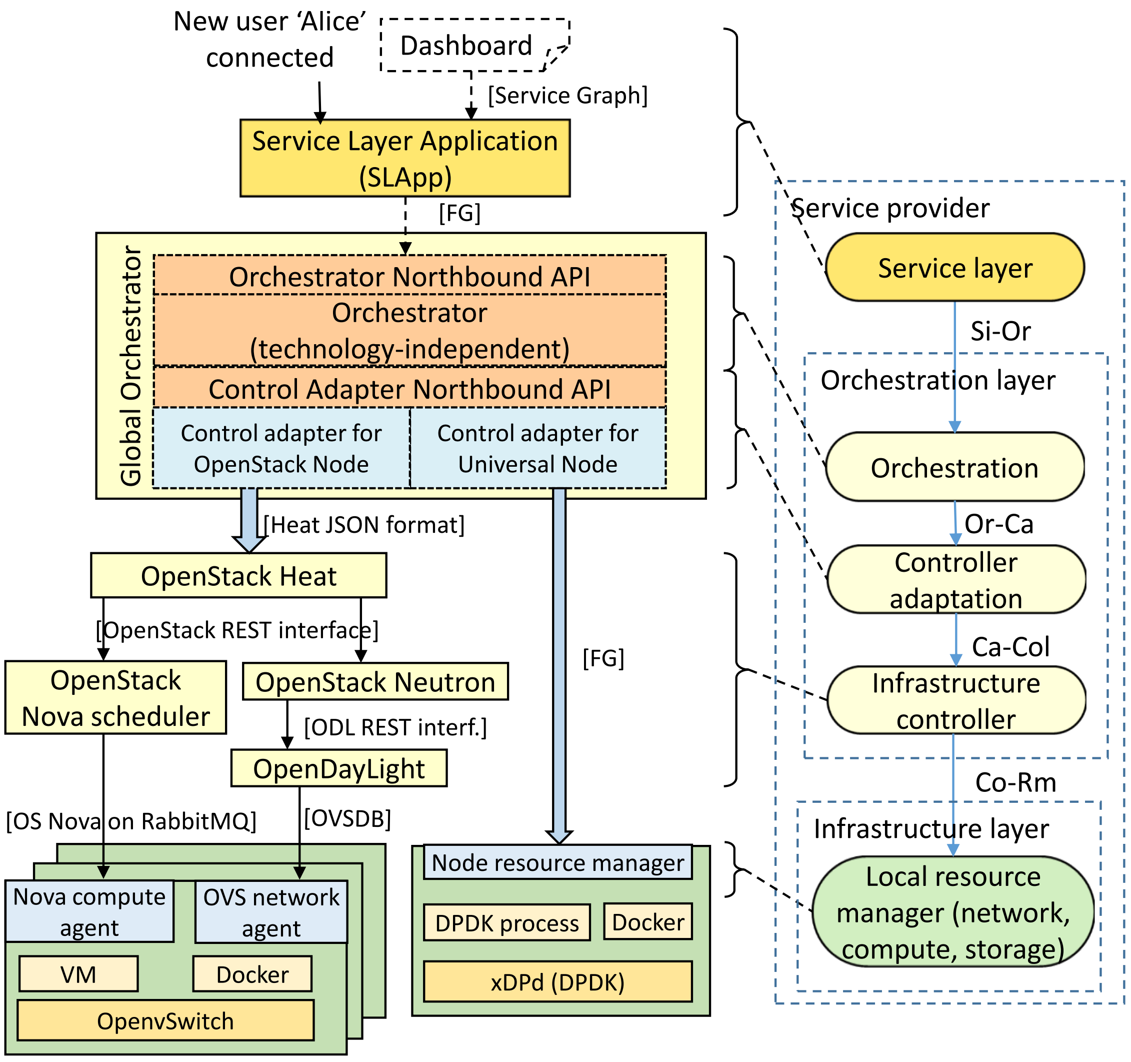


Figure 3.2: Overall view of the system, including the two implementations of the infrastructure layer.

conserving KQIs and policies imposed by the player that defined the service.

As depicted in Figure 3.2, the service layer includes a component that implements the service logic, identified with the service layer application (SLApp) block in the picture. The service layer could also export an API that enables other components to notify the occurrence of some specific events in the lower layers of the architecture. The SLApp module could react to these events in order to implement the service logic required by the specific use case. An example of such events may be a new user device (e.g., a smartphone) that connects to the network, which could trigger the deployment of a new service graph or the update of an existing one.

Finally, the service layer northbound interface enables also cloud-like services, as well as it offers to 3rd-party providers (e.g., content-providers) the possibility to deploy services in the operator infrastructure, orchestrating resources on demand and being billed for their utilization in a pay-per-use fashion.

3.3.2 Orchestration layer

The *orchestration layer* sits below the service layer and it is responsible of two important phases in the deployment of a service.

First, it manipulates the forwarding graph in order to allow its deployment on the infrastructure, adapting the service definition to the infrastructure-level capabilities, which may require the deployment of new VNFs for specific purposes, as well as the consolidation of several VNFs into a single one (Section 3.4.2). Second, the orchestration layer implements the scheduler that is in charge of deciding where to instantiate the requested service. The scheduling could be based on different classes of parameters: *(i)* information describing the VNF, such as the CPU and the memory required; *(ii)* high-level policies and KQIs provided with the forwarding graph; *(iii)* resources available on the physical infrastructure, such as the presence of a specific hardware accelerator on a certain node, as well as the current load of the nodes themselves.

According to Figure 3.2, the orchestration layer is composed of three different logical sub-layers. First, the *orchestration* sub-layer implements the forwarding graph transformation and scheduling in a technology-independent approach, without dealing with details related to the particular infrastructure, which is under the responsibility of the infrastructure layer. The next component, called *controller adaptation* sub-layer, implements the technology-dependent logic that is in charge of translating the forwarding graph into the proper set of calls for the northbound API of the different *infrastructure controllers*, which correspond to the bottom part of the orchestration layer. Infrastructure controllers are in charge of applying the above commands to the nodes operating on the physical network; the set of commands needed to actually deploy a service is called *infrastructure graph* (Section 3.4.3) and, being infrastructure-specific, changes according to the physical node/domain that will host the service that is going to be instantiated. The infrastructure controller should also be able to identify the occurrence of some events in that layer (e.g., unknown traffic arrives at a given node), and to notify it to the upper layers of the architecture. As shown in Figure 3.2, different nodes/domains require different infrastructure controllers (in fact, each resource has its own controller), which in turn require many *control adapters* in the controller adaptation sub-layer.

Having in mind the heterogeneity (e.g., core and edge technologies) and size of the telecom operator network, it is evident how the global orchestrator, which sits on top of many resources, is critical in terms of performance and scalability of the entire system. For this reason, according to the picture, the global orchestrator has syntactically identical northbound and southbound interfaces (in fact, it receives a forwarding graph from the service layer, and it is able to provide a forwarding graph to the next component), which paves the way for a hierarchy of orchestrators in our architecture. This would enable the deployment of a forwarding graph across multiple administrative domains in which the lower level orchestrators expose only some information to the upper level counterparts, which allows the architecture to potentially support a huge number of physical resources in the infrastructure layer. Although such a hierarchical orchestration layer is an important aspect of our architecture, it is out of the scope of this chapter and it is not considered in the implementation detailed in Section 3.6.2.

3.3.3 Infrastructure layer

The *infrastructure layer* sits below the orchestration layer and contains all the physical resources that will actually host the deployed service. It includes different nodes (or domains), each one having its own infrastructure controller; the global orchestrator can potentially schedule the forwarding graph on each one of these nodes. Given the heterogeneity of modern networks, we envision the possibility of having multiple nodes implemented with different technologies; in particular, we consider two classes of infrastructure resources.

The first class consists in cloud-computing domains such as the *OpenStack-based domain* in Figure 3.1, referencing one of most popular cloud management toolkits, each one consisting of a cluster of physical machines managed by a single infrastructure controller. The second class of resources is instead completely detached by traditional cloud-computing environments, representing nodes such as the future generation of home-gateways hosted in the end users' homes. One of such a node, called *Universal Node*, is shown in the bottom-left part of Figure 3.1 and consists of a single physical machine that is provided mostly with software written from scratch. Moreover, the infrastructure controller is integrated in the same machine hosting the required service.

The infrastructure layer does not implement any logic (e.g., packet forwarding, packet processing) by itself; in fact, it is completely configurable, and each operation must be defined with the deployment of the proper forwarding graph. This makes our architecture extremely flexible, since it is able to implement whatever service and use case defined in the service layer.

3.4 Data models

This section details the data abstractions that are used to model and then deploy the network services on the physical infrastructure. Our data models are inspired by the NFV ETSI standard [41], which proposes a service model composed of "*functional blocks*" connected together to flexibly realize the desired service. In order to meet the objectives described in the introduction, we instantiated the ETSI abstract model in multiple flavors according to the details needed in the different layers. All those flavors are inspired by the objective of integrating the functional component description of network services and their topology, together with the possibility to model also existing services provided by cloud computing.

3.4.1 Service graph

The **service graph (SG)** is a high level representation of the service that includes both aspects related to the infrastructure (e.g., which network functions implement the service, how they are interconnected among each other) and to the configuration of these network functions (e.g., network layer information, policies, etc.). Our SG is defined with the set of basic elements shown in Figure 3.3 and described in the following of this section. These building blocks were selected among the most common elements that we expect are needed to define network services.

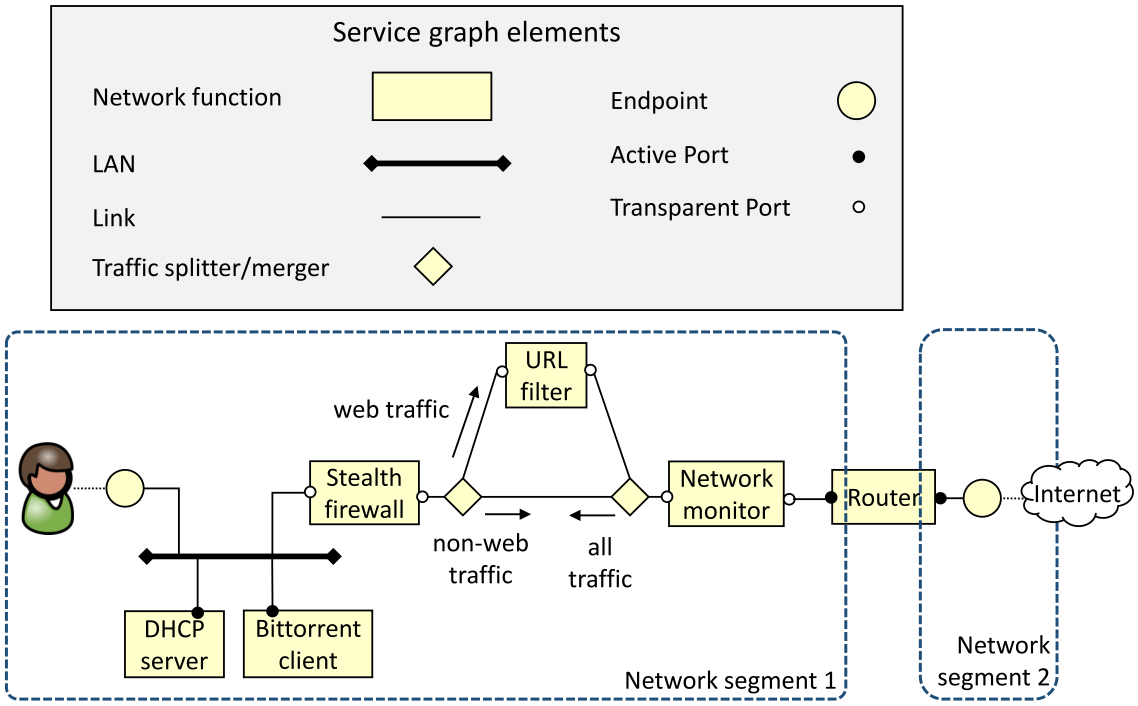


Figure 3.3: Service graph: basic elements and example.

The **network function** (NF) is a functional block that may be later translated into one or more VNF images or to a dedicated hardware component. Each network function is associated with a template (Section 3.4.4) describing the function itself in terms of memory and processing requirements, required processor architecture (e.g., x86-64), number and type of ports, etc.

The **active port** defines the attaching point of a NF that needs to be configured with a network-level address (e.g., IPv4), either dynamic or static. Packets directed to that port are forwarded by the infrastructure based on the link-layer address of the port itself.

The **transparent port** defines the attaching point of a NF whose associated (virtual) NIC does not require any network-level address. If traffic has to be delivered to that port, the network infrastructure has to "guide" packets to it through traffic steering elements, since the natural forwarding of the data based on link-layer addresses does not consider those ports.

The **local area network (LAN)** represents the (logical) broadcast communication medium. The availability of this primitive facilitates the creation of complex services that include not only transparent VNFs, but also traditional host-based services that are usually designed in terms of LANs and hosts.

The **point-to-point link** defines the logical wiring among different components and can be used to connect two VNFs together, to connect a port to a LAN, and more.

The **traffic splitter/merger** is a functional block that allows to split the traffic based on a given set of rules, or to merge the traffic coming from different links. For instance, it is used in Figure 3.3 to redirect only the outgoing web traffic toward an URL filter, while the rest does not cross that NF.

Finally, the **endpoint** represents the external attaching point of the SG. It can be either a logical entity or a specific port (e.g., a physical/virtual NIC, a network tunnel endpoint), active on a given node of the physical infrastructure. An endpoint can be used to attach the SG to the Internet, to an end user device, but also to the endpoint of another service graph, if several of them have to be cascaded in order to create a more complex service. Each endpoint is associated with an *identifier* and an optional *ingress matching rule*, which are required for the SGs attaching rules to operate (detailed in Section 3.4.1).

Figure 3.3 provides a SG example with three NFs connected to a LAN, featuring both active (e.g., the DHCP server and the bittorrent machine, which need to be configured with IP addresses) and transparent ports (the stealth firewall). The outgoing traffic exiting from the stealth firewall is received by a splitter/merge block, which redirects the web traffic to an URL filter and from here to a network monitor, while the non-web traffic travels directly from the stealth firewall to the network monitor. Finally, the entire traffic is sent to a router before exiting from the service graph. In the opposite direction, the traffic splitter/merger on the right will send *all* the traffic coming from Internet to the stealth firewall, without sending anything to the URL filter as this block needs to operate only on the outbound traffic.

As cited above, the SG also includes aspects related to the configuration of the NFs, which represent important service-layer parameters to be defined together with the service topology, and that can be used by the control/management plane of the network infrastructure to properly configure the service. In particular, this information includes network aspects such as the IP addresses assigned to the active ports of the VNFs, as well as VNF-specific configurations, such as the filtering rules for a firewall.

The SG can be potentially inspected to assess formal properties on the above configuration parameters; for example, the service may be analyzed to check if the IP addresses assigned to the VNFs active ports are coherent among each other. To facilitate this work, the SG defines the **network segment**, which is the set of LANs, links and ports that are either directly connected or that can be reached through a NF by traversing only its transparent ports. Hence, it corresponds to an extension of the broadcast domain, as in our case data-link frames can traverse also NFs (through their transparent ports), and it can be used to check that all the addresses (assigned to the active ports) of the same network segment belong to the same IP subnetwork. As shown in the picture, a network segment can be extended outside of the SG; for instance, if no L3 device exists between an end user terminal and the graph endpoint, the network segment also includes the user device.

Cascading service graphs

As introduced above, the SG endpoints are associated with some parameters that are used to connect SGs together (*cascading graphs*). Particularly, the *identifier* is the foundation of the SG attaching rules, as only the endpoints with the same identifier (shown with the same color in Figure 3.4) can be attached together. Instead, the optional *ingress matching rule* defines which traffic is allowed to *enter* into the graph through that particular endpoint, e.g., only the packets with a specific source MAC address.

The rules that define how to connect several graphs together change according to both the number of graphs to be connected and the presence of an ingress matching rule on their endpoints. While the case for two endpoints directly connected looks straightforward, the problem with three or more endpoints is more complex. Figure 3.4 shows two examples in which three endpoints must be connected together. In the first case, two egress endpoints are associated with an ingress matching rule that specifies which traffic must *enter* into the graph through that endpoint. This ingress matching rule must be used, in case of traffic going from the right to the left, to deliver the desired packets to the correct graph, notably HTTP traffic to the “HTTP-SG” and FTP traffic to the “FTP-SG”. This is achieved by transforming the ingress endpoint of the “TCP-SG” into the set of components enclosed in the green shape of Example 1(b), namely a traffic splitter/merger module attached with many new endpoints, each one connected to a different graph. This way, the common “TCP-SG” will be able to dispatch the packet answers to the proper graph.

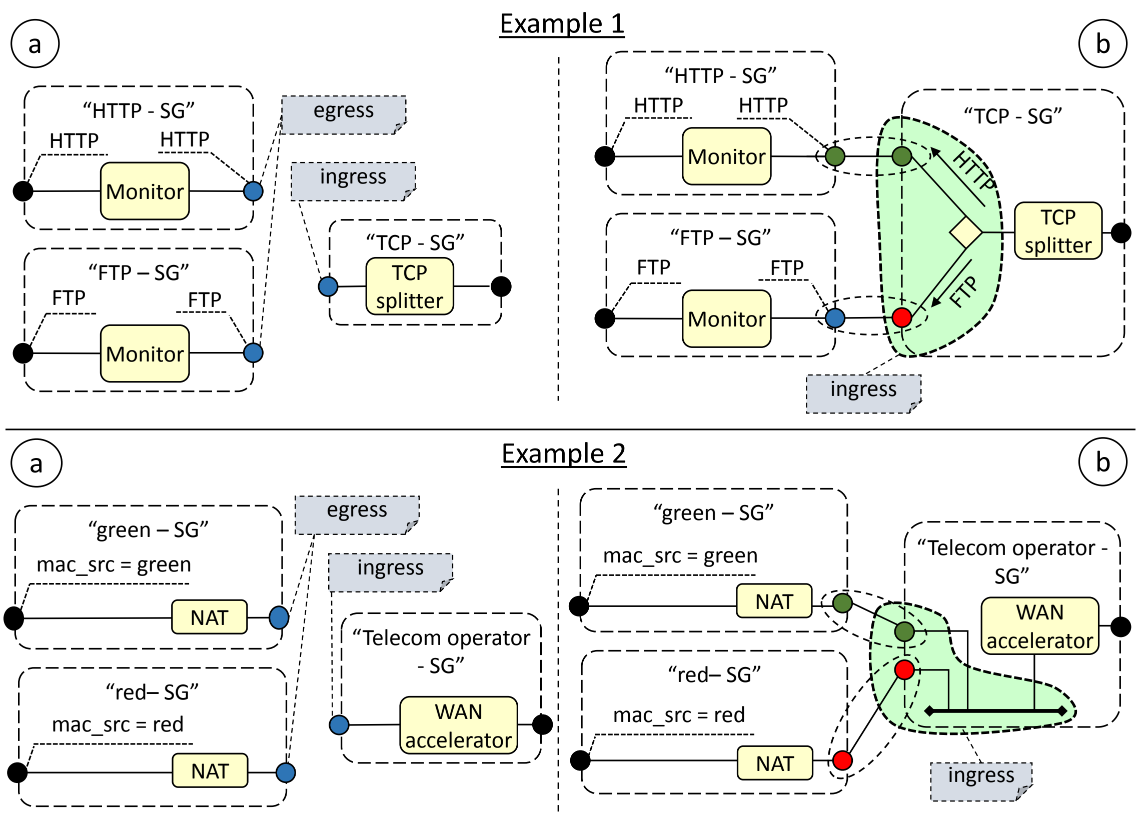


Figure 3.4: Cascading SGs.

The second example in Figure 3.4 shows the case in which the egress endpoints are not associated with any ingress matching rule, which makes it impossible to determine the right destination for the packets on the return path, as a traffic splitter/merger module cannot be used in the "telecom operator-SG" to properly dispatch the traffic among them. In this case, the ingress endpoint of the common "telecom operator-SG" is transformed into a LAN connected to several new endpoints, each one dedicated to the connection with a single other graph. This way, thanks to the MAC-based forwarding guaranteed by the LAN, the "telecom operator-SG" can dispatch the return packets to the proper graph, based on the destination MAC address of the packets themselves.

3.4.2 Forwarding graph and lowering process

The SG provides an high level formalism to define network services, but it is not adequate to be deployed on the physical infrastructure of the network because it does not include all the details that are needed by the service to operate. Hence, it must be translated into a more resource-oriented representation, namely the **forwarding graph (FG)**, through a **lowering process** that resembles to the intermediate steps implemented in software compilers. The FG can be seen as a generalization of the Openflow data model that specifies also the functions that have to process the traffic into the node, in addition to define the (virtual) ports the traffic has to be sent to.

The different steps of the lowering process are shown in Figure 3.5 and discussed in the following.

The **control and management network expansion** enriches the service with the "control and management network", which may be used to properly configure the VNFs of the graph. In fact, most NFs require a specific vNIC dedicated to control/management operations; although this may be an unnecessary detail for the player requiring the service, those network connections have to be present in order to allow the service to operate. In this step, the control network is created as a LAN and all the VNFs that actually have vNICs identified as *control interfaces* in their template (Section 3.4.4) are attached to it automatically. An example of this step is evident by a comparison between Figure 3.5(a) and Figure 3.5(b), in which a control/management network consisting of a L2 switch VNF has been added to the graph, although more complex forms for the control network (e.g., including also other VNFs such as a firewall) can be defined as well.

The **LAN expansion** translates the abstract LAN element defined in the SG into a proper (set of) VNFs that emulate the broadcast communication medium, e.g., a software bridge or an Openflow switch with an associated controller implementing the L2 learning mechanism. This step is shown in Figure 3.5(b) where the LAN is replaced with a software bridge.

The **service enrichment** requires that the graph is analyzed and enriched with those functions that have not been inserted in the SG, but that are required for the correct implementation of the service. An example is shown in Figure 3.5(c), where the graph analysis determines that the network segment connected to the user does not include any DHCP server, nor routing and NAT functionalities; in this case the proper set of VNFs is added automatically.

The **VNFs expansion** can replace a VNF with other equivalent VNFs, properly connected in a way to implement the required service. As an example, the firewall in Figure 3.5(b) is replaced in Figure 3.5(c) by a subgraph composed of a URL filter only operating on the web traffic, while the non-web traffic is delivered to a stateless firewall. As evident, the ports of the "original" VNF are now the endpoints of the new subgraph, which also has a control network dedicated to the new VNFs. Moreover, these new VNFs are in turn associated with a template, and can be recursively expanded in further subgraphs; this is equivalent to the "*recursive functional blocks*" concept provided in the ETSI standard [41], which may trigger further optimization passes.

The **VNFs consolidation** analyzes the FG looking for redundant functions, possibly optimizing the graph. For instance, Figure 3.5(d) shows an example in which two software bridges connected together are replaced with a single software bridge instance with the proper set of ports, hence limiting the resources required to implement the LANs on the physical infrastructure.

The **endpoints translation** converts the graph endpoints in either physical ports of the node on which the graph will be deployed, virtual ports (e.g., GRE tunnels) that connect to another graph running in a different physical server, or endpoints of another FG, if many graphs running on the same server must be connected together.

Finally, the **flow-rules definition** concludes the lowering process. In particular, (*i*) the connections among the VNFs, (*ii*) the traffic steering rules defined by the SG traffic splitter/merger, and (*iii*) the ingress matching rules associated with the endpoints, are translated into a sequence of "flow-space/action" pairs (Figure 3.6). The flow space includes all the fields defined by Openflow 1.3 [68] (although new fields can be defined), while the action can be a forwarding rule either to a physical or a virtual port.

As a final remark, the FG does not specify all low level details such as the physical node on which the service will be deployed, as well as the reference to the precise physical/virtual NICs needed by the VNF to operate, which are replaced by generic VNF entry/exit points, as shown in Figure 3.6. The final translation from abstract to actual VNF ports will be carried out in the next step.

3.4.3 Infrastructure graph and reconciliation process

The **infrastructure graph (IG)** is the final representation of the service to be deployed, which is semantically, but not syntactically, equivalent to the FG. The IG is obtained through the **reconciliation process**, which maps the FG on the resources available in the infrastructure layer, and it consists of the sequence of commands to be executed on the physical infrastructure in order to properly deploy and connect together all the required VNFs.

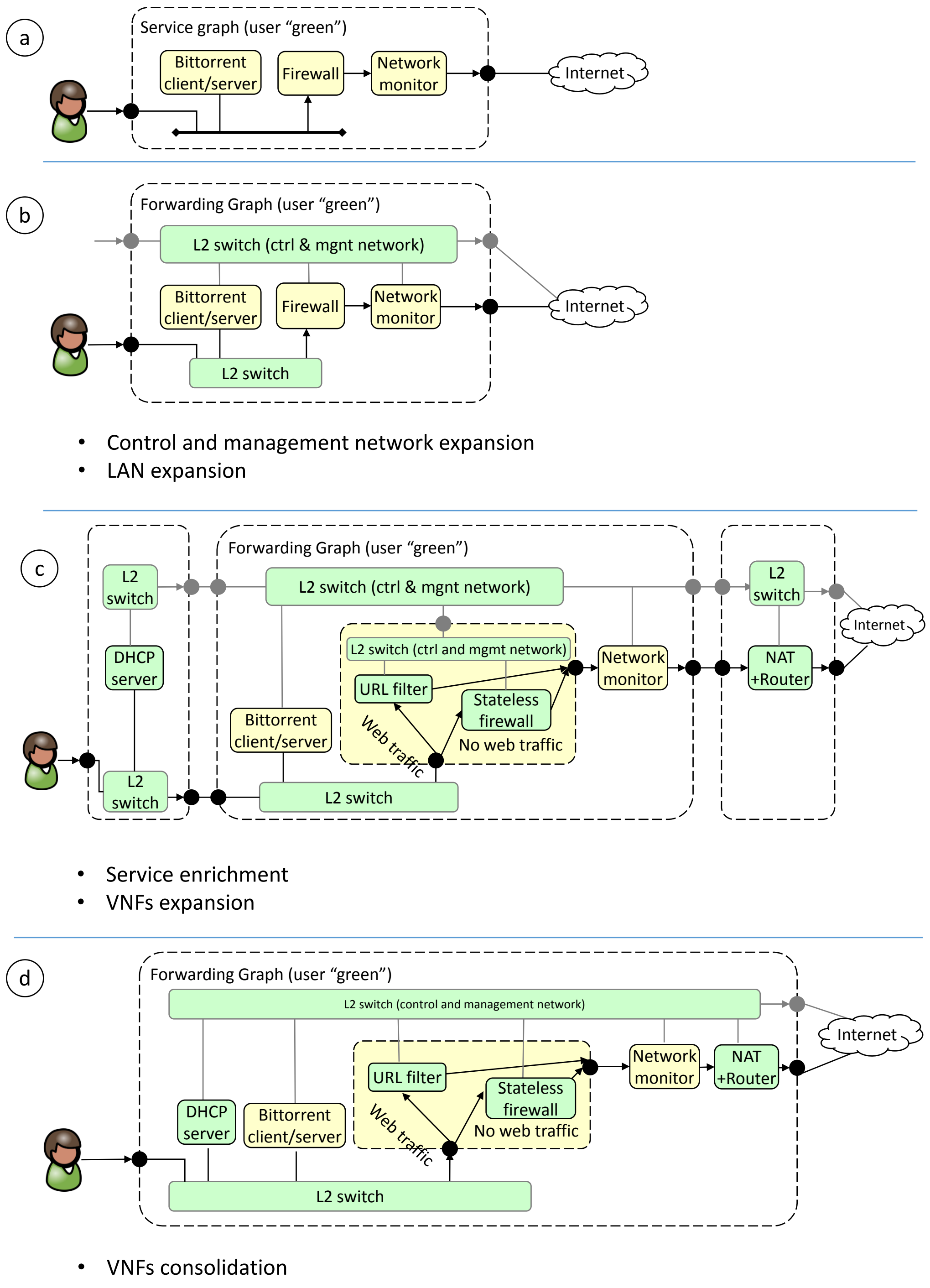


Figure 3.5: From the SG to the FG: the *lowering process*.

```
"forwarding-graph" : {
  "id" : "abcd123",
  "flow-rules" :  [
    {
      "flow-space" : {
        "port"  : "endpoint:1",
      },
      "action" : {
        "type" : "forward",
        "function" : "stateless_firewall:1"
      }
    },
    {
      "flow-space" : {
        "port" : "stateless_firewall:2",
        "tcp_src" : "80"
      },
      "action" : {
        "type" : "forward",
        "function" : "URLfilter:1"
      }
    },
    .......
  ]
}
```

Figure 3.6: Excerpt of a forwarding graph.

This process takes into account that some of the VNFs in the FG can be implemented through some modules (both software and hardware) already available on the node on which the graph is going to be deployed, instead of being implemented with the image specified in the template. For example, if the node is equipped with a virtual switch (vSwitch) that supports also the backward learning algorithm such as Open vSwitch [76], all the L2 switch VNFs in the FG are removed and those functions are carried out through the vSwitch itself, as shown in the right portion of Figure 3.7. Instead, as depicted in the left of Figure 3.7, if the node features a pure Openflow vSwitch (such as xDPd [26]), all the VNFs specified in the FG will be implemented by instantiating the proper image(s) (e.g., a VMs implementing the L2 bridging process). Obviously, other mappings between the VNFs and the resources available on the node are possible, according to the specific capabilities of the infrastructure layer; for instance we can obtain a different IG (hence a different number of deployed VNFs) starting from the same FG. The mapping of a VNF on a specific resource is possible thanks to the definition of a set of standard VNF names that uniquely identify a precise network function, such as the name "`L2 switch`" that identifies a LAN emulation component; this allows the reconciliation module to recognize the function needed, and possibly replace it with a more appropriate implementation.

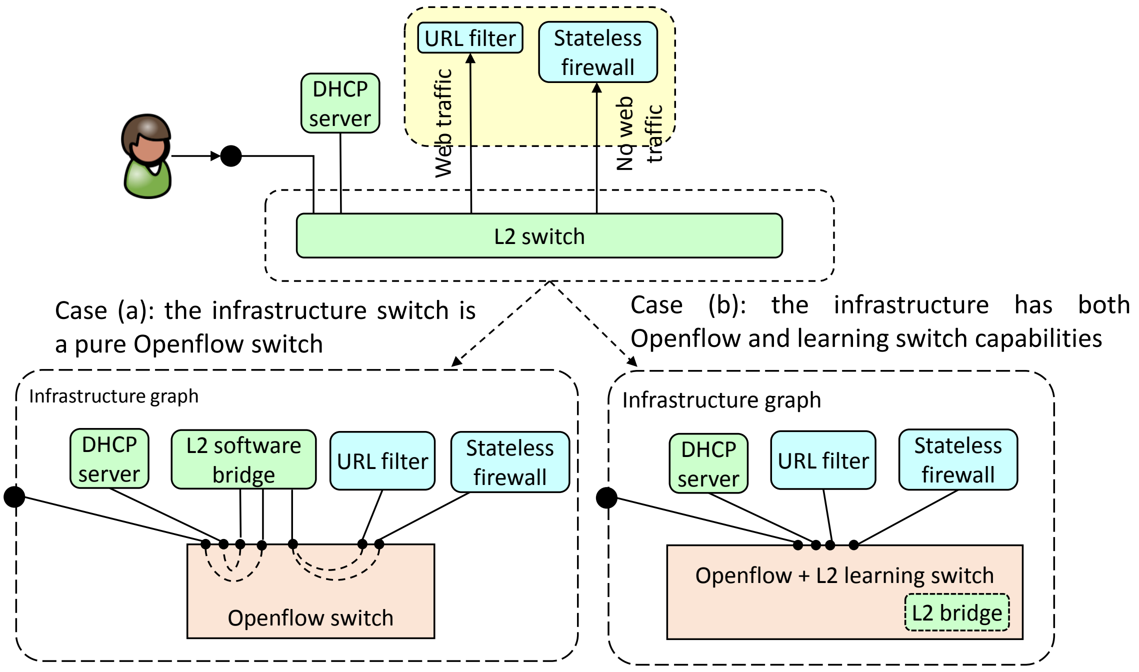


Figure 3.7: Example of the output of the reconciliation process when mapping a L2 switch functionality in case of two different types of infrastructure nodes.

After the reconciliation process, the final IG is converted into the commands (e.g., shell script, protocol messages) required to actually instantiate the graph, such as retrieve the VM image and start it up, create Openflow rules, and more. Particularly, the flow rules defining the links of the FG, i.e., the connections among the VNFs, are properly translated according to the technology used by the physical node to implement the graph. For example, if the physical node interconnects the VNFs through an Openflow vSwitch, each flow rule is converted in a number of Openflow `flowmod` messages, hence combining together SDN and NFV concepts. However, other flavors of the infrastructure layer could implement these connections through other technologies, such as GRE tunnels or VLAN tags. A similar process is applied to VNFs, as their images are retrieved and started using commands that depend on the technology implementing the VNFs (e.g., virtual machine, Docker container, etc.).

3.4.4 Network function template

Each VNF is associated with a template that describes the VNF itself in terms of both physical characteristics (e.g., CPU and memory requirements) and possible

```
"network-function" : {
  "name" : "firewall",
  "expandable": false,
  "uri": "http://myvnfs.com/images/7701f",
  "vnf-type" : "kvm-virtual-machine",
  "memory-requirements": 4096,
  "cpu-requirements": {
    "platform-type": "x86-64",
    "cores-number": 1
  },
  "ports": [
    {
      "label": "control",
      "cardinality" : "1",
      "ipv4-config": "DHCP"
    }
    {
      "label": "external",
      "cardinality": "1",
      "ipv4-config": "none"
    },
    {
      "label": "internal",
      "cardinality": "1-N",
      "ipv4-config": "none"
    }
  ]
}
```

Figure 3.8: Example of a VNF template.

infrastructure-level configurations (e.g., how many vNICs can be configured); an example of such a template is provided in Figure 3.8.

The template contains some information related to the hardware required to execute the VNF, such as the amount of memory and CPU as well as the CPU instruction set. Moreover, the boolean element `expandable` indicates if the VNF consists of a single image, or if it is actually a subgraph composed of several VNFs connected together. In the former case, the `uri` element refers to the image of the VNF, while in the latter it refers to a graph description that must replace the original VNF in the FG. In the case of *non-expandable* VNF, the template also specifies the image type such as KVM-compatible VM, Docker container, and more; for instance, the firewall described in Figure 3.8 is implemented as a single KVM-based virtual machine.

Moreover, the template provides a description of the ports of the VNF, each one associated with several parameters. In particular, the `label` specifies the purpose

Table 3.1: Challenges of the considered use case and related solutions.

	Challenge	Solution
#1	The SLApp recognizes when a new user is connected	API exported by the service layer to be notified of the occurrence of some events
#2	New user's authentication (the infrastructure layer does not implement any processing and forwarding logic by itself)	The SG formalism and the VNF template used to define a graph that includes VNFs implementing the user authentication
#3	Interconnection of several user SGs to a common telecom operator graph	Graph endpoints defined in the SG formalism
#4	A user SG only operates on the traffic belonging to that particular user	Graph endpoint associated with *ingress matching rules* identifying the traffic allowed to flow through the endpoint itself
#5	Each service is implemented on the physical infrastructure	The lowering process and the formalisms (SG, FG, IG) are generic enough to support all the possible services required by the users

of that port, and it is useful in the definition of the SG, since it helps to properly connect the VNF with the other components of the service (e.g., the *external* port of the firewall should be connected towards the Internet, while the *internal* ones should be connected towards the users). The label could assume any value, which is meaningful only in the context of the VNF. The parameter `ipv4-config`, instead, indicates if the port cannot be associated with an IPv4 address (`none`), or if it can be statically (`static`) or dynamically (`DHCP`) configured. Finally, `cardinality` specifies the number of ports of a certain type; for instance, the VNF of the example has one *control* port, one *external* port, and at least one *internal* port (in fact, it has a variable number of *internal* ports, which can be selected during the definition of the SG).

3.5 The validation use case: user-defined network services

Section 3.3 and Section 3.4 respectively provide a general overview of the architecture and a description of the associated data-models; those concepts could be used in different use cases involving multiple players in defining completely virtualized services.

In order to provide a concrete use case to validate our data models, we selected a challenging scenario in which *end users*, such as xDSL customers, can define their own service graphs to be deployed on the telecom operator infrastructure. Particularly, an end user's SG can only operate on the traffic of that particular end user, i.e., on the packets he sends/receives through his terminal device. Vice versa, the telecom operator can define a SG that includes some VNFs that should operate on all the packets flowing through the network; hence, this SG must be shared among all the end users connected to the telecom operator infrastructure.

Our use case presents some interesting challenges that can be solved through the multi-layer architecture and the data-models presented so far. These challenges, together with their solutions, are summarized in Table 3.1.

First, the service layer must be able to recognize when a new end user attaches to the network, and then to authenticate the user himself. The API exported by the service layer to the orchestration layer (Section 3.3.1) could be exploited in our use case just for this purpose.

Note that, since the infrastructure layer does not implement any (processing and forwarding) logic by itself, the authentication mechanism requires the deployment of a specific graph that only receives traffic belonging to unauthenticated users, and which includes some VNFs implementing the user authentication. This could be implemented by means of the SG formalism detailed in Section 3.4.1, together with the VNF template (Section 3.4.4).

Second, after the user is authenticated, the service layer must retrieve his SG and then connect it to the telecom operator graph in a way so that the user traffic, in addition of being processed by the service defined by the user himself, is also processed by the VNFs selected by the telecom operator. Notably, the telecom operator graph should be shared among different users, in order to reduce the amount of resources required by the service. The interconnection of two graphs in cascade can be realized by exploiting the graph endpoints elements provided by the SG formalism, as detailed in Section 3.4.1.

Third, the user SG must be completed with some rules to inject, in the graph itself, all the traffic coming from/going towards the end user terminal, so that the service defined by an end user (only) operates on the packet belonging to the user himself. Also this challenge has been solved thanks to the graph endpoints (Section 3.4.1), which can be associated with rules identifying the traffic that should enter into the graph through a particular endpoint.

Finally, the service layer must require (at the lower layers of the architecture) to deploy the user graph; this operation may require the creation of some tunnels on the network infrastructure so that the user traffic is brought from the network entry point to the graph entry point, which could have been deployed everywhere on the physical infrastructure. The multi-layer architecture proposed in Section 3.3 ensures the deployment of all the SGs defined at the service layer, regardless of the particular services described by the graphs themselves. In fact, both the lowering process that transforms the SG in FG (Section 3.4.2), as well as the instructions provided to the infrastructure components through the IG (Section 3.4.3), are generic enough and can model all the possible services defined at the service layer.

3.6 Prototype implementation

This section presents the preliminary implementation of the architecture introduced in Section 3.3, detailing its components and the engineering choices that have been made in order to create the prototype.

3.6.1 The service layer

Our service layer logic is strictly related to our use case, in which the end users can define generic services to be applied to their own traffic, while the operator can define a service operating on all the packets flowing through the network. Our implementation of the SLApp delegates specific tasks to different OpenStack modules, some of which have been properly extended. In particular, **Horizon**, the OpenStack dashboard, is now able to provide to the end users a graphical interface allowing them to express (out of band) the service they expect from the network, using the building blocks depicted in Figure 3.3. **Keystone**, the token-based authentication mechanism for users and permissions management, is now able to store the user profile, which contains user's specific information such as the description of his own SG. Finally, **Swift**, the OpenStack object storage, has been used to store the VNF templates. Notably, these modules are present also in case of the Universal Node implementation (Section 3.6.3), as the service layer is independent from the actual infrastructure layer.

At boot time, the SLApp asks the orchestration layer to instantiate two graphs, the telecom operator graph and the authentication graphs (Section 3.6.1), which are deployed on one of the available infrastructure nodes. In addition, it configures the network nodes with the proper rules to detect when a new flow is created on the network (e.g., a new user terminal attaches to an edge node), which in turns triggers a call of the proper event handler in the SLAapp that forces an update of the authentication graph, so that it can properly handle the traffic generated from the new connected device. In fact, the SLApp has been notified about the source MAC address of the new packets, which can be used to uniquely identify all the traffic belonging to the new device. This enables the SLApp to update the authentication graph with a new endpoint representing the entry point of the user traffic in the network, and which is associated with an ingress matching rule expressed on the specific MAC address; this way, the new packets can be provided to the authentication graph, wherever the graph itself has been deployed. Finally, the updated graph is passed to the orchestration layer, which takes care of applying the modifications on the physical infrastructure.

A successful user authentication through the authentication graph triggers the instantiation of the SG associated with the user himself. This is achieved by the SLApp, which retrieves the proper SG description from the user profile repository, connects it to the telecom operator-defined SG such as in the example shown in the bottom right part of Figure 3.4, and starts the lowering process aimed at creating the FG. In particular, the SLApp executes the "*control and management network expansion*" and the "*LAN expansion*" as shown in Figure 3.5(b), and the "*service enrichment*" step. In our case, the latter consists in adding a DHCP server and a VNF implementing the NAT and router functionalities, in case these VNFs have not been included by the end user during the definition of his own service.

Before being provided to the orchestration layer, the user-side endpoint of the user SG is associated with *(i)* an ingress matching rule expressed on the MAC address of the user device, so that only the packets belonging to that user are delivered to his graph, and *(ii)* the entry point of such a traffic in the telecom operator network (i.e., the actual port on the physical edge node where the user connects to). This way, the orchestration layer will be able to configure the network in order to bring the user traffic from its entry point into the network to the node on which the graph is deployed.

Finally, the SLApp also keeps track of user sessions in order to allow a user to connect multiple concurrent devices to his own SG. In particular, when a user already logged in attaches to the network with a new device, the SLApp: *(i)* retrieves the FG already created from the orchestration layer; *(ii)* extends it by adding a new endpoint associated with an ingress matching rule operating on all the traffic coming from the new device, and representing the entry point of such packets in the network; *(iii)* sends the new FG to the orchestration layer.

Authentication graph

The *authentication graph* is used to authenticate an end user when he connects to the telecom operator network with a new device, and it is automatically deployed by the service layer when the infrastructure starts, hence turning a service-agnostic infrastructure into a network that implements our use case.

The authentication SG (shown at the top of Figure 3.9) consists of a LAN connected to a VNF that takes care of authenticating the users, a DNS server and a DHCP server. However, the VNF implementing the users authentication is in fact another SG made up of three VNFs: an Openflow switch, an Openflow controller and a web captive portal. The resulting FG, completed with the control and management network, is shown at the bottom of Figure 3.9; as evident, only the control network is connected to the Internet while all the user traffic is kept local.

When an unauthenticated user connects to the network, his traffic is brought to the authentication graph. In particular, the DHCP server returns an initial IP address to the user, which is able to generate traffic. All DNS queries are resolved by the DNS server into the proper IP addresses, but all the web requests are redirected to the captive portal; in fact, the HTTP traffic entering into the Openflow switch is sent to the Openflow controller, which modifies the original MAC and IP destination addresses with those of the web captive portal and then sends back the packet to the Openflow switch, which will deliver it to the captive portal. This VNF provides a HTTP `302 temporary redirect` message to the user in order to notify the client of the redirection and avoiding wrong caching, then a login page is shown. After the (successful) user authentication, the web captive portal contacts the SLApp through the control network and triggers the deployment of the SG associated with that user on the infrastructure.

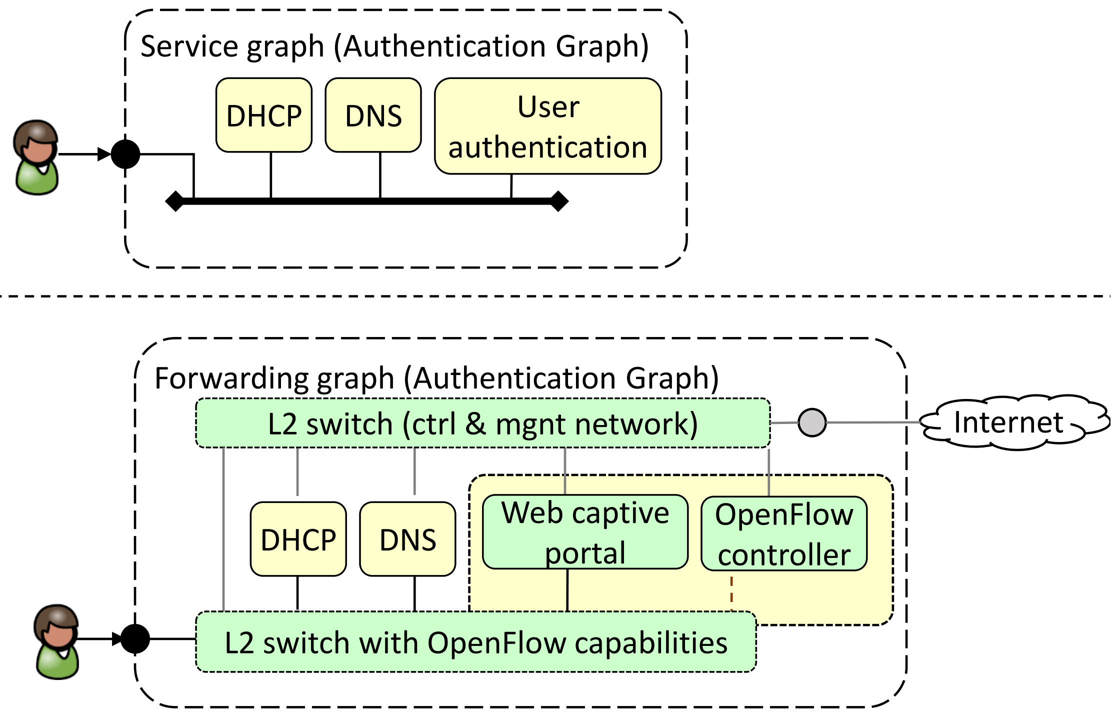

Service graph (Authentication Graph)
DHCP
DNS
User authentication
Forwarding graph (Authentication Graph)
L2 switch (ctrl & mgnt network)
Internet
DHCP
DNS
Web captive portal
OpenFlow controller
L2 switch with OpenFlow capabilities

Figure 3.9: Authentication SG and FG.

3.6.2 Global orchestrator

The **global orchestrator** implements the first two levels of the orchestration layer depicted in Figure 3.2, and consists of a technology-dependent and a technology-independent part.

The technology-independent part receives the FG from the service layer (through its northbound interface) and it executes the following operations as defined in the lowering process (Section 3.4.2). First, for each VNF specified in the graph, it retrieves the corresponding VNF template from the OpenStack Swift service. In case the template is actually a subgraph composed by other VNFs (Section 3.4.4), it executes the "*VNFs expansion*" step (Figure 3.5(c)) and retrieves the description of the new VNFs that, in turn, could be recursively expanded in further subgraphs. The "*VNFs consolidation*" step follows, possibly consolidating multiple function instances as shown in Figure 3.5(d). Finally, the "*flow-rules definition*" step creates a sequence of "flow-space/action" pairs describing how to steer the traffic within the graph.

At this point, the global orchestrator schedules the FG on the proper node(s) of the physical infrastructure. Although the general model presented in Section 3.3.2 supports a scheduling based on parameters such as CPU and memory requirements of the VNFs, KQIs (e.g., maximum latency, expected throughput) and high level policies, the current implementation simply instantiates the entire FG on the same node used as a network entry point for the traffic to be injected into the graph itself. The resulting FG is then provided to the proper control adapter, chosen based on the type of infrastructure node that has been selected by the scheduler and that has to execute the FG (i.e., OpenStack-based Node or Universal Node). These adapters take care of translating the FG into the formalism accepted by the proper *infrastructure controller*, which is in charge of sending the commands to the infrastructure layer. Moreover, they convert the abstract endpoints of the graph (i.e., the ones that have not yet been translated into physical ports e.g., by the service layer) into physical ports of the node; finally, if needed, they instruct the infrastructure controller to create the required GRE tunnels. Tunnels can be used to connect together graphs that have been instantiated on two different nodes (*graph cascading*), or to connect two portions of the same graph that have been deployed on different nodes (*graph splitting*). For instance, in our use case a tunnel is required to bring the traffic generated by unknown user terminals to the authentication graph.

As a final remark, the global orchestrator supports the update of existing graphs. In fact, when it receives a FG from the service layer, it checks if this graph has already been deployed; in this case, both FGs (the one deployed and the new one) are provided to the proper control adapter, which sends to the infrastructure controller either the difference between the two graphs in case of Universal Node, or both FGs in case of OpenStack-based Node, as that implementation will be able to automatically identify the differences thanks to the OpenStack Heat module.

3.6.3 The Universal Node

The **Universal Node** [92] is the first flavor of our infrastructure layer and it consists of a single physical machine running mostly ad hoc software, whose overall architecture is shown in Figure 3.10. In this implementation, the infrastructure controller is *integrated* on the same server running the VNFs. Multiple Universal Nodes are possible on the infrastructure and must be coordinated by the global orchestrator.

The Universal Node receives the FG through the northbound (REST) interface of the **node resource manager**, which is the component that takes care of instantiating the graph on the node itself; this requires to execute the reconciliation process in order to obtain the IG, to start the proper VNF images (downloaded from a **VNFs repository**) and to configure the proper traffic steering among the VNFs. Particularly, the last two operations are executed through two specific modules of the node resource manager, namely the **compute controller** and the **network controller**.

Traffic steering is implemented with a (pure) Openflow DPDK-enabled datapath, based on the **extensible Data-Path deamon (xDPd)** [26]. xDPd supports the dynamic creation of several Openflow switches, called Logical Switch Instances

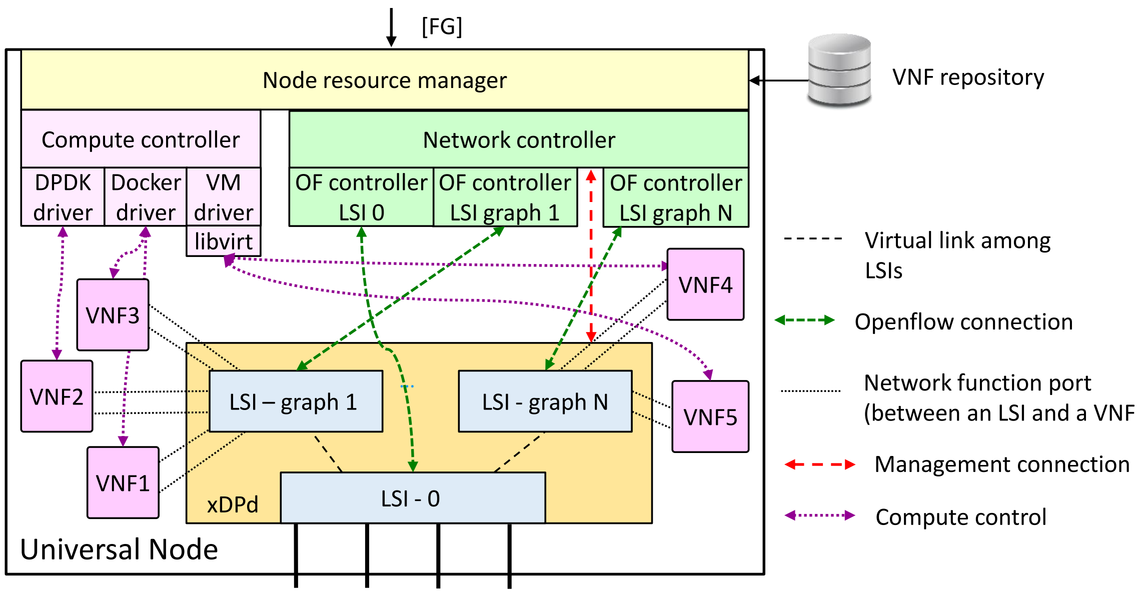


Figure 3.10: Logical architecture of the Universal Node.

(LSIs); each LSI can be connected to physical interfaces of the node, to VNFs, and to other LSIs. A different LSI (called `graph-LSI`) is dedicated to steer the traffic among the VNFs of a specific graph, while the `LSI-0` is in charge of classifying the traffic coming from the network (or from other graphs) and of delivering it to the proper `graph-LSI`. The `LSI-0` is in fact the only one allowed to access the physical interfaces, and the traffic flowing from one `graph-LSI` to another has to transit through the `LSI-0` as well. Since LSIs are *pure* Openflow switches, the reconciliation process described in Section 3.4.3 cannot remove the L2Switch VNFs, which are then implemented using the proper software images because of the unavailability of the backward learning algorithm in xDPd.

When a FG description (either a new one or an update of an existing FG) is received by the node resource manager, this module: *(i)* retrieves a software image for each VNF required and installs it; *(ii)* instantiates a `graph-LSI` on xDPd and connects it to the `LSI-0` and to the proper VNFs; *(iii)* creates a new Openflow controller associated with the graph-LSI, which is in charge of inserting the required forwarding rules (i.e., traffic steering) in the `graph-LSI` itself. In other words, this controller sets up the proper connections among VNFs as required by the FG. In particular, the FG rules that define the paths among VNFs (and physical ports) originate two sequences of Openflow `flowmod` messages: one to be sent to the `LSI-0`, so that it knows how to steer traffic among the graphs deployed on the node and the physical ports; the other used to drive the `graph-LSI`, so that it can properly steer the packets among the VNFs of a specific graph.

When a packet enters into the `LSI-0` and cannot be forwarded to any `graph-LSI`, it is delivered to the `LSI-0` controller using the Openflow `packet in` message; at this point the Openflow controller notifies the service layer of the presence of a new user terminal, which will react by creating the proper network setup (e.g., tunnels) to redirect that traffic to the authentication graph.

The Universal Node supports three flavors of VNFs: DPDK processes [56], Docker containers [3], and VMs (executed in the KVM [6] hypervisor). While the former type provides better performance (in fact, an LSI exchanges packets with DPDK VNFs with a zero-copy mechanism), Docker containers and VMs guarantee better isolation among VNFs, as well as they allow to limit CPU and memory usage. Data exchange between LSIs and Docker containers/VMs takes place through the `KNI` virtual interface available with the DPDK framework.

Finally, the architecture of the Universal Node can support a network-aware scheduling algorithm, which has the potential to optimize the location of each VNF based on the I/O connections of the VNF itself. This allows for example to start two cascaded VNFs on two cores of the same physical CPU in order to keep the traffic within the same NUMA node, or to allocate a VNF on the CPU that is connected to the NIC used to send the packet out to the network. This is possible because the node resource manager, which takes care of both deploying the VNFs and configuring the vSwitch to properly steer the traffic among them, receives the entire FG from the upper layer, which describes both the VNFs to be executed and the connections among them.

3.6.4 The OpenStack-based node

The **OpenStack-based Node** is the second flavor of our infrastructure layer and it consists of a cluster of servers within the same OpenStack domain. As shown in Figure 3.11, all the physical machines of the cluster are managed by a single infrastructure controller, which is composed of a number of OpenStack modules and an SDN controller. Multiple OpenStack-based Nodes are possible on the infrastructure and must be coordinated by the global orchestrator.

OpenStack [13] is a widespread cloud toolkit used for managing cloud resources (network, storage, compute) in data-centers; hence, its support in our architecture represents an interesting choice because of the possibility to deploy our services in an existing (and widely deployed) environment. However, since OpenStack was designed to support the deployment of *cloud* services, several modifications have been made to support FGs (hence *network* services) as well.

As depicted in Figure 3.11, our OpenStack-based Node exploits the following components: *(i)* **Nova**, the compute service; *(ii)* **Neutron**, the network service; *(iii)* **Heat**, the orchestration layer and *(iv)* **Glance**, the VM images repository. Openstack is able to start VMs by interacting with a wide range of different hypervisors (e.g. KVM, Xen [23], VMware); moreover, in order to properly steer

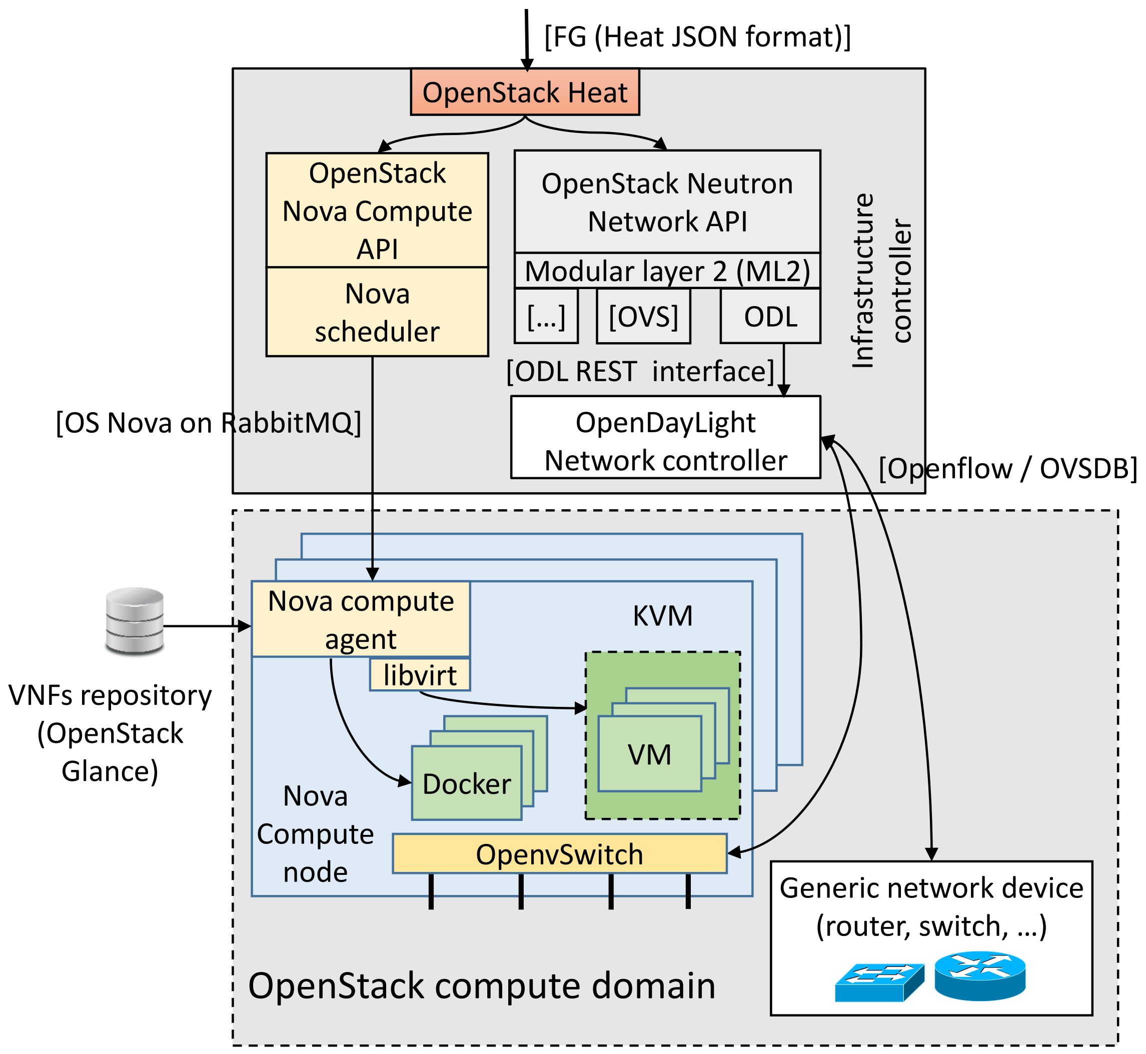


Figure 3.11: OpenStack-based Node.

the traffic between the several servers under its control our prototype integrates also the **OpenDaylight (ODL)** [11] SDN controller. As evident from the picture, Heat, Nova scheduler, Nova API, Neutron and ODL compose the infrastructure controller, while each physical machine executing the VNFs is a Nova compute node, which runs a Nova compute agent, the **Open vSwitch** (OvS) [76] softswitch and the **KVM hypervisor**.

When the global orchestrator decides to deploy a FG in an OpenStack-based Node, the proper control adapter translates the FG description into the format supported by Heat. To be used in our prototype, Heat has been extended in order to support the `flow-rule` primitive, which describes how to steer the traffic between the ports of the VNFs composing a graph. This primitive provides an interface similar to the OpenFlow 1.3 `flowmod`; however, it allows the traffic steering between virtual ports without knowing in advance the physical server on which the respective VNFs will be scheduled. As soon as Heat receives the FG, it performs a reconciliation step that removes, from the graph itself, all the VNFs implementing the L2 switch, since this functionality will be mapped on the OVS instances running on the physical servers. In fact, OVS is able both to forward traffic based on traffic steering rules, as well as to execute the MAC learning algorithm. After this translation, the FG is decomposed into a set of calls to Nova and Neutron.

For the compute part, Nova receives a sequence of commands for each VNF of the graph in order to deploy and start the VNF itself; at this point, the Nova scheduler *(i)* selects the physical server on which the VNF must be deployed using the standard OpenStack "filter & weight" algorithm[3], *(ii)* sends the proper command to the Nova compute instance on the selected node, which in turn *(iii)* retrieves the VNF image from Glance and finally *(iv)* starts the VM[4]. It is worth noting that Nova scheduler has two limitations: *(i)* it schedules a VNF as soon as it receives the command from Heat; *(ii)* it does not have any information on the paths among the VNFs in the graph. As a consequence, the FG could be split on the available compute nodes without taking into account the paths among the VNFs, clearly resulting in suboptimal performance.

For the networking part, when Heat detects that all the VNFs (i.e., VMs) are started, it sends a `flow-rule` at a time to Neutron, which takes care of creating the proper connections among these VNFs. Similarly to Heat, also Neutron has been extended to support the `flow-rule` primitive[5]. When Neutron receives a `flow-rule`, it retrieves the network topology (i.e., the interconnections among the servers forming the OpenStack domain) from ODL and then creates the proper Openflow `flowmod` messages required to steer the traffic on the physical infrastructure. At this point, the `flowmods` are provided to ODL, which sends them to the proper switches; note that these switches could be either inside a Nova compute node, or physical switches used to interconnect several servers, in case the VNFs have been instantiated by the Nova scheduler on many compute nodes.

In addition to the components described so far, each OpenStack deployment also

[3] This algorithm acts as follows: first, all the Nova compute nodes that are not able to run a VM are filtered (e.g., because the VM requires an hypervisor that is not available on the node). Then, a weight is associated with each one of the remaining servers, and the one with higher weight is selected to run the VM. The weights are calculated by considering the resources available on the machine.

[4]Note that no modification has been required by Nova compute in order to support the deployment of the FGs.

[5] The `flow-rule` is functional equivalent to the Neutron official traffic steering extension [21]. However, it has not been used in our prototype because: *(i)* it was not available when this prototype was created (July 2014); *(ii)* it does not support ports that do not belong to a VM.

Table 3.2: Universal Node vs OpenStack-based Node.

	Universal Node	**OpenStack-based Node**
Compatible with existing cloud environments	No	Yes
Complete control of the FG	Yes	No (due to the network node)
Support to smart scheduling of the FG	Possible	Requires many changes to the OpenStack internals
Type of VNFs	Docker containers, DPDK processes, VMs	VMs, Docker containers (not completely supported)

includes the **network node**, which is a particular server running some services such as a NAT, a DHCP server, a load balancer and a router; moreover, by default it is crossed by all the traffic entering/leaving the cluster of servers.

Similarly to the Universal Node, also the OpenStack-based Node notifies the service layer when a new user terminal connects to the node itself in order to allow the system to redirect that traffic to the authentication graph.

3.6.5 Discussion: Openstack-based Node vs. Universal Node

Table 3.2 summarizes the main differences between the Universal Node and the OpenStack-based Node. As shown, the main advantage of the latter is its capability to deploy SGs in an existing cloud environment (albeit with some modifications), which facilitates the introduction of those services in telecom operator networks; in fact, operators often already have OpenStack instances active in their data centers. However, this very important advantage is balanced by severe limitations compared to the Universal Node.

First, OpenStack does not allow the service layer to have the complete control of the service chain, as each OpenStack domain is connected to the Internet through the *network node* (Section 3.6.4) and all the packets towards/from the Internet are forced to traverse the network services running in this component (e.g., NAT and router), even if the SG does not include those functions.

Second, the OpenStack-based Node cannot optimize the placement of the VNFs based on their layout in the FG, e.g., possibly instantiating two consecutive VNFs within the same graph on the same physical server. This is due to the fact that the OpenStack scheduler, implemented in Nova, is unaware of the overall layout of the FG: this information is only received by Heat and it is not passed down to the Nova scheduler. To make things worse, the Nova scheduler is invoked individually per each VNF that has to be scheduled, and therefore it is unable to implement even a simple optimization such as scheduling all the VNFs of the same FG on the same server. This problem is not present in the Universal Node, as it receives the entire FG at the same time, hence it has all the information related to the required VNFs and the paths among them. Then, the Universal Node can potentially schedule the VNFs on the available CPUs by considering their position in the graph, with the obvious advantages in terms of overall performance.

Third, the Universal Node shows better memory consumption compared to the OpenStack-based Node, whose components have not been created for resource-constrained environments as this is unlikely to occur in (almost resource-unlimited) data centers.

Finally, the OpenStack-based Node supports only VM-based VNFs, as the initial support for Docker containers appears still rather primitive (in July 2014). Vice versa, the Universal Node supports VNFs implemented as Docker containers, VMs and DPDK processes, which seems to suggest the possibility to write more lightweight and efficient VNFs, particularly with respect to the potentially better I/O capabilities, which represent a fundamental difference from VNF and traditional VM-based services.

3.7 Prototype validation

To validate the architecture described in this chapter, we carried out several tests aimed at both testing the functionalities implemented, and to measure the performance of the infrastructure layer in terms of throughput, latency introduced and resource required. The tests were repeated both with an infrastructure layer consisting of a single Universal Node, as well as in case of an OpenStack cluster composed of two compute nodes. Note that, in the latter case the graphs are split so that the VNFs are distributed between the two physical servers.

3.7.1 Service overview

The FGs deployed in the tests are shown in Figure 3.12; according to our use case, these graphs include the authentication graph used to authenticate new end users connected to the network, and the telecom operator graph, which provides connectivity to the Internet and that is crossed by the traffic generated from/going towards all the end users. The control network of this telecom operator graph also includes a firewall, so that only the authorized entities (e.g., the telecom operator itself) can control and configure the deployed VNFs.

The end user graph provides an example of traffic steering, since it requires that the web traffic is delivered to a traffic monitor and then to a firewall that blocks the HTTP GET towards specific URLs, while the other packets simply traverse a second traffic monitor VNF. Thanks to the control interface of the traffic monitors we are able to observe the packets flowing through the specific VNF, and hence to validate the correct behavior of the traffic steering mechanism.

During the tests carried out on the OpenStack-based Node, the VNFs are implemented as VMs running on the KVM hypervisor. In contrast, in the tests with the

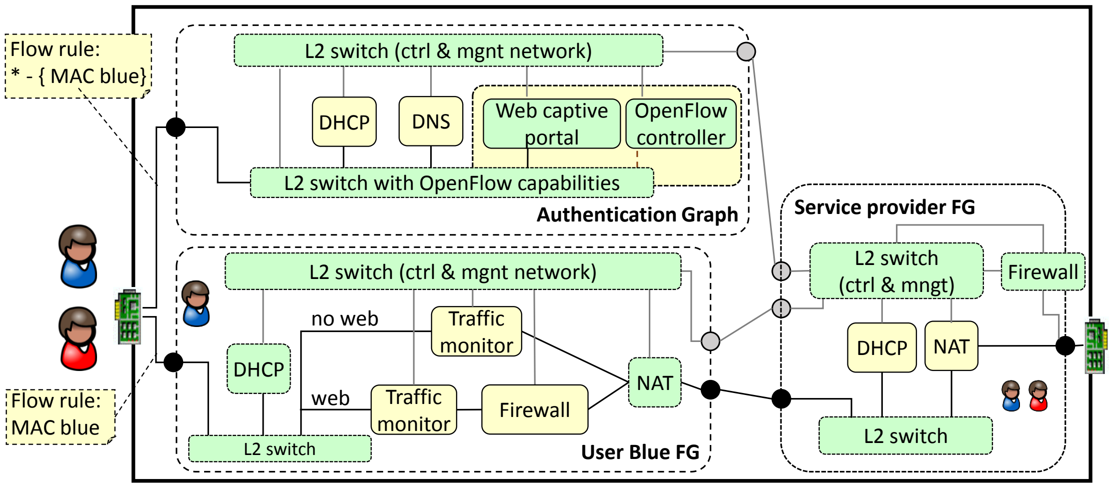


Figure 3.12: Use case scenario.

Universal Node, the firewall is implemented as a DPDK process[6], while all the others VNFs are implemented as Docker containers. In particular, both the VMs and the Docker containers run an Ubuntu operating system, and the VNFs are implemented through *standard* Linux tools (e.g., `iptables`).

As a final remark, according to our use case and the current implementation of the architecture, the end users are directly connected to the node on which their graphs are deployed.

3.7.2 Performance evaluation

This section shows the tests executed in order to measure the performance of the preliminary implementation of our architecture.

During the tests, a machine is dedicated to the execution of the service layer (i.e., SLApp, Keystone and Horizon) and the global orchestrator; it is equipped with 16 GB RAM, 500GB HD, Intel i7-2620M @ 2.7 GHz (one core plus hyperthreading) and OSX 10.9.5, Darwin Kernel Version 13.4.0, 64 bit, which is the same for both the infrastructure nodes.

The infrastructure layer is implemented on a set of servers with 32 GB RAM, 500GB HD, Intel i7-3770 @ 3.40 GHz CPU (four cores plus hyperthreading) and Ubuntu 12.04 server OS, kernel 3.11.0-26-generic, 64 bits. In case of the Universal Node, one of those machines executes all the software. In case of OpenStack-based Node, a first machine hosts the infrastructure controller (Heat, Nova scheduler,

[6]This firewall is a quite simple single thread process based on the `libpcre` regular expression engine, which drops all the packets matching specific regular expressions.

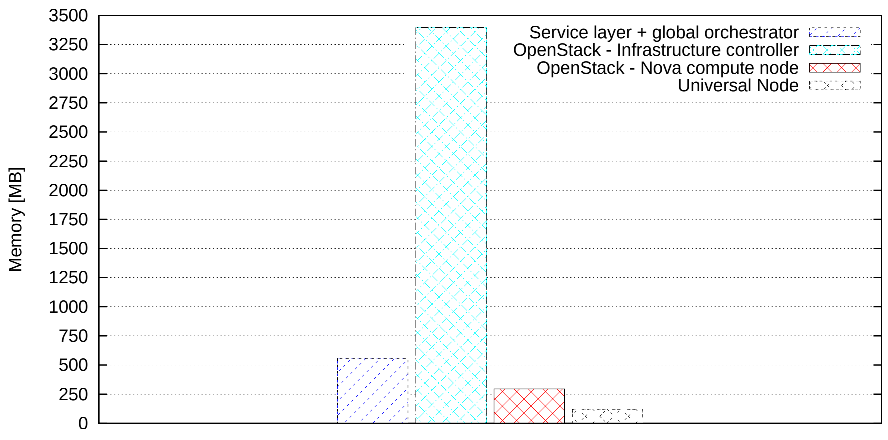


Figure 3.13: Memory consumption.

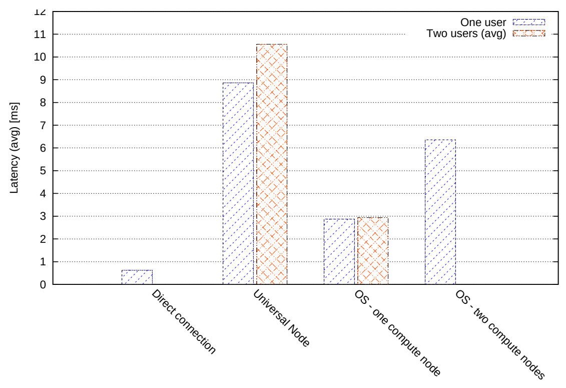
Nova API, Neutron and ODL) and the network node, while two other machines are dedicated to the implementation of two Nova compute nodes.

The memory required by the different components of the system is reported in Figure 3.13, in which the consumption related to the Nova compute node and to the Universal Node has been measured without any VNF deployed. As shown, the infrastructure controller for the OpenStack-based Node is the heaviest component, while the requirements of the Universal Node, which is almost based on ad hoc modules, is quite reduced.

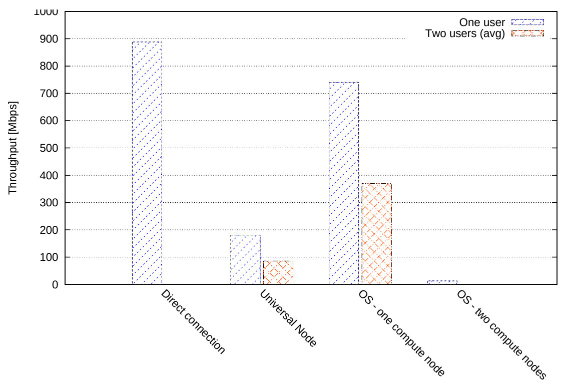
According to Figure 3.14, we repeated the performance tests in the following conditions: *(i)* user device and server directly connected using a gigabit Ethernet link; *(ii)* user devices connected, through a gigabit Ethernet network, to the node on which the graphs are deployed, which is in turn connected to the server through a second gigabit Ethernet link. Moreover, as node running the VNFs, we used: the Universal Node, an OpenStack-based Node with a single server, and an OpenStack-based Node consisting of two servers connected with a gigabit Ethernet link.

The first test carried out aims at measuring the latency introduced by the deployed services (Figure 3.12); in particular, the user device(s) sends 100 `ping` towards the server, and the results were averaged and reported in Figure 3.14(a). Figure 3.14(b) shows instead the results of the second test executed, aimed at measuring the throughput obtained during the download of a file of 512 MB from the server; the download has been done using the Linux tool `wget`, which uses the HTTP protocol. Hence, according to the user graph, while the ping is not handled by the firewall, this VNF is instead involved during the file transfer.

As expected, the deployment of a SG on the network does not come for free, since the numbers obtained are reduced with respect to the case in which no service is instantiated between the user device and the server. However, as evident from the figure, this penalty is limited when the graph is deployed on an OpenStack-based



(a)



(b)

Figure 3.14: Performance of the infrastructure layer: (a) ping; (b) file transfer.

Node consisting of a single server, both in case a single user graph is instantiated (in addition to the authentication and the telecom operator graphs) and in case of two (identical) user graphs. In this last condition, the test has been executed with both the users pinging/transferring the file at the same time, and the results have been averaged in the graphs.

Instead, when the graphs are scheduled in an OpenStack cluster of two nodes, performance are worse in both the types of test; for this reason, the measurements have not been repeated with two users connected to the node. The low performance are a consequence of the fact that the standard scheduling algorithm implemented in OpenStack scheduled the user VNFs in a way so that each packet crosses four times the link between the two compute nodes. This confirms the necessity of the introduction, in the Nova scheduler, of an algorithm that schedules the VNFs on the physical servers according to their interconnections in the graph[7].

Surprisingly, results obtained with the Universal Node are extremely low, unless the entire graph is deployed on a single server. We are currently investigating the reason for this poor performance, although we suspect they are related to the packet exchange mechanism between the vSwitch and the Docker containers (currently based on the DPDK KNI ports), which should be carefully optimized.

[7]Note that the same algorithm should also be implemented in the global orchestrator, which schedules the VNFs on the proper nodes of the infrastructure layer.

3.8 Conclusion and future works

This chapter presents a network orchestration architecture that, starting from the service required by multiple players (e.g., end users, telecom operator), takes care of instantiating it on the physical infrastructure of the network, by exploiting the opportunities offered by the Network Functions Virtualization (NFV) and Software Defined Networking (SDN) paradigms.

The contribution of this chapter is twofold. First, we propose a new formalism, called *service graph (SG)*, to flexibly model end-to-end network services. The SG data-model describes how to deliver flexible network services, leveraging existing elements and the traffic steering primitives introduced by NFV/SFC. It is worth noting that this SG definition is completely compliant with NFV principles of abstract description of a service, but enriches its traditional expressiveness to model legacy networks and services.

The second contribution is made by the introduction of the *forwarding graph (FG)* and the "lowering process" that leads to the deployment of an optimized service. This translation process is capable to adapt the service delivering to available resources of the underlying infrastructure; moreover, it is also able to detect specific capabilities of selected nodes adapting the infrastructure graph obtained as output.

In order to validate our model, we implemented two prototypes of nodes for the physical infrastructure: the *Universal Node* and the *OpenStack-based Node*. While the former consists of a single server mainly based on ad hoc components, the latter is implemented as a cluster of servers orchestrated by the an extended version of the OpenStack framework. Experimental results showed that, while the Universal Node has lower requirements in terms of memory, its performance are overcome by the OpenStack-based Node in almost all the tests carried out.

It is worth pointing out that the modifications proposed to the "vanilla" OpenStack were designed by avoiding to change existing API or disrupt former primitive behavior provided by the platform. Therefore, those add-ons can be silently integrated in a previous installation, transparently enriching the network capabilities of an OpenStack domain.

As a plan for the future, we foresee two different challenges to be pursued in order to let this architecture to properly scale to the telecom operator network size. First, the proposal of an algorithm to implement a network-aware scheduling, capable of deploying VNFs on the physical infrastructure by considering the paths expressed into the graph.

Second, the definition of a hierarchical orchestration layer through the whole telecom operator network. This would allow the deployment of a FG across multiple administrative domains, in which the lower level orchestrators expose only some information to the upper level counterparts. This scenario is perfectly compatible with our architecture and will be the object of further analysis; in fact, the global orchestrator presented in this chapter has syntactically identical northbound and southbound interfaces, and hence a hierarchy of orchestrators is possible.

As a final remark, the configuration parameters for the network functions, as well as the possibility of assessing formal properties on them, are out of the scope of this chapter and will be investigated in our future work.

Part II

Optimizing packets movement between Virtual Network Functions

Overview

As shown in Part I of this dissertation, Network Function Virtualization (NFV) [43] is a recent network paradigm with the goal of transforming in software images, those network functions that for decades have been implemented in proprietary hardware and/or dedicated appliances, such as NAT, firewall, and so on. These software implementations of network functions, called Virtual Network Functions (VNFs) in the NFV terminology, can be executed on high-volume standard servers, such as Intel-based blades. Moreover, many VNFs can be consolidated together on the same server, with a consequent reduction of both fixed (CAPEX) and operational (OPEX) costs for network operators.

Recently, the European Telecommunication Standard Institute (ETSI) started the Industry Specification Group for NFV [41], with the aim of standardizing the components of the NFV architecture. Instead, the IETF Service Function Chain [57] working group takes into account the creation of paths among VNFs; in particular, they introduce the concept of Service Function Chain (SFC), defined as the sequence of VNFs processing the same traffic in order to implement a specific service (e.g., a comprehensive security suite). Hence, a packet entering in a server executing VNFs may need to be processed in several VNFs (of the same chain) before leaving such a server. Moreover, several chains are allowed on a single server, which process different packet flows in parallel.

According to [28]: "NFV is heavily based on cloud computing technologies; in fact, VNFs are typically executed inside Virtual Machines (VMs) or in more lightweight virtualized environments (e.g., Linux containers [8]), while the paths among VNFs deployed on a server are created through virtual switches (vSwitches). However, cloud computing and NFV differ both in the amount and in the type of traffic that has to be handled by applications and vSwitches. This difference is due to the following reasons. First, traditional virtualization has to deal most with compute-bounded tasks, while network I/O is the dominant factor in NFV (the main operation of a VNF is in fact to process passing traffic). Second, a packet may need to be handled by several VNFs before leaving the server; this adds further load to the vSwitch, which has to process the same packet multiple times. Finally, common techniques to improve network I/O such as Generic Receive Offload (GRO) and TCP Segmentation Offload (TSO) may not be appropriate for NFV, since some VNFs (e.g., L2 bridge, NAT) need to work on each single Ethernet frame, and not on TCP/UDP segments."

Then, this part of the dissertation explores new mechanisms to interconnect VNFs chained on the same physical server, which are different with respect to those used in traditional cloud computing environments and that take into account the differences (in terms of network traffic) between NFV and the cloud computing world.

More in detail, both Chapter 4 and Chapter 5 focus on solutions to improve the efficiency of the packet exchange between the virtual switch and the VNFs, especially when a massive number of (tiny) VNF instances are executed on the same server, as in the case in which each end user is enabled to deploy his own VNFs. Both the chapters provide an extensive performance evaluation of the proposed mechanisms in order to prove their goodness in the NFV scenario; in addition, Chapter 5 validates the presented algorithm also through a formal verification of its main safety and security properties.

Chapter 6 proposes instead an architecture that transparently optimizes the data transfer between virtual machines, by letting the virtual switch forwarding plane out of the picture in case the service to be implemented requires that all the traffic coming from a first VNF has to be processed into a second one. The prototype demonstrates the huge advantages of this architecture and the possibility to implement it with localized modifications, mainly inside the virtual switch.

Chapter 4

Supporting fine-grained virtual network functions through Intel DPDK

4.1 Introduction

Network Functions Virtualization (NFV) proposes to implement in software the many network functions (e.g., NAT, firewall, etc.) that today run on proprietary hardware or dedicated appliances, by exploiting IT virtualization; this approach allows to consolidate several Virtual Network Functions (VNFs) on the same high-volume standard server (e.g., Intel-based blade).

The most notable difference between classical IT virtualization and NFV is the degree of network traffic that has to be handled within a single server, as traditional virtualization has to deal mostly with *compute-bounded* tasks, while *network I/O* represents the dominant factor in the NFV case. This poses non trivial challenges when writing both VNFs and the system framework (e.g., the virtual switch), as many low-level details such as memory access patterns, cache locality, task allocation across different CPU cores, synchronization primitives and more, may have a dramatic impact on the overall performance.

To facilitate the development of network I/O-intensive applications, Intel has proposed the Data Plane Development Kit (DPDK) [56], a framework that offers efficient implementations for a wide set of common functions such as NIC packet input/output, easy access to hardware features (e.g., SR-IOV, FDIR, etc.), memory allocation and queuing.

In the work presented in this chapter[1], we exploit the primitives offered by DPDK

[1]The content of this chapter has already been published in [32]. This work is also partially described in the master thesis of Mauro Annarumma, who collaborated in the development of the

to investigate the case in which a huge number of VNFs are executed simultaneously on the same server, e.g., when the network is partitioned among several players (potentially individual users), each one having a set of VNFs that operate only on the traffic of the player itself. This requires the system to execute many VNF *instances*, leading to a situation in which thousands of VNFs may be running on the same server, although each instance will be characterized by a minuscule workload.

This chapter focuses on the design of the components that deliver (and receive back) the traffic to the VNFs. Particularly, it proposes several possible architectures, each one targeting a specific working condition, for transferring data between the virtual switch (shown in Figure 4.1) and VNFs, exploiting (whenever possible) the primitives offered by the Intel DPDK framework. Our goals include the necessity to scale with the number of VNFs running on the server, which means that we should ensure high throughput and low latency even in case of a massive number of VNFs operating concurrently, each one potentially traversed by a limited amount of traffic.

This chapter is structured as follows. Section 4.2 provides an overview of DPDK, while Section 4.3 describes the general architecture of the framework to (efficiently) provide traffic to a massive number of VNFs. Several implementations of this framework are then provided in Section 4.4, while their performance are evaluated and compared in Section 4.5. Section 4.6 discusses related works, and finally Section 4.7 concludes the chapter and proposes some future extensions to this work.

4.2 DPDK overview

Intel DPDK is a software framework that offers to programmers a set of primitives that help to create efficient (high speed) VNFs on x86 platforms.

DPDK assumes that processes operate in polling mode in order to be more efficient [70] and reduce the time spent by a packet traveling in the server. This would require each process to occupy one full CPU core (in fact, DPDK processes are pinned to a specific CPU core for optimization reasons), hence the number of processes running concurrently is limited by the CPU architecture. Although this scheduling model is not mandatory, DPDK primitives are definitely more appropriate when applications are designed in that way; for example, DPDK does not offer any interrupt-like mechanism to notify a VNF for the arrival of a packet on the NIC.

DPDK supports multi-process applications, consisting of a primary process enabled to allocate resources such as `rte_ring` and `rte_mempools`, which are then shared among all the secondary processes. A DPDK process, in turn, consists of at least one *logical core (lcore)*, which is an application instance running on a CPU core.

prototype.

To manage memory, DPDK offers the **`rte_malloc`** and the **`rte_mempool`**. The former looks similar to the standard *libc* `malloc`, and can be used to allocate objects *(i)* on huge pages (in order to reduce IOTLB misses), *(ii)* aligned with the cache line and *(iii)* on a particular NUMA socket in order to improve the performance of the applications. The **`rte_mempool`**, instead, is a set of pre-allocated objects that can be acquired, and later possibly released, by `lcores` according to their needs. Since the same `rte_mempool` can be shared across `lcores`, a per-core cache of free objects is available to improve performance. In addition to the performance techniques already mentioned with respect to the `rte_malloc`, all objects within the `rte_mempool` are aligned in order to balance the load across different memory channels. This is particularly useful if we always access the same portion of the object, such as the first 64B of packets.

To exchange data among each others, `lcores` can use the **`rte_ring`**, a lockless FIFO queue that allows burst/bulk-single/multi-enqueue/dequeue operations. Each slot of the `rte_ring` contains a pointer to an allocated object, hence allowing data to be moved across `lcores` in a zero-copy fashion. If the `rte_ring` is used to exchange network packets, each slot of the buffer points to an **`rte_mbuf`**, which is an object in the `rte_mempool` that contains a pointer to the packet plus some additional metadata (e.g., packet length).

Finally, the **Poll Mode Driver (PMD)** is the part of DPDK used by applications to access the network interface cards (NICs) without the intermediation (and the overhead) of the operating system. In addition, it also allows applications to exploit features offered by the Intel NIC controllers, such as `RSS`, `FDIR`, `SR-IOV` and `VMDq`. The PMD does not generate any interrupt when packets are available in the NIC, hence the `lcores` that receives packets from the network should implement a polling model. As a final remark, packets received from the network are stored into a specific `rte_mempool`.

4.3 General architecture

The general architecture of our system is shown in Figure 4.1. The virtual switch (vSwitch) (*i*) receives packets from both NICs and VNFs, (*ii*) classifies and (*iii*) delivers them to the proper VNF according to the service chain each packet belongs to. Finally, when a packet has been processed by all the VNFs associated with its service chain, (*iv*) the vSwitch sends it back to the network.

Following the DPDK recommendations, the vSwitch considered in this work operates in polling mode as it is supposed to process a huge amount of traffic (each packet traverses the vSwitch multiple times), while VNFs may follow either the polling or interrupt-based model, depending on considerations that will be detailed in the following section.

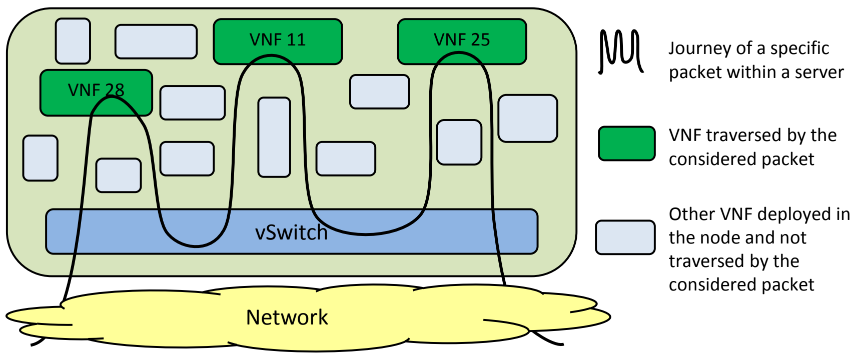

VNF 11
VNF 25
VNF 28
vSwitch
Network
Journey of a specific packet within a server
VNF traversed by the considered packet
Other VNF deployed in the node and not traversed by the considered packet

Figure 4.1: High-level view of a server with a vSwitch and several VNFs.

4.4 Implementations

This section details five possible implementations of the architecture described in Section 4.3, which mainly exploit features offered by DPDK to *(i)* access to the network interface cards, and to *(ii)* move packets among the vSwitch and VNFs. Each implementation is a multi-process DPDK application, where the vSwitch is the primary process (single `lcore`) and each VNF is a different (single `lcore`) secondary process (except for those described in Section 4.4.5).

Unfortunately, vanilla DPDK does not support the execution of two different secondary processes on the same CPU core, which is a fundamental requirement in our use case, since we envision thousands of VNFs deployed on the same physical server. To overcome this limitation, we modify the `lcore_id` internal DPDK variable in the initialization phase of each secondary process, so that each VNF has its own DPDK internal data structures (and then no conflict can arise among VNFs).

4.4.1 Double buffer

In this implementation (Figure 4.2) each physical network interface is configured with a single input and a single output queue; all the packets entering in the node are first processed by the vSwitch, which accesses to the NICs through the PMD library. Each VNF exchanges packets with the vSwitch through a couple of `rte_rings`: one used for the communication vSwitch → VNF, the other used for sending back to the vSwitch those packets already processed by the function itself. Finally, all the elements of the `rte_rings` point to `rte_membufs` in the same `rte_mempool`, allocated by the vSwitch at startup.

In this case VNFs operate in polling mode, hence they never suspend spontaneously. Hence, this implementation is appropriate for those cases in which a limited number of VNFs is active, even not higher than the number of CPU cores available on the server.

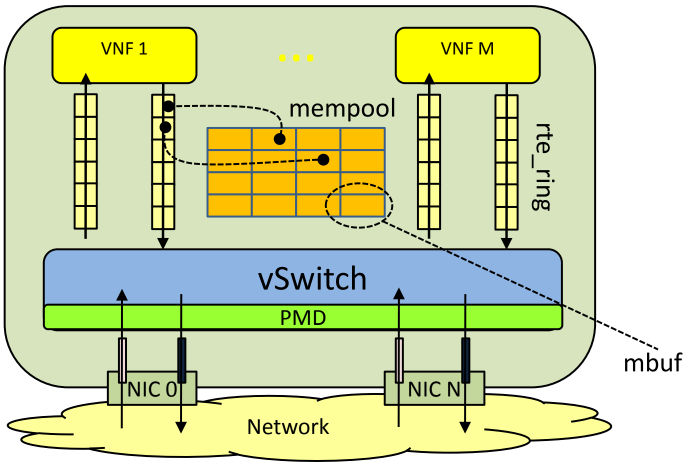


Figure 4.2: Implementation based on a (different) pair of rings shared between the vSwitch and each VNF.

4.4.2 Double buffer + semaphore

In this second implementation, VNFs operate in blocking mode: the vSwitch uses in fact a POSIX named semaphore to wake up a VNF when a given number of packets is available for the VNF itself. When all the packets in the buffer have been processed, the VNF suspends itself and waits for the next signal from the vSwitch. Obviously, this mechanism is complemented by a packet aging timeout that wakes up the VNF if there are packets waiting for too long, hence avoiding data starvation.

This implementation is appropriate when VNFs need to process a limited amount of traffic. In this case, the polling model would unnecessarily waste a huge amount of CPU resources, while a blocking model allows to increase the density of the VNFs active on the same server. In fact, in this case a VNF suspends itself when no packets are available, freeing the CPU that can be used by another VNF that actually has packets to be processed.

4.4.3 Single buffer towards the vSwitch + semaphore

In the implementations described so far, the vSwitch may have to handle a huge number of "downstream" buffers coming from VNFs, which may require a considerable amount of time while working in polling mode.

In this third implementation all VNFs share a single "downstream" `rte_ring` toward the vSwitch, which exploits the lock-free multi-access capability of that structure. This would result in a saving of CPU cycles when iterating on a fewer `rte_rings`, as well as an improved cache effectiveness thanks to the better locality in memory access patterns.

This implementation *may* be appropriate when a large number of VNFs are active on the same server, as we expect that in each one of its running rounds the vSwitch would find a few applications with packets ready to be consumed. Unfortunately, according to [56], the multi-producer enqueue function implemented in DPDK does not allow two or more VNFs executed on the same CPU core to use the same `rte_ring`. Hence, although very appealing, this architecture has not been implemented because it would support only a limited number of VNFs (less than the number of CPU cores).

4.4.4 Double buffer + FDIR

This fourth implementation aims at reducing the load on the vSwitch, which is undoubtedly the most critical component of the system, by allowing some VNFs to receive directly the traffic coming from the NICs.

To this purpose, we use the FDIR (Flow Director) facility, which allows each NIC to be initialized with several input queues (incoming traffic is distributed based on the value of specified packet fields) and a single output queue. Each input queue is then associated with a different VNF, while the output queue is just accessed by the vSwitch, as shown in Figure 4.3. When a VNF is started, the vSwitch adds a new `FDIR` perfect filter on all the NICs, and binds this filter with a specific input queue of each port. This way, the first classification of packets is offloaded to the NIC, hence the vSwitch has just to move packets between VNFs and send on the network those packets already processed by the entire service chain. However, this higher efficiency is paid with more complex VNFs, which have to handle multiple input queues, namely those created by the NIC (accessed through the PMD) and the `rte_ring` shared with the vSwitch.

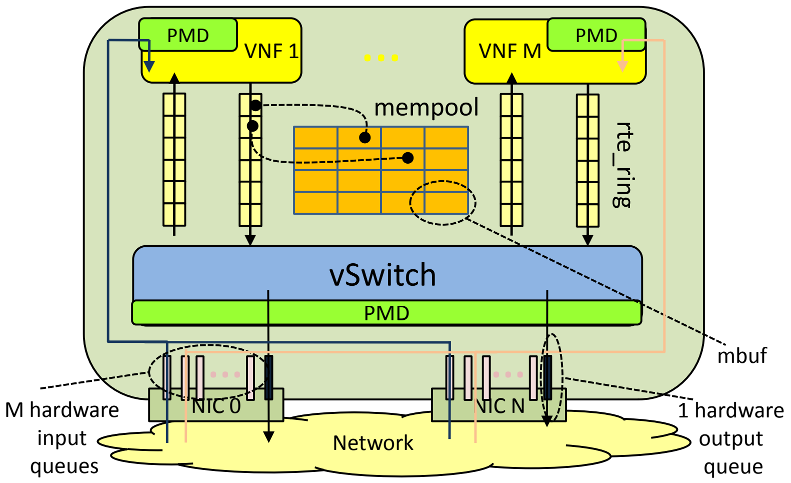


Figure 4.3: Implementation that exploits the `FDIR` feature.

Since the number of hardware queues available on the NICs is limited, this architecture is appropriate when the number of VNFs is reduced. Alternatively, if the number of VNFs is huge, an hybrid architecture may be used: some VNFs only receives traffic from the vSwitch, while others (which are at the beginning of the service chains) are directly connected to a queue of the NIC.

4.4.5 Isolated buffers + semaphore

The last implementation targets the case in which VNFs are not trusted, and hence we cannot allow them to share a portion of the memory space with the rest of system, as in the architectures presented so far[2].

Then, in this implementation only the vSwitch is a DPDK process, while each VNF is a separated (non-DPDK) process. This way, the `rte_mempool` containing traffic coming from the NICs can only be accessed by the vSwitch, which will provide each packet only to the proper VNF. To this purpose, the vSwitch shares with each VNF a distinct set of three buffers: two are similar to the DPDK `rte_rings`, and contain a reference to a slot in the third one, which is actually a simple memory pool containing only the packets exchanged between the vSwitch and the VNF. This requires one additional copy each time a packet has to be delivered to the next VNF in the chain, i.e., from the `rte_mempool` to the per-VNF buffer when the packet has just be received from the NIC, and between those per-VNF buffers in the next steps of the service chain.

Since in this implementation we cannot exploit DPDK features neither in the VNF, nor in the per-VNF buffers, we had to implement (manually) all the techniques provided by the DPDK for efficient packet handling; among the others, buffers starting at a memory address that is multiple of the cache line size and storing each packet to an offset that is multiple of the cache line size.

This implementation aims at providing the adequate *traffic isolation* among VNFs and is appropriate when an operator does not have the control on the VNFs deployed on its network nodes, e.g., when tenants are allowed to install "opaque" VNFs on the network, which are not trusted by the operator itself.

4.5 Performance evaluation

This section evaluates the performance of the implementations described in Section 4.4. Tests are executed on dual E5-2660 Xeon (eight cores plus hyperthreading) running at 2.20GHz, 32GB RAM and one Intel X540-based dual port 10Gbps

[2]In fact, all the processes belonging to the same DPDK application (i.e., the vSwitch and *all* the VNFs, in our use case) share all the data structures created by the primary process, such as the `rte_mempool`, the `rte_rings`, and more.

Ethernet NIC. Two other machines are used respectively as a traffic generator and traffic receiver, connected through dedicated 10Gbps Ethernet links.

Each test lasted 100 seconds and was repeated 10 times, then results are averaged. Each graph representing the maximum throughput is provided with a bars view that reports the throughput in millions of packets per second on the left-Y axis, and a points-based representation that reports the throughput in Gigabit per second on the right-Y axis. Instead, latency measurements are based on the `gettimeofday` Unix system call and include only the time spent by the packets in our system, without the time needed to actually send/receive data on the network.

Tests are repeated with packets of different sizes and with a growing number of running VNFs; moreover, each packet traverses two of these VNFs. Traffic is generated so that two consecutive packets coming from the network must be provided to two different VNFs in order to stress more the system. The size of the buffers has been chosen in order to maximize the throughput of the system.

Our VNFs are simple UNIX processes that simply calculate a signature across the first 64 bytes of each packet, which represents a realistic workload as it emulates the fact that most network applications operate only on the first few bytes (i.e., the headers) of the packet in read-only mode. Moreover, the fact that our VNFs are not full-fledged virtual machines as suggested in the NFV paradigm does not represent a limitation, because we focus on the communication mechanism between the different components, which is orthogonal to the architecture of the components themselves.

Similarly, the vSwitch is a simple virtual switch that supports only forwarding rules based on MAC addresses; while this looks limiting compared to other equivalent components such as Open vSwitch [76], it allows us to focus on the transmit/receive portions of the switch, limiting the overhead due to the presence of other features.

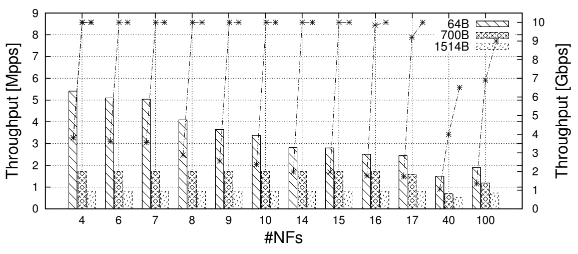
Finally, unless otherwise specified, we used only the CPU whose socket is directly connected to the NIC.

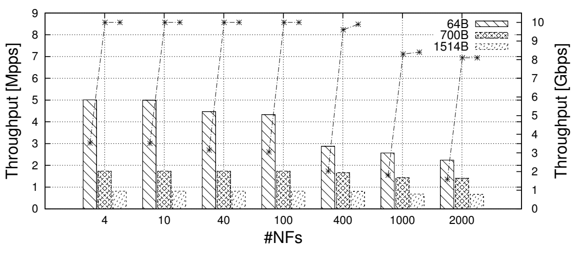
4.5.1 Double buffer

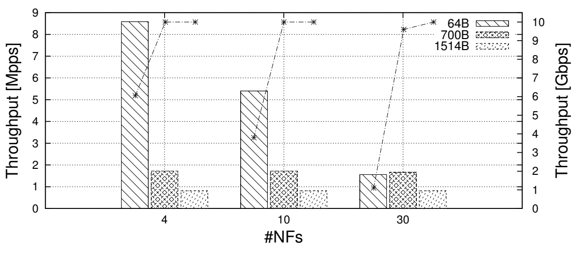
Figure 4.4(a) shows the throughput achieved with a growing number of VNFs deployed on the "double buffer" architecture. In particular, from the figure it is evident that the throughput is maximized when no more than one VNF is executed on a physical core[3]. In fact, it drops of about 20% (with 64B packets) when the number of VNFs changes from 7 to 8, due to the fact that we start allocating VNFs on the logical cores of CPU0 as well, hence having multiple VNFs that share the same

[3]It is worth noting that, for performance reasons, one physical core is always dedicated to the vSwitch; hence, the machine used in the tests has still 7 physical cores (on CPU0) that can be assigned to VNFs.

physical core.


(a) “Double buffer" architecture.


(b) “Double buffer + semaphore" architecture.


(c) “Double buffer + FDIR” architecture.

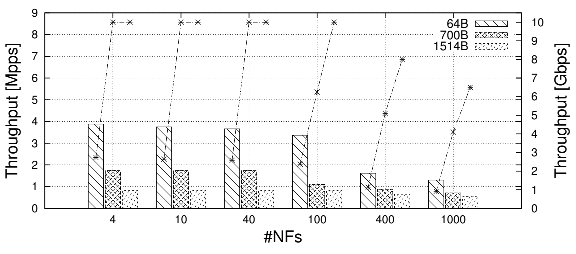
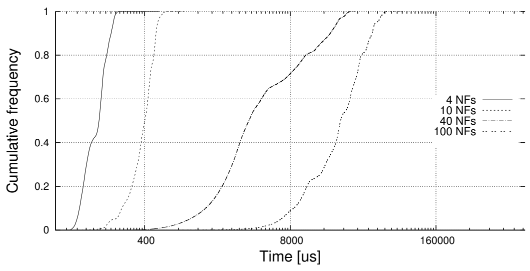

(d) “Isolated buffers + semaphore” architecture.

Figure 4.4: Throughput with a growing number of VNFs.

Figure 4.5(a) plots the latency experienced by packets in our system and shows that its value tends to increase considerably with the number of VNFs, as shown by an average value of 24.44ms with 100 VNFs.


(a) “Double buffer” architecture.

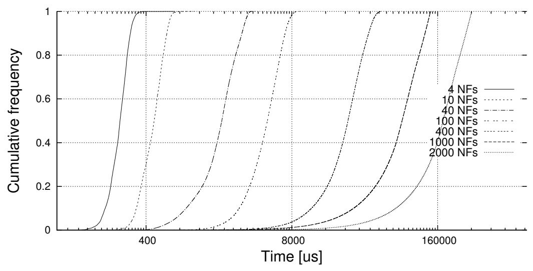

(b) “Double buffer + semaphore” architecture.

Figure 4.5: Latency introduced by the framework.

This degradation of performance (both throughput and latency) when increasing the number of VNFs is a consequence of the execution model implemented in VNFs, which perform a busy waiting on the input `rte_ring` in order to receive packets from the vSwitch, and never suspend themselves. Hence, when there are more VNFs than the number of CPU cores, the CPU could be allocated (by the operating system) to a VNF with no packets to be processed, while a VNF that actually has available packets, may be waiting for a CPU core.

Then, this model looks appropriate only if the number of VNFs is smaller than the number of CPU cores available; after that point the throughput drops and the latency becomes barely acceptable.

4.5.2 Double buffer + semaphore

Figure 4.4(b) depicts the throughput obtained with a growing number of VNFs with the architecture described in Section 4.4.2. In particular, it shows that the VNFs implemented in blocking mode achieve higher throughput than in the previous case, in which they operated in polling mode. This allows the system not only to go faster in any working condition (even when a few VNFs are active), but to support an higher number of VNFs without any significant drop in terms of performance, achieving just over 8Gbps with 700B packets even with 2000 VNFs.

Interesting, the better throughput is not achieved at the expense of the latency, as shown in Figure 4.5(b). In fact, if with a few VNFs we can assist to a negligible worsening (with 4 and 10 VNFs, the average latency is less than 100μs higher compared to Figure 4.5(a)), things become rapidly far better with an higher number of VNFs, achieving an average of 1,89ms and 4,83ms respectively with 40 and 100 VNFs. In fact, a busy waiting model is expected to provide a lower latency as it avoids the overhead of waking up a VNF, and the necessity to queue several packets before waking up a VNF, which obviously impact on the latency. However, all those properties disappear when the number of VNFs exceeds the number of CPU cores, as the operating system has to schedule in/out different processes anyway, hence obtaining a result that looks similar to the blocking model. Moreover, the operating system scheduler is not aware if a VNF has packets to be processed or not, and consequently it could allocate the CPU to a VNF uselessly. Instead, the semaphore enables this implementation to schedule a VNFs only when it actually has packets to be processed.

Finally, it is remarkable the fact that, with 2000 VNFs, the 7 CPU cores allocated to them (the last is dedicated to the vSwitch) are loaded only at 18% in average, which shows the efficiency of the system. However, as evident, the latency is definitely not acceptable (an average of 160ms with 2000 VNFs), that is the reason why we did not try to squeeze even more VNFs on the system, although, from the point of view of the throughput, there was still room for more of them.

4.5.3 Double buffer + FDIR

Figure 4.4(c) shows the throughput achieved with the "double buffer + FDIR" implementation.

In this case we experienced a limitation of the DPDK framework: although the NIC controller exports 64 hardware queues (hence it can distribute the traffic to 64 different consumer processes), the DPDK forces each one of those processes (as part of a multi-process DPDK application) to be allocated on a different CPU core. As a consequence, we were only able to execute 30 VNFs using both the Xeon CPUs available in our server, while the remaining two logical cores were allocated to the operating system and to the vSwitch.

With this limitation, our tests confirm that FDIR could provide a considerable speedup to the system, allowing our server to reach a throughput that is up to 41% better compared to the best of the previous implementations (e.g., 6.05Gbps with 64B packets and 4 VNFs). However, this gain tends to decrease when adding more VNFs (particularly when we start to allocate VNFs on the second CPU, which forces the traffic to traverse the QPI bus), reaching a point, with 30 VNFs, in which this solution does no longer provide advantages at all.

The latency measured in this test case is slightly better than the one provided in Figure 4.5(a), as packets coming from the NIC are immediately delivered to the VNFs without passing in the vSwitch.

4.5.4 Isolated buffers + semaphore

Figure 4.4(d) shows the throughput achieved with the implementation that provides isolation among VNFs. Comparing this graph with that depicted in Figure 4.4(b), it is evident a deterioration in performance, as a consequence of the additional copies needed to guarantee traffic isolation among VNFs. Instead, latency looks very similar to that presented in Figure 4.5(b).

4.6 Related work

This section provides a brief overview of the main proposals that are related to the work presented in this chapter. Particularly, it focuses on other architectures targeted at efficiently steering traffic among VNFs deployed on the same server.

NetVM [55] is a platform built on top of KVM and DPDK, designed to efficiently provide network traffic to VNFs deployed as different virtual machines (VMs). Similarly to our proposals, VNFs and the vSwitch exchange packets through `rte_rings`; however, while the NetVM vSwitch exploits several threads to provide packets to VNFs, our proposals use a single CPU core as we would like to allocate all the others to the (many) VNFs.

ClickOS [64] defines instead an efficient packet exchange mechanism between a vSwitch based on VALE [82], and VNFs implemented as tiny VMs based on Click [71]. Then, the data exchange mechanism proposed is designed to work with VMs (executed by the Xen [23] hypervisor), while the packets exchange architectures introduced in this chapter are agnostic with respect of the technology used to run VNFs (e.g., VMs, Docker containers). Moreover, we propose different solutions to be used according to the number of VNFs actually deployed (and hence, executed concurrently).

Also Xen [23] and Hyper-Switch [78] address the problem of efficiently exchanging packets between VMs running on the same server. However, their architecture is designed for packets that originate or terminate their journey in a VM, not for pass-through VNFs.

Although used to implement interconnections among VNFs, virtual switches such as OpenvSwitch [76] and the eXtensible Datapath daemon (xDPd) [26] are orthogonal to our proposal. In fact, while they implement the classification and forwarding mechanism, such vSwitches do not focus on data exchange mechanisms optimized for the NFV scenario. Then, the packet exchange mechanisms introduced in this chapter can be built on top of those existing solutions to improve their data transfer capabilities, hence the overall performance of the system.

4.7 Conclusion

This chapter focuses on the case in which a massive number of (tiny) virtual network function instances are executed simultaneously on the same server and presents five possible implementations, each one with specific operating characteristics, of a system that moves efficiently packets across the many VNFs running on the server itself. All the proposed implementations are based, as much as possible, on the features offered by the Data Plane Development Kit, a framework recently proposed by Intel to efficiently implement data plane applications.

Results obtained, particularly in terms of throughput, are quite satisfying for almost all the implementations proposed, confirming the goodness of the primitives exported by the DPDK; only in few cases we spotted some limitations which are specific of our target domain. From the point of view of the latency, we experienced huge packet traveling times when the server was packed with many VNFs active at the same time. In general, when the number of VNF exceeded 100, the average latency introduced by the system may become unacceptable in real implementations.

To our view, this suggests that our particular use case, with a massive number of (tiny) VNFs, may not be satisfied with the current generation of the hardware, in which CPUs are dimensioned for a few, fat, jobs, while we have here many, tiny tasks. This suggests that our future investigations should take into consideration different hardware platforms, such as the ones with a massive number of (tiny) cores, which may be more appropriate for our case.

As a future activity, we plan to implement the proposed packet exchange architectures in an existing virtual switch, in order to validate them in real environments.

Chapter 5

Efficient data exchange algorithm for chained virtual network functions

5.1 Introduction

New paradigms have recently emerged that aim at transforming the network into a more flexible and programmable platform. In particular, Network Function Virtualization (NFV) [43] proposes to replace dedicated middleboxes, used to deliver a multitude of network services by means of a growing number of (cascading) dedicated appliances [53], with software images that run on general-purpose servers. This results in a more flexible deployment of network applications (e.g., NAT, firewall) and in lower capital and operating costs for the hardware, thanks to the possibility to deploy many different (small) Virtual Network Functions (VNF) on the same (standard) computer. In addition, while appliances are often dedicated to a single tenant, servers can be multitenant, hence being able to host services belonging to different players, such as a traffic monitor belonging to the operator and a firewall belonging to a given company, with even more advantages in terms of consolidation.

When several VNFs are executed in the same server, an incoming packet can traverse an arbitrary number of VNFs before leaving the middlebox (i.e., a *function chain*, as shown in Figure 5.1). The exact sequence of functions traversed by a packet can be determined only at run-time, by inspecting the packet. In fact, (*i*) packets belonging to different tenants can traverse different functions, and (*ii*) packets belonging to the same tenant can experience different paths (e.g., when only a portion of traffic needs to undergo a deep packet inspection). Packets can also be modified in transit (e.g., a NAT changes the source IP address), hence requiring that a packet is re-analized when it leaves a VNF, to determine what is next. As depicted in Figure 5.1, this requires that each server includes a virtual switch (vSwitch) that classifies each packet to determine which is the next function to traverse and then delivers the packet to it.

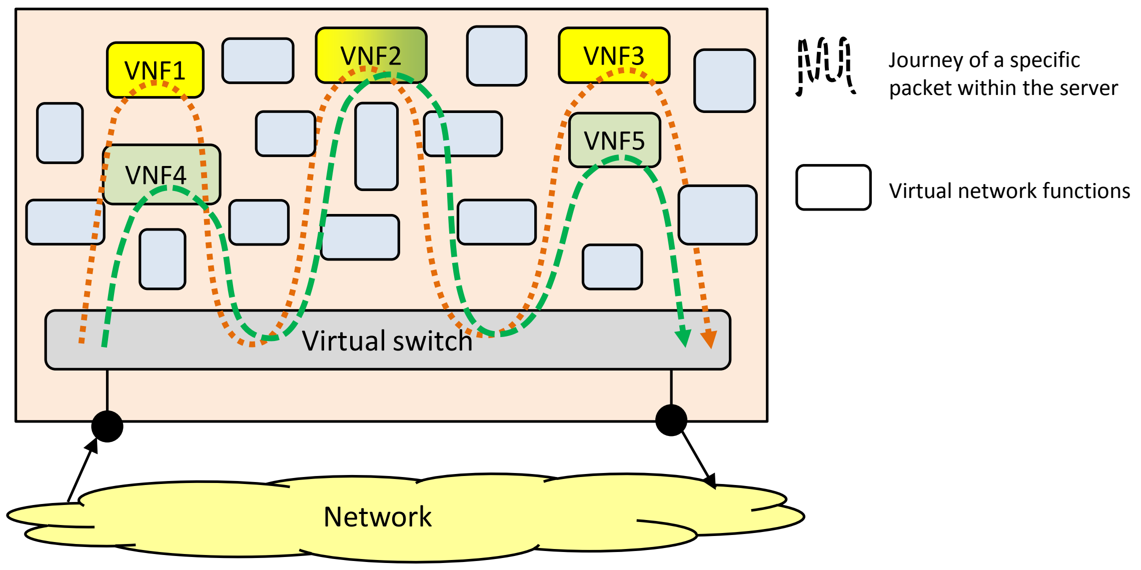

VNF1
VNF2
VNF3
VNF4
VNF5
Virtual switch
Network
Journey of a specific packet within the server
Virtual network functions

Figure 5.1: Function chains deployed in a server.

Based on the experience gained in the work described in Chapter 4, this chapter[1] proposes and evaluates a new architecture for moving network packets between different functions, by means of a vSwitch. This solution, which is based on circular lock-free First-In-First-Out (FIFO) buffers managed by ad-hoc algorithms, is designed to: *(i)* guarantee *traffic isolation* between VNFs, so that a VNF can only access the portion of traffic that is expected to flow through it, hence limiting the potential hazards due to malicious applications and provide an effective support to multitenancy; *(ii)* provide excellent *scalability* by allowing to consolidate a huge number of VNFs on the same server; *(iii)* achieve high *performance* in terms of data movement speed among different VNFs, similarly to what is required in physical servers among different hardware modules [98]. Scalability and performance are obtained also by taking care of implementation details such as exploiting cache locality as much as possible and limiting the number of context switches. The correctness of the data exchange algorithm (e.g. absence of concurrency hazards) is guaranteed by means of formal verification.

This chapter focuses on VNFs that *(i)* implement a pass-through behavior (each packet received is sent again to the network), *(ii)* may drop packets or *(iii)* may

[1]This chapter (partially published in [30]) is also part of the PhD thesis of Matteo Virgilio ("Study and analysis of innovative network protocols and architectures"), who collaborated in this work.

generate new packets as a consequence of a packet just received (e.g., an ARP reply as a consequence of an ARP request). This allows us to cover the vast majority of network middleboxes, including for example NATs, firewalls, traffic monitors, and intrusion detection systems. Instead, applications that may need to generate new packets asynchronously, e.g., in order to open a connection with some remote service or to retransmit a TCP packet, are out of the scope of the work described in this chapter and left as a future work.

The rest of the chapter is organized as follows. Section 5.2 explores related existing solutions able to exchange data among different software components. Given the nature of our solution, we are particularly interested in covering FIFO queue designs and producer/consumer paradigms, in order to emphasize the differences between our work and the existing mechanisms. Section 5.3 details the operating principles of the proposed architecture and the way the data exchange is managed by the different modules. Section 5.4 presents a formal verification of the data exchange algorithm, which rigorously proves its correctness from a safety and security perspective. Section 5.5 proposes some implementation guidelines that can be used to further improve the performance of the data exchange. Section 5.6 presents a wide range of experiments that validate our architecture both in ideal conditions and in real scenarios. Finally, Section 5.7 concludes the chapter.

5.2 Related Work

The efficient *lock-free* implementation of FIFO queues has been largely investigated in the past. For instance, [69] and [46] propose lock-free algorithms that operate on FIFO queues managed as non-circular linked-lists. Similar proposals can be found in [77] and [50], which also require to manage a pool of pre-allocated memory slots. However, all the solutions proposed so far are usually based on uni-directional flows of data according to the *producer-consumer* paradigm, which is not an optimal solution for managing the bi-directional data flows occurring in the virtualized environments we are considering. In fact, in these environments, a packet always goes from the virtual switch to the VNF and then back to the virtual switch. Using classical uni-directional producer-consumer solutions requires the VNF to remove data just received from a first queue and to write them into a second queue used for sending the data back. This implies that data are always copied once in this trip, which may limit the throughput of the system particularly when several VNFs have to be traversed (hence several copies have to be carried out).

Another possible way to efficiently exchange data between applications can be seen in the context of a lock-free operating system, in which [65] and [66] present a single producer/consumer and a multi-producer/multi-consumer algorithm to manage circular FIFO queues. A similar idea has been proposed by Intel in the DPDK library [56] and in [91], whose algorithms have been designed to operate in contexts where many processes can concurrently insert items into a *shared* buffer or remove them. However, those proposals are not applicable in our case because they cannot guarantee isolation between VNFs due to the presence of a unique shared buffer. Similar considerations can be made for ClickOS [64] (based on the VALE virtual switch [82]) and NetVM [55], which instead targets network function chains. ClickOS uses two unidirectional queues with the necessity to copy packets once; NetVM uses two unidirectional queues between "untrusted" functions, while switching to a unique shared buffer (handled in zero-copy) among "trusted" functions, hence impairing traffic isolation requirement. MCRingBuffer [59], instead, defines an algorithm to exchange data between one producer and one consumer running on different CPU cores, which is particularly interesting for its efficient implementation of memory access patterns; in fact, part of those techniques are reused in our implementation as well (Section 5.5).

Solutions such as Xen [23], and Hyper-Switch [78] address the problem of efficiently exchanging packets between different entities such as virtual machines (VM) running on the same server, which looks similar to our problem of chaining network functions. However, their architecture is designed for packets that originate or terminate their journey in a VM, not for pass-through functions. This implies different architectural choices such as different buffers for packets flowing in different directions, albeit integrated with a complex data exchange mechanism based on swapping memory pages rather than copying packets between the hypervisor and the VM [23]. It is also worth mentioning that network-aware CPU management techniques have been proposed in the context of Xen for improving the performance of virtual servers hosting these network applications [48].

Virtual switches such as Open vSwitch (OvS) [76] and the eXtensible Datapath daemon (xDPd) [26] are used to implement network function chains (as shown in Chapter 3 and in [27]), although they appear in some way orthogonal to our proposal. In fact, they implement the classification and forwarding mechanism (either based on the traditional L2 forwarding or on the more powerful Openflow protocol [68]), but do not focus on the data exchange mechanism which is often based on bi-directional FIFO queues (in some case a shared memory can be configured). In this respect, our mechanism can be built on top of those existing solutions to improve their data transfer capabilities, hence the overall performance of the system.

As a final remark, it is worth pointing out that this chapter focuses on the problem of efficiently moving packets between different VNFs within a server, while it does not consider the problem of efficiently receiving/sending packets from/to the network. This aspect, orthogonal to our proposal, is instead considered in [44], [81] and [56].

5.3 The data exchange architecture

This section describes the proposed architecture, designed to efficiently implement function chaining within a single server. In particular, the section first provides an overview of the architecture and then dives into the details of the packet exchange algorithm.

In our architecture, we define the *Master* as the module that plays the role of the vSwitch, while VNFs are represented by modules called *Workers*. Moreover, a *token* is a generic data unit exchanged between the Master and the Workers. The token represents a packet in the NFV use case, but our mechanism could be used to exchange any kind of data, according to the specific use case implemented.

5.3.1 Operating context

VNFs are pieces of software operating on the data plane of the network that, in the vast majority of cases, forward their packets with minimal (or no) changes, allowing packets to continue their journey toward the final destination. However, some functions (e.g., firewall) may need to drop packets, which should not be sent back to the network after their processing. Other functions, instead, may need to send new packets as a consequence of a previously received packet. For example, a bridging module may need to duplicate a broadcast packet several times (e.g., once for each interface of the middlebox) and then provide all these copies to the next functions in the chain. Similarly, another VNF may extend a packet (e.g., by adding a new header) so that it exceeds the MTU of the network; this packet must then be fragmented, and all the fragments must be sent out.

Hence, our architecture must take all these requirements into account and must be able to efficiently move all the above traffic within the middlebox in order to allow flexible function chains. As introduced in Section 5.1, this requires a fast and efficient mechanism to move data between the vSwitch and the VNFs, which translates into the necessity of a dedicated data dispatching mechanism, being this component one of those that has the biggest impact on the system performance.

5.3.2 Architecture Overview

As shown in Figure 5.2, our architecture is based on a set of lock-free ring buffers; in particular, the Master shares two buffers with each Worker, which are managed through different (but not independent) parts of the same exchange algorithm.

The *primary buffer* is used to exchange pass-through tokens, i.e., data that go from the Master to the Worker, and back from the Worker to the Master. In particular, the proposed solution has the peculiarity of allowing the Worker to return data back without any copy. Instead, the *auxiliary buffer* is used to support another kind of traffic we envision in our use case scenario, namely Workers that can possibly

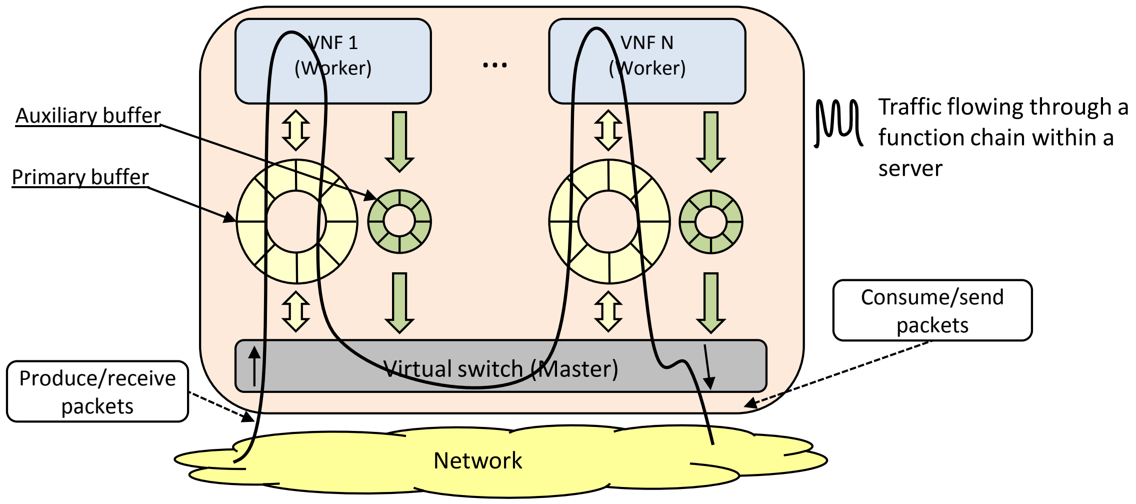


Figure 5.2: Deployment of the algorithm within a server.

generate new tokens as a consequence of the data received from the Master, such as an *ARP reply* packet generated in response to an *ARP request*, or when a packet has to be modified but it results in an excess of the MTU, hence requiring the creation of another packet. This second buffer is unidirectional and it is only used by the Worker to provide "new" data to the Master.

Each buffer slot (both primary and auxiliary) includes some flags in addition to the real data, which are used to identify the content of each slot; more details will be presented in the next sections. Finally, buffer slots are currently of fixed length and equal to the network MTU size; however the extension of the algorithm to handle variable slot sizes, tailored to the length of the packet actually received, is trivial.

5.3.3 Execution model

The Master operates in polling mode, i.e., it continuously checks for new tokens and inserts them into the primary buffer shared with the target Worker. This operating mode has been chosen because the network node (and then the Master itself) is supposed to be traversed by a huge amount of traffic; hence, a blocking model would be too penalizing because it would require an interrupt-like mechanism to start the Master whenever new data are available. This could significantly drop the performance with high packet rates [70]. In fact, interrupt handling is expensive in modern superscalar processors because they have long pipelines and support out of order and speculative execution [40], which tends to increase the penalty paid by an interrupt.

Vice versa, since the traffic entering into a specific Worker is potentially a small portion compared to the one handled by the Master, a blocking model looks more appropriate for this module. This ensures the possibility to share CPU resources more effectively, which is important in multi-tenant systems where potentially a large number of Workers are active. Hence, when a Worker has no more data to be processed, it suspends itself until the Master wakes it up by means of a shared semaphore.

5.3.4 Basic algorithm: handling pass-through data

The algorithm used to move data from the Master to the Workers (and back) requires the sharing of some variables (underlined in the pseudocode shown in the following), a semaphore, and the primary buffer between the Master and each Worker. In particular, in this section we assume the presence of the Master and a *single* Worker, while its extension to several Workers is trivial.

The shared buffer is operated through four indexes. `M.prodIndex` and `W.prodIndex` are shared between the Master and the Worker. The former index points to the next empty slot in the buffer, ready to be filled by the Master, while the latter points to the next slot in the buffer that the Worker will make available to the Master again after its processing. `M.prodIndex` is incremented by the Master when it enqueues new tokens, while `W.prodIndex` is incremented by the Worker when it makes new tokens available to the Master again. `M.consIndex` is a private index of the Master, which points to the next token that the Master itself will remove from the buffer. Finally, `W.consIndex` is a private index of the Worker, which points to the next token to be processed by the Worker itself. In addition to these indexes, the algorithm exploits the shared variable `workerStatus`, which indicates whether the Worker is suspended or it is running.

Algorithm 1 provides the overall behavior of the Master and shows how it cyclically repeats the following three main operations: *(i)* in lines 14-21 it produces new data (line 19), which corresponds to the reception of packets from the network interface card (NIC) in our case, and immediately provides them to the Worker through the primary buffer (line 20); *(ii)* it reads the tokens already processed by the Worker from the primary buffer (line 22), and finally *(iii)* it wakes up the Worker if there are data waiting for service for a long time in order to avoid starvation (line 23). From lines 14-21, it is evident that the Master produces several tokens consecutively, in order to better exploit cache locality. Furthermore, if the buffer is full (line 15), it stops data production and starts removing the tokens already processed by the Worker from the buffer.

Algorithm 2 details the mechanism implemented in the Master to send data to the Worker. As shown by line 8, a token is inserted into the slot pointed by the shared index `M.prodIndex` as soon as it is produced; however, the Worker is awakened only if at least a given number of tokens (i.e., `MASTER_PKT_THRESHOLD`) are waiting for service in the primary buffer (lines 10-13). Thanks to this threshold,

Algorithm 1 Executing the Master

```
1:  Procedure master.do()
2:
3:  {Initialize shared variables}
4:  M.prodIndex ← 0
5:  W.prodIndex ← 0
6:  workerStatus ← WAIT_FOR_SIGNAL
7:
8:  {Initialize private variables of the Master}
9:  M.consIndex ← 0
10: timeStamp ← 0
11:
12: {Execute the algorithm}
13: while true do
14:    for i = 0 to (i < N or timeout()) do
15:       if M.prodIndex == (M.consIndex−1) then
16:          {The buffer is full}
17:          break
18:       end if
19:       data ← master.produceData()
20:       master.writeDataIntoBuffer(data)
21:    end for
22:    master.readDataFromBuffer()
23:    master.checkForOldData()
24: end while
```

we avoid to wake up the Worker for each single token that needs to be processed, hence improving performance because (*i*) it reduces the number of context switches and (*ii*) it increases cache locality, for both data and code. Since a token is inserted into the buffer as soon as it is produced regardless of the fact that the Worker is running or not, and since the Worker will suspend itself only when the buffer is empty (as detailed in Algorithm 5), the Worker is able to process a huge amount of data consecutively, thus improving system performance.

Our algorithm avoids the starvation of tokens sent to a Worker, especially when the system is in underload conditions. This is done by means of a timeout event, which wakes up the worker even if the abovementioned threshold is not reached yet. In particular, the Master acquires and stores the current time whenever it inserts a new token and the buffer is empty (lines 3-6 of Algorithm 2). This way, the Master knows the age of the oldest token and it is able to possibly wake up the Worker also depending on the value of a given time threshold, as shown in Algorithm 3.

The functions described in Algorithm 2 and Algorithm 3 need to know whether the Worker is already running or not in order to avoid useless Worker awakenings.

Algorithm 2 The Master writing data into the primary buffer

```
 1: Procedure master.writeDataIntoBuffer(Data d)
 2:
 3: if M.prodIndex == M.consIndex then
 4:    {The buffer is empty}
 5:    timeStamp ← now()
 6: end if
 7:
 8: buffer.write(M.prodIndex,d)
 9: M.prodIndex++
10: if buffer.size() > MASTER_PKT_THRESHOLD and
    (workerStatus ≠ SIGNALED) then
11:    workerStatus ← SIGNALED
12:    wakeUpWorker()
13: end if
```

Algorithm 3 The Master waking up the Worker due to a timeout

```
1: Procedure master.checkForOldData()
2:
3: if buffer.size() > 0 and (workerStatus ≠ SIGNALED) and
   ((now() − timeStamp) > TS_THRESHOLD) then
4:    workerStatus ← SIGNALED
5:    wakeUpWorker()
6: end if
```

This information is carried by the shared variable `workerStatus`, which is set to `SIGNALED` by the Master just before waking up the Worker (line 11 of Algorithm 2 and line 4 of Algorithm 3), and changed to `WAIT_FOR_SIGNAL` by the Worker just before suspending itself (line 22 of Algorithm 5). This way, the Master can test this shared variable to have an indication about the Worker status, and then wake it up only when necessary.

Algorithm 4 shows how the Master removes the data that have already been processed by the Worker. In particular, it consumes all the tokens until the index `M.consIndex` does not reach the index `W.prodIndex`, incremented by the Worker each time it has handled a batch of tokens, as detailed in Algorithm 5. In this way, also the Master reads several consecutive data from the primary buffer in order to better exploit cache locality.

Notice that Algorithm 4 also considers those tokens provided by the Master to the Worker, and dropped by the Worker itself. In case of dropped data, the Master receives back an empty slot, identified through the flag `dropped`. The content of a slot is only consumed if this flag is zero, otherwise the Master just increments the `M.consIndex` and moves on to the next slot of the buffer, as shown in lines 7-10.

Algorithm 4 The Master reading data from the primary buffer

```
 1: Procedure master.readDataFromBuffer()
 2:
 3: if buffer.size() then
 4:    if M.consIndex ≠ W.prodIndex then
 5:       timeStamp ← now()
 6:       while M.consIndex ≠ W.prodIndex do
 7:          if not buffer.dropped(M.consIndex) then
 8:             master.consumeData(buffer.read(M.consIndex))
 9:          end if
10:          M.consIndex++
11:       end while
12:    end if
13: end if
```

This prevents the Master from reading a slot with a meaningless content.

Algorithm 5 details the operations of the Worker. As evident from lines 12-23, whenever a Worker wakes up, it processes all the tokens available in the primary buffer (i.e., all the slots of the buffer with indexes less than `M.prodIndex`). Only at this point (line 24), as well as after it has processed a given amount of data (lines 13-16), the Worker updates the shared index `W.prodIndex`, so that the Master can consume all the tokens already processed by the Worker itself. This way, the Master will be notified for data availability only when a given amount of tokens are ready to be consumed, with a positive impact on performance. It is worth noting that this batching mechanism is different from the one implemented when the Master sends data to the Worker. In fact, in that case, the Worker is woken up when the amount of data into the buffer is higher than a threshold, while the `M.prodIndex`, used by the Worker to understand when it has to suspend itself, is incremented each time a new data is inserted. Here, instead, the `W.prodIndex` (i.e., the index used by the Master to know when the consuming of tokens must be stopped) is not updated each time the Worker processes a data. As a consequence, it is possible that some tokens have already been processed by the Worker, but it has still to update the `W.prodIndex` and then the Master cannot consume them in the current execution of Algorithm 4. This results in a slightly higher latency for these tokens, but in better performance for the system thanks to this batching processing enabled into the Master. As a final remark, lines 18-20 show that the Worker can drop the token under processing by setting the `dropped` flag in the current slot of the primary buffer.

Figure 5.3 depicts the status of the primary buffer[2] and the indexes used by

[2]For the sake of clarity, the figure represents the shared buffer as an array instead of a circular

Algorithm 5 Executing the Worker

```
**Procedure** worker.do()

{Initialize private variables of the Worker}
W.consIndex $\leftarrow$ 0
pkts_processed $\leftarrow$ 0

{Execute the algorithm}
**while** true **do**
   waitForWakeUp()
   W.consIndex $\leftarrow$ $\underline{\text{W.prodIndex}}$
   pkts_processed $\leftarrow$ 0
   **while** W.consIndex $\neq$ $\underline{\text{M.prodIndex}}$ **do**
      **if** pkts_processed == WORKER_PKT_THRESHOLD **then**
         pkts_processed $\leftarrow$ 0
         $\underline{\text{W.prodIndex}}$ $\leftarrow$ W.consIndex
      **end if**
      toBeDropped $\leftarrow$ $\underline{\text{buffer}}$.process(W.consIndex)
      **if** toBeDropped **then**
         $\underline{\text{buffer}}$.setDropped(W.consIndex)
      **end if**
      W.consIndex++
      pkts_processed++
   **end while**
   $\underline{\text{W.prodIndex}}$ $\leftarrow$ W.consIndex
   $\underline{\text{workerStatus}}$ $\leftarrow$ WAIT_FOR_SIGNAL
**end while**
```

the algorithm in four different time instants. In Figure 5.3(a) the buffer is empty, and then all the indexes point to the same position. Instead, in Figure 5.3(b) the Master has already inserted some data into the buffer, but the Worker is still waiting since the `MASTER_PKT_THRESHOLD` has not been reached yet. Figure 5.3(c) depicts the situation in which the Master has woken up the Worker, which has already processed two items. Notice that, since the `WORKER_PKT_THRESHOLD` has not been reached yet, the `W.prodIndex` still points to the oldest token in the buffer. Instead, in Figure 5.3(d) this threshold is passed and the Master has already consumed some data.

FIFO queue.

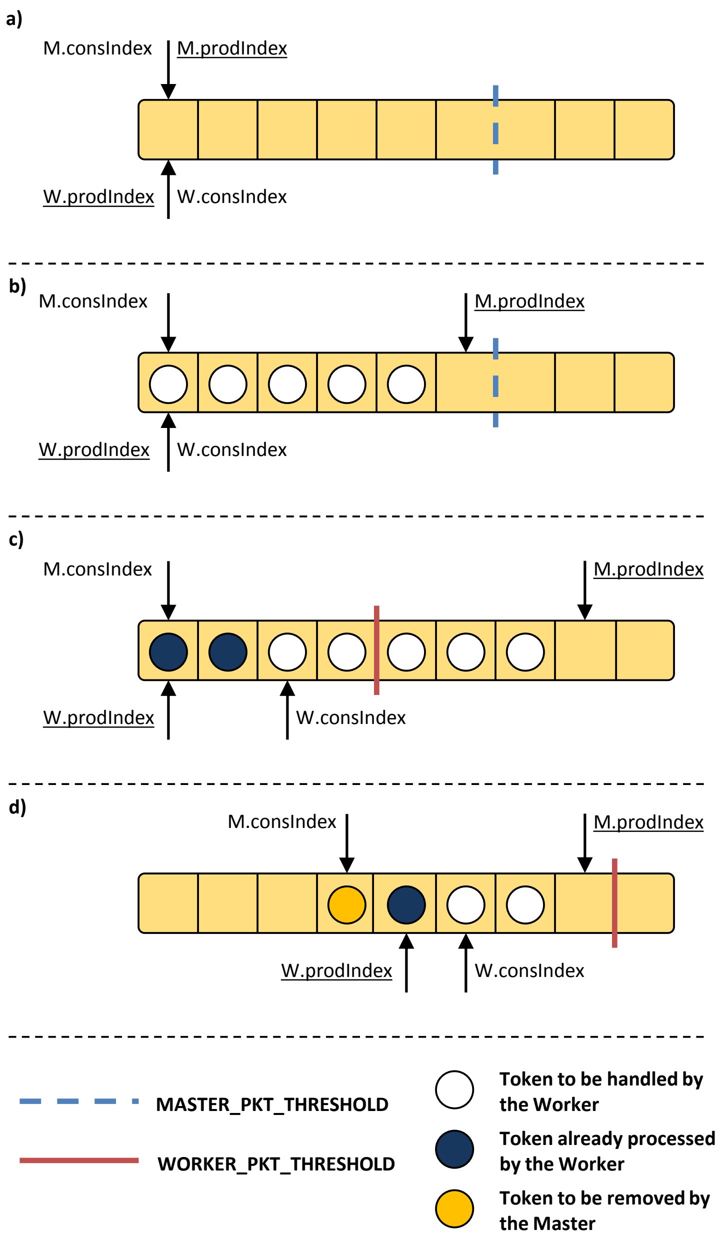


Figure 5.3: Run-time behavior and indexes of the algorithm.

5.3.5 Extended algorithm: handling Worker-generated data

Our architecture handles also Workers that may need to generate *new* data as a consequence of the token just received from the Master but, as evident, this cannot be done with the primary buffer alone as Workers cannot *inject* new data into the primary buffer. In fact, the Worker can just modify (potentially completely) pass-through tokens, i.e., data received from the Master that must be sent back to the

Master itself or, at most, it can drop these tokens.

Since network applications forward most of the packets without performing any manipulation on it, we decided to keep the primary buffer as simple as possible for the sake of speed, while adding a new lock-free ring buffer, i.e., the *auxiliary buffer*, to handle the case in which new data have to be provided to the Master. This buffer, in which the Worker acts as the producer while the Master plays the role of the consumer, is managed through two indexes; moreover, it requires a further flag in each slot of the primary buffer, which indicates whether the next token should be read from the primary or the auxiliary buffer.

Algorithm 6 details how the Worker sends new data to the Master, as a consequence of the processing of the token at position `W.consIndex` in the primary buffer. As shown in lines 3-11, several data can be generated for a single token received from the Master, which are all linked to the same slot of the primary buffer. A first flag, called `aux`, is set in the slot of the primary buffer to signal that the next slot to read is the one on top of the auxiliary buffer (line 13). Instead, the `next` flag set in a slot of the auxiliary buffer tells that the next packet has still to be read from the auxiliary buffer, instead of returning to the next slot of the primary buffer.

Algorithm 6 The Worker writing *new* data into the auxiliary buffer

```
1:  Procedure worker.writeDataIntoAuxBuffer(Data[] newData, Index W.consIndex)
2:
3:  while data ← newData.next() do
4:     if auxProdIndex == (auxConsIndex-1) then
5:        {The auxiliary buffer is full}
6:        break
7:     end if
8:     auxBuffer.write(auxProdIndex,data)
9:     auxBuffer.setNext(auxProdIndex)
10:    auxProdIndex++
11: end while
12: auxBuffer.resetNext(auxProdIndex-1)
13: buffer.setAux(W.consIndex)
```

The reading procedure is described in Algorithm 7. When the Master encounters a slot with the `aux` flag set in the primary buffer, it processes a number of tokens in the auxiliary buffer, starting from the slot pointed by `auxConsIndex` until the `next` flag is set. Moreover, according to lines 4-7 of Algorithm 6, if the auxBuffer is full, new tokens that the Worker may want to send to the Master are dropped.

Figure 5.4 depicts the primary buffer with some slots linked to the auxiliary buffer. In particular, the slot pointed by `M.consIndex` is associated with two data of the auxiliary buffer, i.e., the one pointed by `auxConsIndex` and the following one, which has the `next` flag reset to indicate that the next slot is not linked with the

Algorithm 7 The Master reading data from the auxiliary buffer

```
Procedure master.readDataFromAuxBuffer()

while true do
    master.consumeData(auxBuffer.read(auxConsIndex))
    if not auxBuffer.next(auxConsIndex) then
        auxConsIndex++
        break
    end if
    auxConsIndex++
end while
```

current slot in the primary buffer. Instead, the next token in the primary buffer is not associated with the secondary buffer (the `aux` flag is reset), while the third slot contains data dropped by the Worker; despite this, the slot is linked to three data in the auxiliary buffer. In other words, the configuration in which `aux == 1` and `dropped == 1` is valid and it enables to completely replace a packet with a new one.

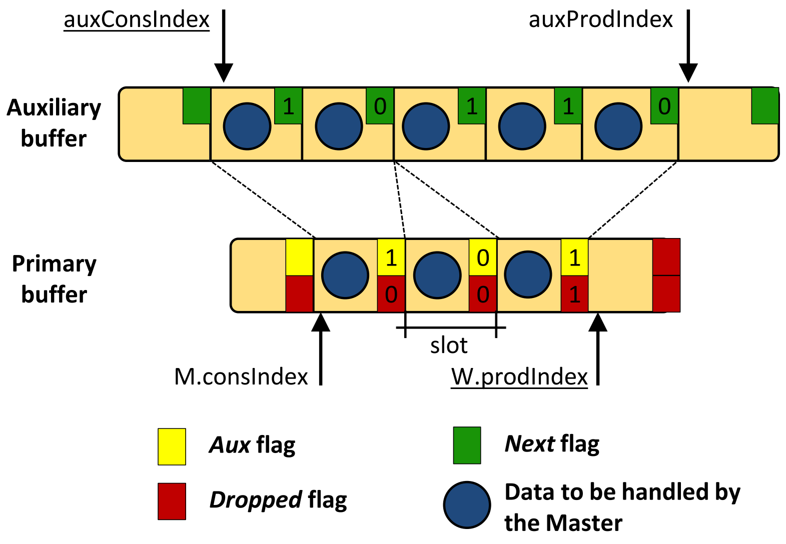


Figure 5.4: Binding primary buffer - auxiliary buffer.

5.4 Formal verification

Assessing the correctness of an algorithm is often not straightforward, hence we built an abstract model of the Master with a single Worker in order to formally check some fundamental properties. We do not consider a plurality of Workers because the interaction between the Master and a Worker is independent of the interaction with any other Worker, hence this approach is sufficient to demonstrate the correctness of the whole system. In particular, we only focus on the primary buffer as its operation is one of the main contributions of our work and hence it needs a proof of correctness. The auxiliary buffer, instead, is not explicitly verified as it is managed as a standard producer/consumer system, which has been already studied and validated in the existing literature [39].

The model of our algorithm has been developed in Promela [51], a well known modeling language that, in conjunction with the SPIN [52] model checker generator, can be used to formally verify distributed and concurrent software against certain desired properties. The main purpose of the model checking technique is to explore all the possible states of a system and verify whether the specified properties hold in *each* execution path. Whenever the model checker finds an execution path that leads to a property violation, it provides the full counter-example with all the steps needed to reach the undesired behavior.

When creating an accurate model of the system, it is very important to keep the nature of the problem *tractable*, as model checking verification tools tend to exploit a massive amount of memory (state-space explosion problem). Therefore, the actual model of the data exchange mechanism has been built by omitting some implementation details that are not relevant for the analyzed properties in order to reduce the overall number of states. This is possible because many system states (or runs) are mapped to the same abstract state (or run). A more detailed description of our model will be provided in Section 5.4.2.

5.4.1 Properties specification

Given the structure of our algorithm, we can identify six properties that *must* be always satisfied. The first two properties refer to conditions on some key variables that must be verified to guarantee that no slot will be erroneously overwritten, formally defining regions of the buffer that are "owned" by one of the two modules (the Master and the Worker) at a given time.

Property 1. *`W.prodIndex` must never exceed `M.prodIndex`.*

`M.prodIndex` indicates the first empty position in the primary buffer that must be fulfilled by the Master. Hence, it represents a boundary that the Worker must never pass.

Property 2. *`M.consIndex` must never exceed `W.prodIndex`.*

`M.consIndex` represents the position of the token being processed by the Worker, while `W.prodIndex` identifies the position of the last "processable" token (for the Worker).

We also consider two additional safety properties, which must be satisfied by the system. Specifically we require that:

Property 3. *The number of pending tokens delivered by the Master to the Worker and not yet processed by the Worker itself is, at any time, a non negative integer not exceeding the maximum number of elements that the buffer can store, namely (N - 1), where N is the total buffer size:*

$$0 <= tokens\_master\_to\_worker <= (N - 1)$$

Property 4. *The number of pending tokens delivered by the Worker to the Master and not yet processed by the Master itself is, at any time, a non negative integer not exceeding the maximum number of elements that the buffer can store, namely (N - 1), where N is the total buffer size:*

$$0 <= tokens\_worker\_to\_master <= (N - 1)$$

Our circular buffer implementation always leaves at least one empty position, in order to distinguish the cases in which the buffer is completely full or completely empty. This is why the actual buffer capacity is N-1.

Finally, we consider two more properties related to the overall system behavior.

Property 5. *Deadlock absence.*

This property is automatically checked by SPIN, and in our case it means that neither the Master nor the Worker ever enter an infinite waiting situation.

Property 6. *Livelock absence.*

This last property ensures that some useful work is eventually done by the Master. Here the notion of "useful work" is intended as the Master capability to produce, sooner or later, new tokens for the Worker, e.g., by inserting new data into the buffer. This is an important property verified under the assumption that a fairness constraint exists during the verification phase, i.e., we assume the process scheduler gives the possibility to both the Master and the Worker to execute, sooner or later, some instructions. This is a reasonable hypothesis since most modern execution environments implement scheduling algorithms to avoid process starvation.

5.4.2 Model details

The primary buffer

Our abstract model does not require the modeling of realistic data into the buffer but only the buffer status; hence, only indexes are modeled. Another parameter that is crucial for the model is the buffer size, meaning the actual number of tokens that can be stored into the buffer.

The semaphore and the functions implementation

The model of the semaphore consists in an asynchronous channel shared between the master and the worker. Basically, the *blocking wait* operation corresponds to reading a packet from the channel, while the *signaling* operation is modeled by writing a packet into the channel. This is a very common pattern, useful to implement various kinds of communication/synchronization primitives between two or more entities.

The functions presented in the pseudocode in Section 5.3 are modeled as Promela processes since the language does not provide an explicit way to represent functions and their returned value. We exploit the following pattern: the caller sends a token through a synchronous channel shared with the callee in order to pass the control to the invoked process. Then, it performs a *receive* operation on the same channel in order to be awakened from the other end-point when the processing has terminated. Notice that the channel can also be used to pass arguments to and from the called process/function.

The Master and the Worker

The two main entities, the Master and the Worker, are modeled as two independent, concurrently running processes. They share the `M.prodIndex` and `W.prodIndex` variables, and the channel/semaphore (as stated in our pseudo-code in Section 5.3.4). In order to decrease the amount of states to be verified by the model checker, and hence reduce the overall verification time to a reasonable value, we use the following abstractions: *(i)* the if-statement of Algorithm 3 excludes the check on the timestamp value as the whole model does not contain any explicit information about the elapsing time; *(ii)* the `timeout()` function that is present in the loop guard (Algorithm 1) is replaced by a non-deterministic choice (i.e., rather than modeling a realistic mechanism to implement a timeout event, we instructed the model checker to extract a random value to decide if a timeout has occurred or not). Both these abstractions provide a significant performance enhancement without any loss in terms of exhaustiveness of the verification.

	Parameters			
	BUFFER SIZE	MASTER THRESHOLD	WORKER THRESHOLD	**VERIFICATION RESULT**
Property 1	[1,8]	[1,8]	[1,8]	SUCCESS
Property 2	[1,8]	[1,8]	[1,8]	SUCCESS
Property 3	[1,8]	[1,8]	[1,8]	SUCCESS
Property 4	[1,8]	[1,8]	[1,8]	SUCCESS
Property 5	[1,8]	[1,8]	[1,8]	SUCCESS
Property 6	[2,8]	[1,8]	[1,8]	SUCCESS

Table 5.1: Algorithm verification.

5.4.3 Verification results

The model explained above can be exhaustively verified for different values of the main model parameters, as shown in Table 5.1. For each property, the table specifies the considered range of values for the buffer size, the `MASTER_PKT_THRESHOLD` and the `WORKER_PKT_THRESHOLD`. For the sake of scalability of the verification process and without losing in generality, we used rather small values compared to a realistic buffer, which could contain millions of tokens. In fact, possible structural bugs would be detected also in a small size system deployment.

Some inconsistent parameters settings in the considered ranges, such as a threshold greater than the buffer size, are skipped in our verification work. Notice also that, with a buffer size equal to one token, Property 6 is not considered as the buffer cannot contain any token and therefore the master is not able to perform any useful work, according to our definition. Properties 1-5 are verified even without forcing any fairness criterion between the execution of the Worker and the Master, since their satisfaction does not necessarily depend on a particular sequence of processes scheduling.

In conclusion, the results of our verification process completely demonstrate the correctness of the algorithm from different points of view (absence of indexes misbehavior or accidental packets overwriting, and absence of deadlocks and livelocks).

The complete Promela code is publicly available at [34].

5.5 Implementation

Since the achievable performance depends not only on design but also on implementation issues, this section presents some implementation choices that can improve performance and scalability and that have been adopted in our ptototype implementation.

Private copies of shared variables. As in many algorithms derived from the producer-consumer problem, also in our case we need to keep two processes in sync by means of a pair of shared variables, one written only by the first process, the other one written only by the second process. Although in this case concurrency issues are limited (no contention can occur because the two processes never try to write the same variable at the same time), the actual implementation on real hardware can introduce additional issues, as shown in MCRingBuffer [59]. In fact, when a first CPU core modifies the content of a variable that is shared with a different CPU core, the entire cache line (64 bytes long on the modern Intel architectures) of the second core containing that variable is invalidated. If the second core needs to read that variable, the hardware has to retrieve this value either from the shared cache (e.g., the L3 in many recent Intel architectures) or from the main memory, with a consequent performance penalty. In our algorithm, this problem occurs for `M.prodIndex`, incremented by the Master whenever a new token is inserted into the primary buffer and read by the Worker, and for `W.prodIndex`, incremented by the Worker and read by the Master. However, our algorithm is robust enough to operate correctly even if those variables are not perfectly aligned. As a consequence, the Worker creates a private copy of `M.prodIndex` just after waking up, while the Master copies in a private variable the content of `W.prodIndex` before reading data from the shared buffer. The Master and the Worker can perform their operations according to the value of their local copies, which are re-aligned with the actual values only periodically; this does not preclude the correct system operation while ensuring a significant reduction of cache misses.

Shared variables on different cache lines. Because of the same problem mentioned in the previous paragraph (a CPU core can invalidate an entire line of cache in the other cores), our code implements a cache line separation mechanism (similar to MCRingBuffer [59]), which consists in storing each shared variable (possibly extended with padding bytes) on a different cache line. This way, when the Master changes, for instance, the value of `prodIndex`, the cache line containing `workerIndex` is not invalidated in the private cache of the core where the Worker is executed.

Alignment with cache lines. In case of a cache miss, the hardware introduces a noticeable latency because of the necessity to transfer the data from the memory to the cache, which happens in blocks of fixed size (the *cache line*). From that moment, all the memory accesses within that block of addresses are very fast, as data are served from the L1 cache. In order to minimize the number of cache misses (and the associated performance penalty), our prototype was engineered to align the most frequently accessed data so that they span across the minimum set of cache lines. In particular, the starting memory address of the packet buffers and their slot sizes are multiple of the cache line size; the same technique is used for minimizing the time for accessing the most important data used in the prototype.

Use of huge memory pages. Huge pages are convenient when a large amount of memory is needed because they decrease the pressure on the Translation Lookaside Buffer (TLB) for two reasons. First, the load of virtual-to-real address translation is split across two TLBs (one for huge pages and the other for normal memory), preventing normal applications (based on normal pages) from interfering with the packet exchange mechanism (which uses huge pages). Second, they reduce the number of entries in the TLB when a large amount of memory is needed. We use the huge pages for the shared (primary and auxiliary) buffers; the drawback is the potential increase of the total memory required by the algorithm because the minimum size of each buffer increases from 4KB to 2MB.

Preallocated memory. Dynamic memory allocation should be avoided during the actual packet processing, as this would heavily decrease the performance of the whole system. In our case, all the buffers used by the packet exchange mechanisms are allocated at the startup of each Worker, allowing the system to add/remove workers at run-time while at the same time avoiding dynamic memory allocation.

Emulated timestamp. Getting the current time is usually rather expensive on standard workstations as it requires the intervention of the operating system and, often, an I/O operation involving the hardware clock. In our case we emulate the timestamp that is needed to wake up a Worker when packets are waiting for service for too long time, by introducing the concept of *current round*, that is the number of loops executed by the Master in Algorithm 1. As a consequence, our implementation schedules a Worker for service when there are packets waiting for more than N rounds; this number can be tuned at run-time based on the expected load on the Master.

Batch processing. Batch processing is convenient because it keeps a high degree of code locality, with a positive impact on cache misses. Our prototype implements batch processing whenever possible, e.g., the Master reads all waiting packets from a worker before serving the next, and Workers process all the packets in their queue before suspending themselves; the drawback is the potential increase of the latency in the data transfer.

Semaphores. A simple POSIX semaphore is used to wake up a Worker that has data waiting to be processed (i.e., at least `MASTER_PACKET_THRESHOLD` packets are queued, or some packets are waiting for long time and the timeout expires). Although POSIX semaphores are implemented in kernel space, their impact is acceptable as they are rarely accessed by algorithm design. Instead, no explicit signal is used in the other direction: the shared variables `M.consIndex` and `W.prodIndex` are used by the Master to detect the presence of packets that need to be read from the buffer.

Threading model. Context switching should be avoided whenever possible because of its cost, particularly when this event happens frequently (such as in packet processing applications, which are usually rather simple and often handle a few packets in a row). For this reason, the Master is a single thread process, cycling on a busy-waiting loop and consuming an entire CPU core, while Workers (which are single-thread processes as well) work in interrupt mode and share the remaining CPU cores. While the Master can be simply parallelized over multiple cores as long as the function chains are not interleaved[3], by design our implementation keeps it locked to a single core as we would like to allocate the most part of the processing power to the (huge number of) Workers, which will host the network functions that are in charge of the actual (useful, from the perspective of the end users) processing.

5.6 Experimental results

In order to evaluate performance and scalability (using the implementation choices described in the previous section), we carried out several tests on our prototype implementation running on a workstation equipped with an Intel i7-3770 @ 3.40 GHz (four CPU cores plus hyperthreading), 16 GB RAM, 16x PCIe bus, a couple of Silicom dual port 10 Gigabit Ethernet NICs based on the Intel x540 chipset (8x PCIe), and Ubuntu 12.10 OS, kernel 3.5.0-17-generic, 64 bits. An entire CPU core is dedicated to the Master; instead, Workers have been allocated on the remaining CPU cores in a way that maximizes the throughput of the system. All the following graphs are obtained by averaging results of 100s tests repeated 10 times.

The data exchanged among the Master and the Workers consists of synthetic network packets of three sizes, 64 bytes to stress the forwarding capabilities of the

[3]Interleaved chains may introduce additional complexity because multiple masters may collide when feeding a single Worker; this would require an extension of our algorithm (no longer lock-free) that is left to a future work.

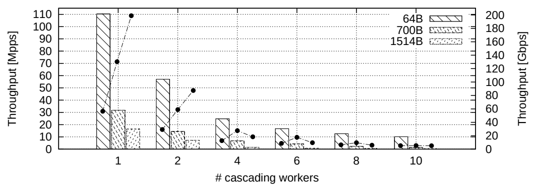
chain, 700 bytes that matches the average packet size in current networks, and 1514 bytes to stress the data transfer capabilities of the system. We present first a set of experiments where packets exchanged between the Master and the Workers are directly read/written from/to the memory, without involving the network; those tests aim at validating the performance of the algorithm in isolation, without any disturbance such as the cost introduced by the driver used to access to the NIC or the overhead of the PCIe bus. Later, Section 5.6.6 will present some results involving a real network, where the workstation under test is connected with a second workstation acting as both traffic generator and receiver, with two 10Gbps dedicated NICs. This setup allows to derive the precise latency experienced by packets in our middlebox. In this case we use the PF_RING/DNA drivers [44] to exchange packets with the NIC, which allows the Master to send/receive packets without requiring the intervention of the operating system. In addition, data coming from the network is read in polling mode in order to limit additional overheads due to NIC interrupts, and in batches of several packets in order to maximize code locality. Similar techniques are used also when sending data to the network after all the processing took place.

5.6.1 Single chain - Throughput

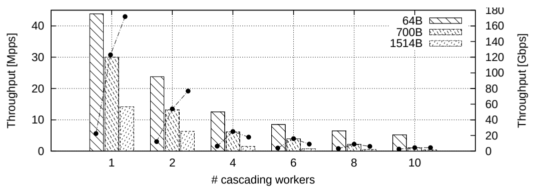
This section reports the performance of our algorithm in a scenario where all packets traverse the same chain, which is statically defined. Tests are repeated with chains of different lengths and the measured throughput is provided in graphs that include *(i)* a bars view corresponding to the left Y axis that reports the throughput in millions of packets per second and *(ii)* a point-based representation referring to the right Y axis, that reports the throughput in Gigabits per second.

Figure 5.5 shows the throughput offered by the function chain in different conditions. As expected, the overall throughput of the chain (i.e., the packets/bits that exit from the chain) decreases with the number of Workers because of our choice to reserve the most part of the CPU power to the Workers, hence limiting the Master to a single CPU core.

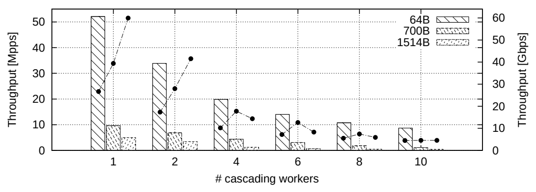
Figure 5.5(a) shows the throughput that could be achieved in ideal conditions, that is: *(i)* with dummy Workers, i.e., that do not touch the packet data, and *(ii)* with the Master always reading the same input packet from memory and copying it into the buffer of the first Worker of the chain, which reduces the overall number of CPU cache misses experienced at the beginning of the chain. This provides an ideal view of the system, where the penalties due to memory accesses are kept to a minimum. Results reported in Figure 5.5(b) are instead gathered in a more realistic scenario, i.e., with Workers that access to the packet content and calculate a simple signature across the first 64 bytes of packets. This may represent a realistic workload, as it emulates the fact that most network applications operate on the first



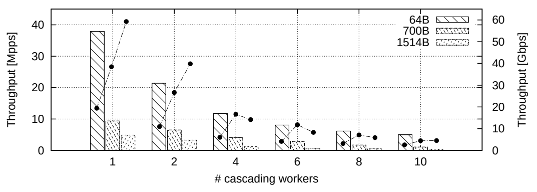
(a) Dummy Workers and a single packet in memory.



(b) Real Workers and a single packet in memory.



(c) Dummy Workers and 1M packets in memory.



(d) Real Workers and 1M packets in memory.

Figure 5.5: Throughput of a single function chain with the algorithm presented in this chapter.

bytes (i.e., the headers) of the packet. This test shows that performance is reduced compared to Figure 5.5(a) for two reasons: (*i*) because of the higher number of cache misses generated by the cores assigned to the Workers and caused by the Workers accessing to the packet content, and (*ii*) because of the additional processing time spent by the Workers for completing their job.

Next tests consider a scenario where the input data for the chain is stored in a buffer containing 1M packets, thus emulating a real middlebox that receives traffic from the network. In particular, Figure 5.5(c) refers to a scenario with dummy Workers such as in Figure 5.5(a) and shows how an apparently insignificant different memory access pattern can dramatically change the throughput. In fact, the Master experiences frequent cache misses when reading packets at the beginning of the chain. This modification alone halves the throughput compared to Figure 5.5(a), particularly when packets have to traverse chains of limited length, while in case of longer chains this additional overhead at the beginning is amortized by the cost of the rest of the chain.

Finally, Figure 5.5(d) depicts a realistic scenario where Workers access the packet content (such as in Figure 5.5(b)), and the Master feeds the chain by reading data from a large initial buffer (1M packets). Even in this case our algorithm is able to guarantee an impressive throughput, such as about 38 Mpps with 64B packets.

In order to confirm that, with the current workload, the Master represents the bottleneck of the system, Figure 5.6 shows the internal throughput of the chain, namely the total number of packets moved by the Master, with an increasing number of Workers, in the same test conditions of Figure 5.5(d). This figure gives an insight of the processing capabilities of the Master, which slightly increases with a growing number of Workers and proves the effectiveness of our algorithm as the number of packets it processes essentially does not depend on the number of Workers.

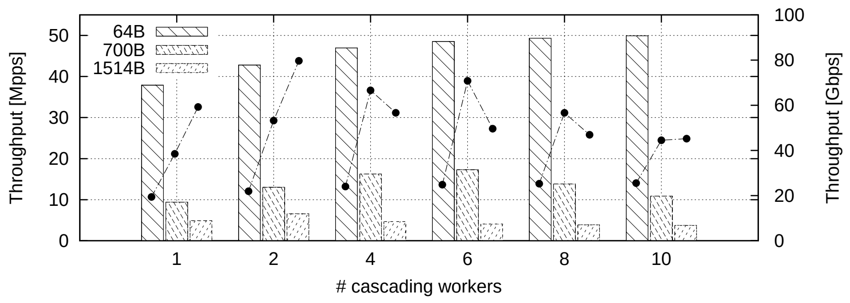


Figure 5.6: Internal throughput of the function chain, with real Workers and a 1M packets in memory.

5.6.2 Single chain - Latency

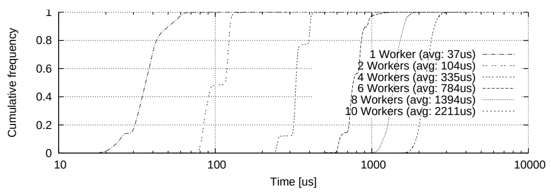
Some architectural and implementation choices, such as working with batches of packets, aim at improving the throughput but may badly affect the latency. For this reason, this section gives an insight about the latency experienced by packets traversing our chains. Measurements are based on the `gettimeofday` Unix system call and, in order to reduce its impact on the system, only sampled packets (one packet out of thousand) have been measured.

Figure 5.7(a) shows the latency of 64B packets when traversing a function chain consisting of a growing number of Workers, in case of real Workers and 1M packets in memory. As expected, the latency increases with the length of the chain; however its value is definitely reasonable for most of networking applications, reaching an average value of about 2.2ms in case of 10 cascading Workers, being far less with shorter (and more realistic) chains.

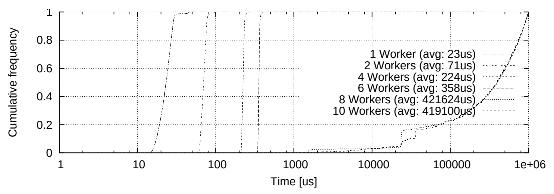
5.6.3 Single chain - Comparison with other approaches

This section aims at demonstrating the advantages of our data exchange algorithm by comparing our proposal with two other approaches that could be used to exchange packets between the Master and the Workers.

In this respect, we cannot directly compare our algorithm with existing solutions such as VALE [82], OvS [76] and xDPd [26], because they include the overhead of packet classification (e.g., L2 forwarding, Openflow matching), which would affect the performance of the data exchange algorithm. As a consequence, we distilled the fundamental design choices of the most important alternative approaches and



(a) Our algorithm.

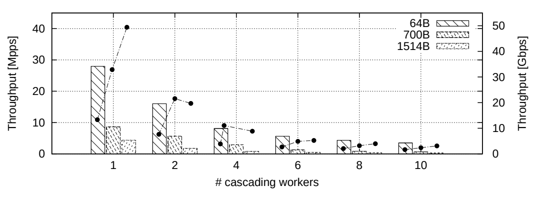


(b) Zero-copy buffers among the Master and the (polling) Workers.

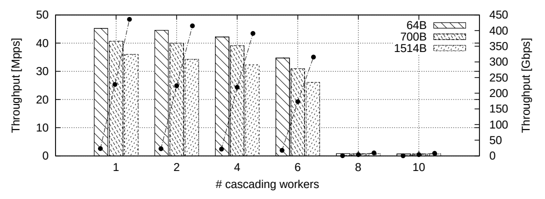
Figure 5.7: Latency introduced by the function chain with a growing number of cascading Workers.

we carefully implemented them in a way that they could be compared with our algorithm, implemented by using, whenever applicable, the guidelines listed in Section 5.3. Particularly, the comparison aims at validating the advantages of two important aspects of our algorithm: the absence of a data copy in the Worker, and the blocking mode operating model of the Worker.

The first alternative approach we compare with is based on the traditional producer/consumer paradigm, in which the Master shares two buffers with each Worker: the first is used by the Master to provide packets to the Worker, while the second operates in the opposite direction. The second approach closely follows the processing model suggested by Intel in the DPDK library [56]: two buffers (based on the traditional producer/consumer paradigm) are shared between the Master and each Worker. However, these buffers contain pointers, which means that the actual data is stored in a shared mempool and never moved between the components of the function chain (zero-copy). Moreover, both the Master and Workers operate in polling mode. Although this solution neither provides isolation among the Workers, nor limits the CPU consumption, it is compared with our proposal because nowadays it represents the “standard” way to implement network function chains.



(a) Unidirectional buffers shared between the Master and the Workers.



(b) Zero-copy buffers among the Master and the (polling) Workers.

Figure 5.8: Throughput of a single function chain when other data exchange algorithms are used.

Tests are executed in realistic conditions, namely with Workers accessing packets and 1M packets in memory, and therefore the above results should be compared with the performance obtained in Figure 5.5(d).

As expected, the throughput of the chain drops of about 30% when unidirectional buffers are used, as shown by comparing Figure 5.5(d) and Figure 5.8(a). This is mainly due to the operating principles of our primary buffer, which allows the Worker to send back a packet to the Master without moving the packet itself, while in this alternative approach one additional data copy in the Worker has to be performed.

Instead, the second alternative approach slightly outperforms our algorithm until the number of jobs (one Master plus N Workers) is lower than the number of available CPU cores, as evident by comparing Figure 5.8(b) with Figure 5.5(d). This is due to the absence of data copies and to the polling-based operating mode implemented in the Workers. However, a stronger performance degradation with respect to our solution (it offers less than 1 Mpps throughput) is noticeable when 8 (or more) Workers are active because at least two of them have to share the same CPU core.

The second alternative approach has also been evaluated in terms of latency introduced on the flowing packets. Similarly to what happens for the throughput, it outperforms our proposal when the number of jobs running is less than the number of CPU cores, as evident by comparing Figure 5.7(a) and Figure 5.7(b). For instance, six chained Workers introduce an average latency of 358μs, agains the 784μs obtained with our algorithm. Instead, in case of more Workers, the average latency of the second alternative approach reaches 420ms, which is a consequence of the fact that many polling processes share the same CPU core, and is definitely not acceptable. Hence, this solution neither provides isolation among Workers (due to the zero-copy), nor acceptable performance when the number of Workers exceeds the number of available cores, being inappropriate for our objectives.

5.6.4 Single chain - Other tests

Additional tests have been performed in order to evaluate some other aspects of the system.

Threads vs. Processes

Threads appear more convenient than processes because they share the same virtual memory space, while processes, instead, have distinct virtual memory spaces. In our system, where the data exchange mechanism requires a shared memory between the Master and a Worker, this could have an impact on both the caching efficiency and the TLB behavior and, consequently, on the overall performance of the system. With respect to the former, two processes sharing the same physical memory address have two virtual addresses, which requires two entries in the L1/L2 caches[4]; threads, instead, have the same virtual address, hence potentially allowing the same cache line to be used by different threads. With respect to the latter, having multiple threads that share the same memory space facilitates the work of the TLB as the same (virtual) address space is present in many threads, and then the number of entries in the TLB is reduced. Instead, processes are expected to generate an higher number of TLB misses.

In order to guarantee memory isolation among Workers, which is a key point in a multi-tenant network node, the Master and all the Workers are implemented as different processes, which suggests a possible performance penalty compared to the thread-based implementation. However, our experiments dismantle this belief as the overall performance is definitely similar in both cases. The reason is that the L1/L2 caches are private per each physical core, but the Master and the Workers are usually executed in different cores. Hence, an address already cached by the core executing the Master cannot be already found in the cache of the core executing the Worker, forcing the latter to retrieve that data from the (physically addressed) L3 cache, no matter whether it is a thread or a process. As a consequence, as far as performance is concerned, our system shows no differences between a thread-based and a process-based implementation.

Normal memory vs. huge pages

We also evaluated the impact of our choice of using huge pages (each one consisting of 2MB of memory in our testbed) instead of normal pages (4KB) for the shared buffers. Although it may sound strange, results of the two approaches do not differ significantly in the test scenarios considered so far. This is a consequence of our specific test conditions, where the Master and the Workers use a very little amount of memory in addition to the shared buffers. Hence, we repeated the test with Workers executing a deep packet inspection algorithm based on a Deterministic Finite Automata (DFA), which requires a huge amount of memory to keep the DFA used to recognize the given patterns into the packets. In this case, the adoption of the huge pages for the shared buffer results in roughly a 10% improvement in terms of throughput.

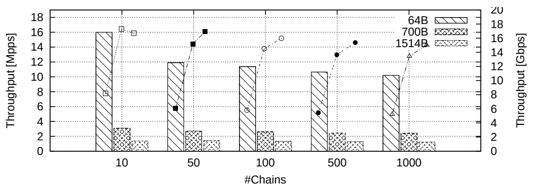
5.6.5 Multiple chains

While previous tests focused on packets traversing a growing number of VNFs all belonging to the same chain, this section evaluates the case in which multiple function chains are executed in parallel and each packet traverses only one of them. This

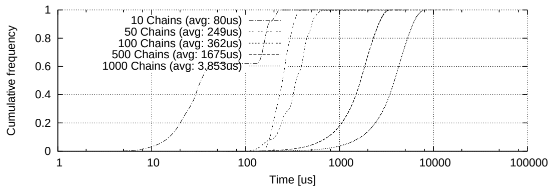
[4]The L3 cache operates with physical addresses.

significantly stresses the CPU cache, as *(i)* the Master has to receive packets from an high number of buffers, and *(ii)* the packets read by the Master are likely to be copied in different buffers for the next processing step.

Data read from the initial memory buffer (containing 1M packets) is provided, in a round robin fashion, to a growing number of function chains, each one composed of two Workers. During the tests, each Worker is involved in two chains meaning that, when 1000 Workers are deployed, packets are spread across 1000 different function chains. Workers are allocated among six CPU cores in a way that minimizes the number of times a packet has to be moved from one core to another, in order to limit CPU cache synchronization operations among cores (Section 5.5).



(a) Throughput.



(b) Latency.

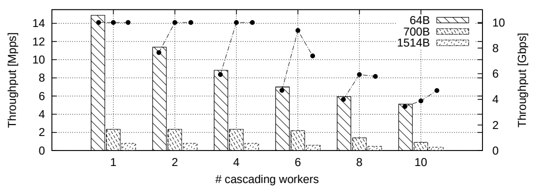
Figure 5.9: Results with a growing number of function chains running in parallel, each one spotting two Workers in cascade.

Figure 5.9(a) provides the overall throughput measured at the end of all the chains, which smoothly decreases with the increment of the number of chains; particularly, it is equal to several Gbps also with 1000 chains in the system, thus confirming the effectiveness of our algorithm. Figure 5.9(b) shows instead the cumulative distribution of the latency experienced by 64B packets traversing the chains, which ranges from an average value of 80μs in case of 10 function chains, to an average value of 3.8ms when 1000 chains are active.

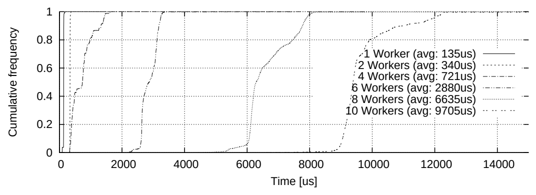
5.6.6 Network tests

This section evaluates our algorithm in a real deployment scenario, i.e., a workstation that receives/sends traffic from the network. In this case the overall performance of the system depends on the algorithm as well as additional aspects such as the driver used for accessing the NIC; anyway, these results provide an insight of the behavior of the algorithm when used in the context it was designed for.

The throughput obtained in this scenario, whose testing conditions are the same as those of Figure 5.5(d), is depicted in Figure 5.10(a). Results are limited by the speed of the input NIC in several cases, particularly with large packets and (relatively) short chains. With longer chains (i.e., 10 cascading workers) the throughput



(a) Throughput.



(b) Latency.

Figure 5.10: Results with a function chain of growing length, with the Master accessing to the network.

is even slightly better than what was obtained in Figure 5.5(d) without the network. This can be due to the fact that real NICs create an input buffer that is much smaller than the 1M packets buffer used in the previous test, hence potentially improving the data locality.

With respect to latency, Figure 5.10(b) shows the cumulative distribution function of the latency introduced by network function chains of different length when traversed by 64B packets. Those numbers measure all the time between the instant at which the packet is scheduled for transmission in the traffic generator, and the time it is received by our testing software in the traffic receiver. In this case we consider all the time spent by the packet in our middlebox, plus the network latency and the time spent in the traffic generator/receiver after/before hitting our timestamping code. Our measurements demonstrate that the latency, albeit still acceptable, is about 4-5 times higher than in Figure 5.7(a); in addition to the additional sources of delay already mentioned, we need to consider also that the packet has to spend an additional time in the input buffer before being picked up and sent through the chain by the Master, because of its batch-based reading mode.

5.7 Conclusion

This chapter proposes an efficient way to move data between virtual network functions (the Workers) and a virtual switch module (the Master), in order to implement virtual network function chains. The architecture is based on a different pair of circular buffers shared between the Master and each Worker and aims at achieving a scalable and high performance system while guaranteeing traffic isolation among the different (huge number of) Workers.

One of the peculiarities of this approach is that, through the primary buffer, data are sent to a Worker and then returned back to the Master for further processing with zero-copy. A form of batching has also been introduced in order to amortize the cost of context switches, while introducing a safeguard mechanism to avoid packet starvation in case of Workers traversed by a limited amount of traffic. The auxiliary buffer, instead, is used by the Worker to send new data to the Master.

Formal verification techniques have been applied, in order to rigorously prove the absence of deadlocks and livelocks, and also to guarantee that no packet can be accidentally overwritten due to concurrency issues, such as race conditions or incorrect use of shared indexes.

Finally, performance and scalability of the proposed solution have been evaluated by means of a wide range of experiments made on a real implementation.

Chapter 6

Transparent optimization of inter-virtual network function communication in Open vSwitch

6.1 Introduction

Network Function Virtualization (NFV) [43] transforms many network functions in software images executed on standard high-volume servers. Complex services can be delivered by rearranging multiple Virtual Network Functions (VNFs) in arbitrary chains (or *graphs*, Figure 6.1(a)), with multiple VNFs often executed on a single physical server. Usually, VNFs are instantiated as virtual machines (VMs)[1], while the traffic steering is carried out by a virtual switch (vSwitch) that classifies and forwards the packets according to specific rules sent through OpenFlow [68] messages, as shown in Figure 6.1(b).

Figure 6.1(a) shows a generic graph and contains both point-to-point (*p-2-p* in this chapter) and point-to-multipoint links. While point-to-multipoint links require a vSwitch to classify and send each packet to the proper VNF, p-2-p links, which are definitely more common in current service graphs, could be implemented by a direct communication path, hence taking the vSwitch out of that portion of the data plane. This, in turn, may result in higher throughput and lower latency, as well as in lower resource consumption thanks to the CPU saved by avoiding a further pass in the vSwitch.

Starting from this consideration, this chapter[2] proposes an architecture, called

[1]Although also lightweight containers such as Docker [3] can be used to run VNFs, they are not considered in this chapter. Then, in the following, the terms VNF and VM will be used interchangeably.

[2]The work of this chapter is partially described in the master thesis of Mauricio Vasquez Bernal,

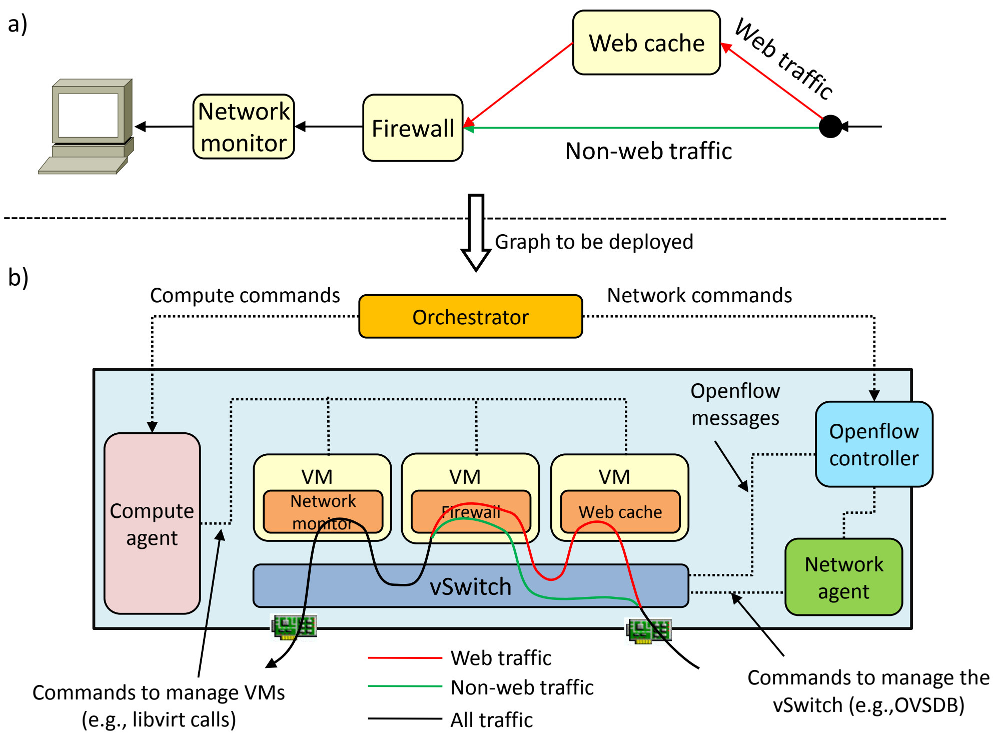


Figure 6.1: Traffic crossing several VNFs: (a) the "abstract" service graph; (b) its implementation on a server.

"*direct VM2VM*", that optimizes inter-VNF communications by creating a direct connection between two VMs, hence bypassing the vSwitch in case of p-2-p links. This architecture has capability to accelerate *transparenty* and *dynamically* the packets exchange between the VMs, and it is integrated in a *widespread* vSwitch.

Transparency refers to the possibility for an application to exploit the advantages of the Direct VM2VM technology without even knowing it is there, and an OpenFlow controller to attach to a vSwitch without noticing it has been modified. In fact, most of the extensions needed by this technology are kept in the vSwitch, with minimal modifications in few other components.

Dynamicity refers to the capability to either optimize a path or return to a traditional VM-to-vSwitch-to-VM implementation on the fly, based on the run-time analysis of the graph that is being instantiated or modified. In fact, new service requests may change the graph, either transforming a direct connection into a branch or vice versa, hence requiring the vSwitch to adapt the actual paths between VMs to the best possible implementation.

Finally, the direct VM2VM technology has been implemented in a *widespread*

who collaborated in the development of the prototype vSwitch, namely Open vSwitch (OvS) [76]; particularly, it extends the version of OvS based on the Intel Data Plane Development Kit (DPDK) [56], which exploits the optimized packet processing capabilities of that library to achieve high throughput on standard high-volume hardware. For the same reason, this chapter focuses on VMs that *execute DPDK-based network applications*; in fact, we expect that in a near future NFV applications will leverage the power of optimized libraries such as DPDK for most of the low-level packet processing tasks.

This chapter is structured as follows. Section 6.2 provides an overview of the technologies exploited in our work, while Section 6.3 presents the architecture of the prototype. Experimental results are shown in Section 6.4, while Section 6.5 analyzes the related works. Finally, Section 6.6 concludes the chapter and draws our future plans.

6.2 Background

Among the several types of ports supported by OvS, `dpdkr` is considered the fastest one. It consists of a pair of DPDK queues (`rte_rings`) that contain pointers to packets; packets are in fact stored in a piece of memory (`rte_mempool`) allocated in huge pages, shared between OvS and the entities that exploit `dpdkr` ports. Consequently, `dpdkr` ports exchange packets in a zero-copy fashion. Moreover, this port does not have any notification mechanism, and hence both ends of the port (i.e., VNF and OvS) operate in polling mode.

A `dpdkr` port can connect the vSwitch to DPDK applications executed inside VMs. Notably, OvS exports to applications the `dpdkr` port as two `rte_rings` (`RX` and `TX`); hence applications have to explicitly write/read packets to/from such rings, and do not have any concept of network interface. `rte_rings` are provided to the VM through the Inter-VM Shared Memory (`ivshmem`) technology [63], a standard interface for the KVM hypervisor [6] that is used to share memory between the host and the guest operating systems (OS). The memory region to be shared is exposed to the guest as a PCI Base Address Registers (BAR); then, applications can `mmap` [9] it into their own virtual address space.

DPDK includes a library [5] that enables the creation of `ivshmem` devices, which include also some information about the data structures mapped in the device itself, such as their virtual address in the virtual memory of OvS (Figure 6.2). This is used by DPDK in the guest OS to `mmap` the shared structures at the *same* virtual address used by OvS, which allows the application and OvS to exchange pointers to packets and de-reference them without any additional translation, which is a crucial factor in high performance environments.

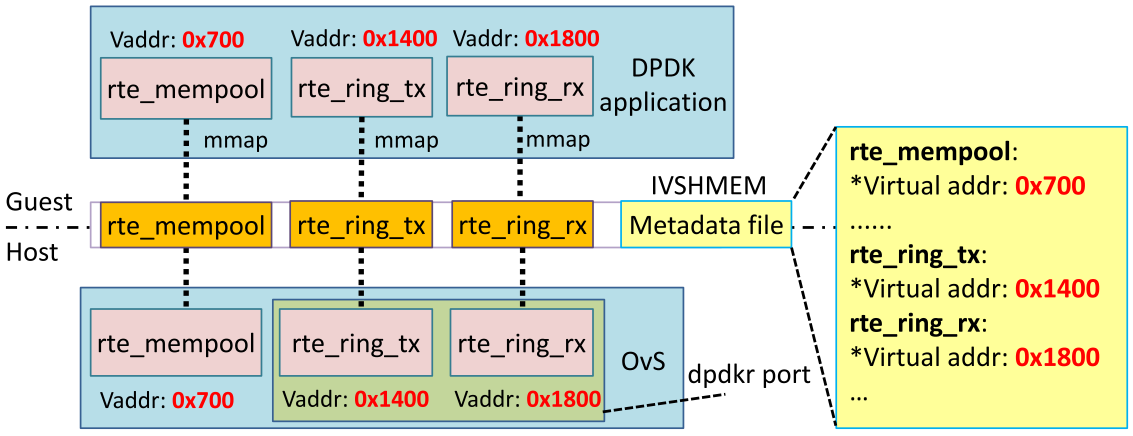


Figure 6.2: Sharing DPDK data structures between OvS and VMs.

6.3 Architecture

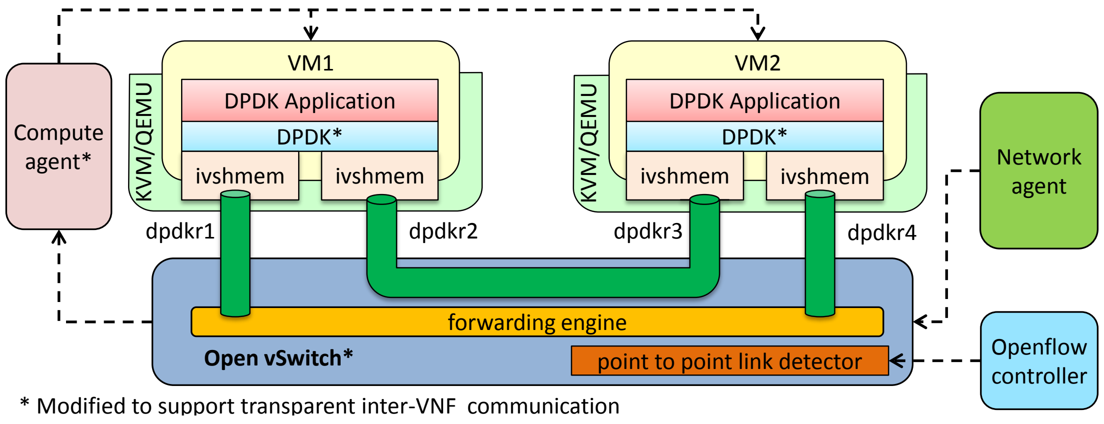


Figure 6.3: Different implementations for the `dpdkr` port.

The DPDK-based applications we consider run inside VMs that are connected to OvS through `dpdkr` ports terminated in the forwarding engine of the vSwitch. This module handles packets according to the content of its forwarding table, which can be configured with Openflow `flowmods` [68]. All the connections among VMs are implemented in this way, regardless of the nature of the connection itself, i.e., p-2-p or point to multipoint.

In our proposal, shown in Figure 6.3, p-2-p links are implemented using two modified `dpdkr` ports connected directly to each other and detached from the OvS forwarding engine. Although OvS is no longer involved in moving packets exchanged by VMs, the two modified ports are still exported by OvS as standard `dpdkr` ports. This keeps the compatibility with external entities (applications, compute/network agents, Openflow controller), as they can continue to issue commands involving those ports as they usually do (e.g., get statistics, turn them on/off, etc.), without noticing any change in their actual implementation.

6.3.1 Detecting p-2-p links

We extended OvS with a new *p-2-p link detector* module (Figure 6.3), which analyses each rule (e.g., `flowmod`) received by the vSwitch in order to dynamically detect the creation or destruction of a p-2-p link between two `dpdkr` ports. In the current implementation, this operation requires a time $O(N)$ where `N` is the number of forwarding rules installed, but this algorithm could be replaced with a more efficient version in the future.

We implemented the *p-2-p link detector* in OvS because, while OvS is the defacto vSwitch in the current NFV architectures, no standard emerged yet in case of, e.g., the orchestrator. Working in the standard component makes the integration of the direct VM2VM technology into the other components easier, and it allows to possibly reduce the number of elements to be modified.

When a new p-2-p link is detected, OvS creates two `dpdkr` ports mapped on the same piece of memory, which contains a pair of `rte_rings` and that will be shared by both the communicating VMs (the `rte_ring` used as TX in one VM, has to be used as RX in the other VM, and vice versa). This way they are directly connected, and then packets can be exchanged without the intervention of the OvS forwarding engine.

6.3.2 Handling the new `ivshmem` device

The `rte_rings` forming a `dpdkr` port are provided to the VM as part as an `ivshmem` device, together with a metadata file indicating, for each data structure, its address in the virtual memory of OvS (Section 6.2).

As shown by comparing Figure 6.2 and Figure 6.4, the metadata file included in the new `ivshmem` device has been modified to specify the virtual addresses of the *old* `rte_rings` used by OvS, and not those used for the new rings contained in the device itself. This allows DPDK to `mmap` the new `rte_rings` at the same virtual address of the old ones, so that the application can continue to work without realizing that the `rte_rings` are changed (more details in Section 6.3.3). For the same reason, we had to extend DPDK to allow applications to bind to `rte_rings` with "default" names (e.g., `dpdkr0_tx`) and dynamically remap them to the actual name used by the rings, hence avoiding the problem that those names are generated dynamically.

Now, the new `ivshmem` device has to be connected to the proper VM. Since OvS does not know which VM is attached to a specific port (it just knows ports and the rules used to forward packets among them), for this operation OvS has

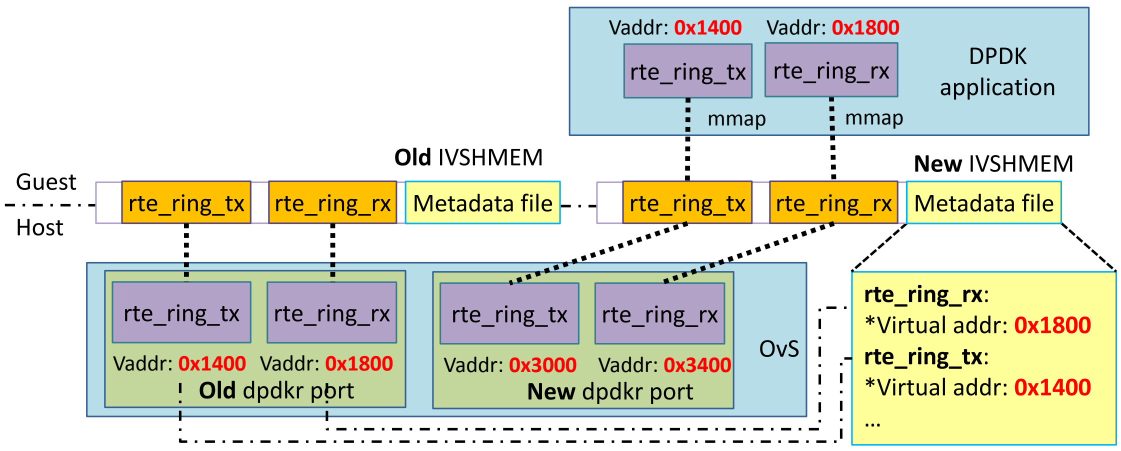


Figure 6.4: `ivshmem` device for port remapping.

to rely on an external component. In our prototype we adopted two strategies: (*i*) a meaningful message is printed on the console with an example of the proper command that can be issued (manually) to attach the new `ivshmem` device to a VM, and (*ii*) we modified the compute agent of our Universal Node orchestrator (Chapter 3) to issue this command automatically to QEMU/KVM; other solutions such as the OpenStack Nova [14] agent can be exploited as well. In our case, we modified the REST API of our compute agent with a new function that triggers the attachment of an `ivshmem` device to a VM. The compute agent is able to recognize the specific VM involved and, through the proper API (e.g., libvirt [7]), hotplugs the `ivshmem` device in it.

6.3.3 Remapping process

The *remapping process* consists in recognizing, then changing, dynamically and transparently, the pair of `rte_rings` an application is using. Then, it allows existing DPDK-based applications using `dpdkr` ports to support our technology without any modification, except for the necessity to be recompiled with our modified DPDK.

Vanilla DPDK does not recognize when a new `ivshmem` device is hotplugged in the VM. Hence, our prototype extends DPDK in order to register a handler (through the `lib_udev` library) that is executed each time a new `ivshmem` device is connected to the PCI bus. In this handler, DPDK identifies the old `rte_rings` to be removed (thanks to the virtual addresses specified in the metadata file), and marks them as "to be remapped". The remapping process is in fact not actually done in this handler, since its execution is asynchronous with respect to the application, which may be accessing the `rte_rings` to send/receive packets during the execution of the handler itself. The remapping of the `rte_rings` used by an application is done by DPDK when such an application transmits or receives packets, as shown in Algorithm 8.

Algorithm 8 Remapping process in the VM.

```
 1: procedure sendPackets (rte_ring *ring, list *pkts) {receivePackets is equivalent}
 2: if ring $\in$ toBeRemapped then
 3:     for all ring $\in$ toBeRemapped do
 4:         munmap(ring.virtualAddress)
 5:         mmap(ring.virtualAddress,ring.device)
 6:         ring.mapped $\leftarrow$ true
 7:         toBeRemapped.remove(ring)
 8:     end for
 9:     while not ring.usable do
10:         {do nothing}
11:     end while
12: end if
13: {send/receive packets as usual}
14: end procedure
```

For this reason, we extended DPDK so that, before transmitting/receiving packets on an `rte_ring`, it checks whether such a ring has to be remapped or not (line `2`). If so, according to lines `3-8`, the remapping of all the `rte_rings` contained into the new `ivshmem` device is done. This requires to unmap the old `rte_rings` and to `mmap` the corresponding new ones at the same virtual addresses just released (lines `4-5`).

In order to avoid packet loss and reordering in this transient, we defined a synchronization mechanism between DPDK in the guest and OvS, based on two flags inserted in the `rte_ring` structure: `mapped` and `usable`. As shown in line `6`, DPDK sets the former as soon as an `rte_ring` has been remapped. At this point OvS, which was blocked on such a shared flag: *(i)* copies the (pointers to) packets present in the old `RX rte_ring` into the new one; *(ii)* handles the packets already inserted by the application in the old `TX rte_ring`; *(iii)* notifies DPDK in the guest that the new `rte_ring` can be used, by means of the `usable` flag. At this point (line `9-11`) the application can finally transmit/receive the packets.

6.3.4 Port statistics

For compatibility reasons, OvS should allow to get statistics on all the `dpdkr` ports, regardless of their actual implementation. While OvS counts the number of bytes/-packets flowing through standard `dpdkr` ports while forwarding them, this is not possible in case of p-2-p link, since the vSwitch no longer accesses to packets transmitted on such a connection. Then, we extended *(i)* DPDK so that, each time a packet is sent, the code automatically updates the counters stored into the used `rte_ring`, and *(ii)* the `rte_ring` structure so that it contains statistics related to the ring itself. Then, when OvS needs to export statistic of a `dpdkr` port forming a direct connection, it just reads them from the proper `rte_ring` structure.

6.4 Experimental validation

We characterized our prototype [93] on an Intel Xeon E5-2690 v2 @ 3 GHz (ten physical cores plus hyperthreading), 64 GB RAM, two 10G Intel 82599ES NICs, Ubuntu 15.04, kernel 3.19.0-15-generic, 64 bits. Our code is based on OvS `2.4.9` and DPDK `2.1.0`. We compare our solution with traditional connections implemented through the forwarding engine of (vanilla) OvS, both from the point of view of the maximum throughput achieved and the latency introduced by service chains traversed by 64B packets. Finally, we report the time required by our prototype to detect a p-2-p link and create the direct path between the two VMs involved.

In all the tests, we consider chains of VMs connected only through p-2-p links. We generate bidirectional traffic at the maximum speed, 64B packets, with a maximum theoretical throughput of 20Gbps when the traffic is sent through physical

NICs. Unless otherwise specified, each VM uses a single CPU core, has two ports and runs a DPDK application that simply moves packets from one port to another. Thanks to the transparency of the direct VM2VM technology, the same forwarding VM has been used unchanged in all tests.

6.4.1 Throughput with internal traffic

This test exploits a variable number of VMs running the DPDK packet forwarder connected with two VMs acting as traffic source and sink, as depicted in Figure 6.5. This test validates our approach in isolation, without the overhead introduced by accessing to the physical NIC and by the PCIe bus.

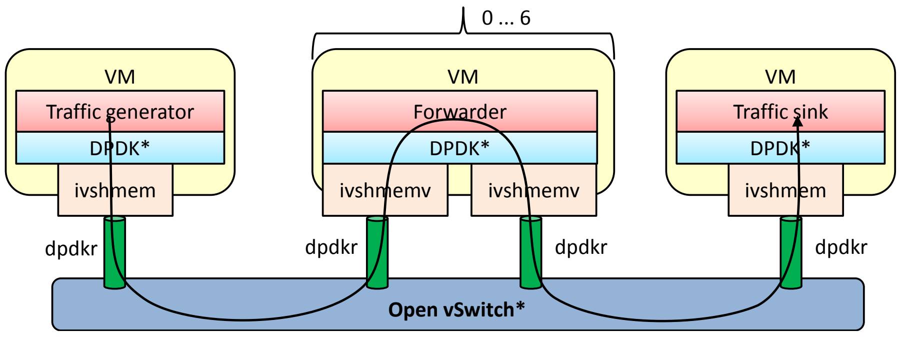


Figure 6.5: Test setup.

Figure 6.6 shows that the performance of our prototype is more than one order of magnitude higher compared to the traditional approach. Furthermore, the throughput remains almost constant when new intermediate VMs are added, unless we exhaust the number of available CPU cores. However, this test was biased by a bug in OvS that caused an impressive drop of the throughput [15] when providing more than one CPU core to OvS; therefore, OvS was executed on a single CPU core.

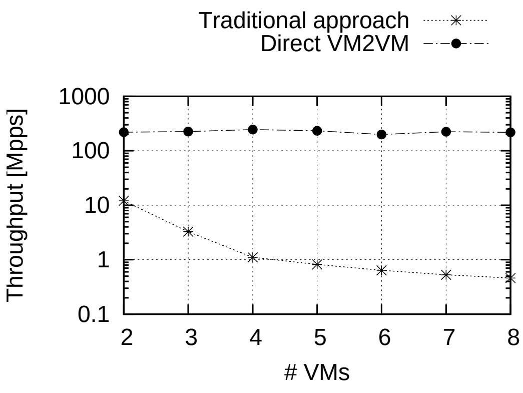


Figure 6.6: Memory-only traffic.

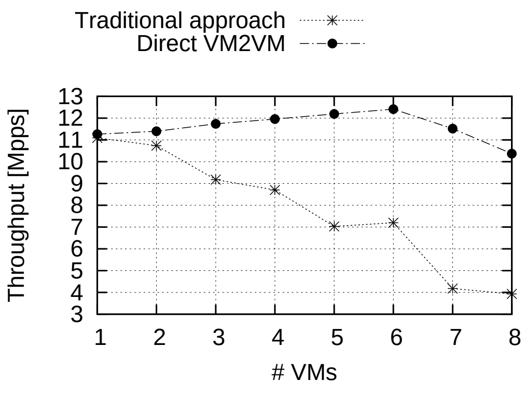


Figure 6.7: Traffic using physical NICs.

6.4.2 Throughput with physical NICs

In this test we kept only the forwarding VMs on the server running OvS, while the traffic to the source/sink VMs is delivered through a couple of 10Gps links. This scenario validates our approach when packets are received from the network, as well as it shows how the throughput changes according to the number of cores assigned to OvS without the impact of the bug mentioned above.

Since DPDK-based OvS operates in polling mode, some CPU cores are dedicated to the vSwitch itself. Particularly, OvS works at best when each port (either physical or virtual) is associated with a specific core, as each receiving queue can be associated with a dedicated thread that takes care of handling the packet until transmission. In our test we assigned to OvS all the cores it needs until we run into cores exhaustion. Core allocation is shown in Figure 6.8, together with the number of cores actually used to get the results depicted in Figure 6.7.

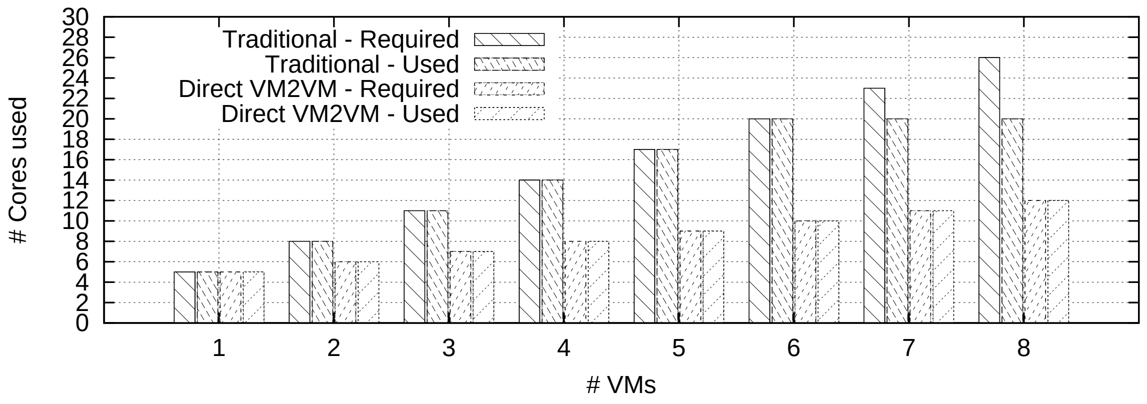


Figure 6.8: # of cores required/assigned during the tests.

Tests show that a chain of VMs exploiting our direct VM2VM provides better throughput and require less CPU cores than the same chain implemented using the traditional approach. Moreover, the throughput is almost constant when direct VM2VM is used, while it presents mostly a decreasing trend when all the connections are implemented through the forwarding engine of OvS[3].

6.4.3 Latency

Results in Figure 6.9 show that the latency of a chain of VMs (using the same testbed of Section 6.4.2, but tuning the TX speed in order not to have losses in the chain) is almost constant until three chained VMs, in both the traditional and

[3]In case of single VM, direct VM2VM cannot be exploited, then the throughput is the same obtained with the traditional approach.

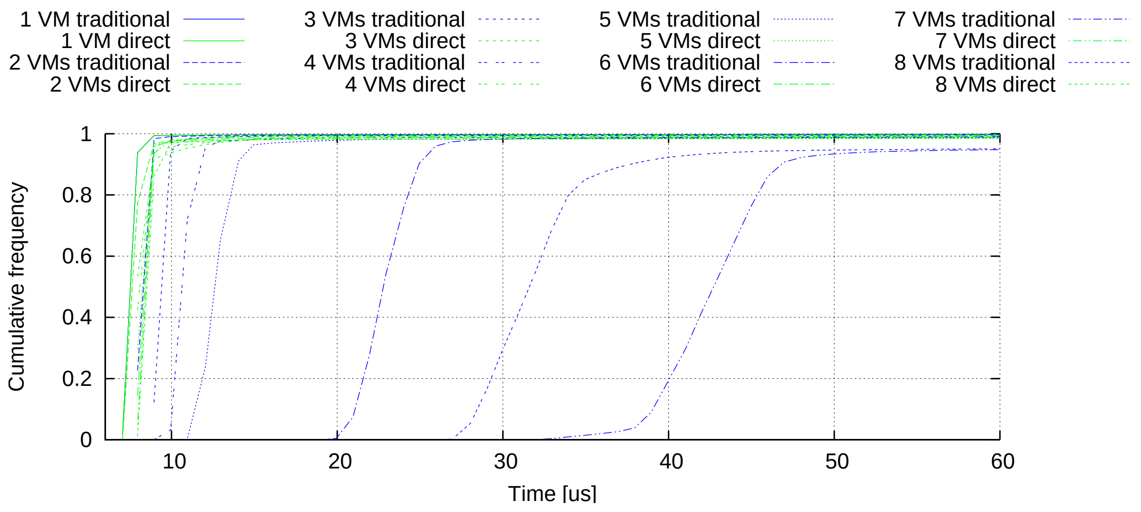


Figure 6.9: Latency when physical NICs are involved.

VM2VM case. Instead, vanilla OvS presents higher latency with longer chains. This confirms that our approach is always better in terms of latency and it scales also better with the number of chained VMs.

6.4.4 Establishment time

Figure 6.10 reports the time needed to establish a direct channel between two VMs, from the moment in which OvS receives a new rule (`flowmod`) that triggers the creation of a p-2-p link, to the moment in which the forwarding application starts to use the new direct `dpdkr` port. This time depends on different components such as DPDK, OvS, compute agent, guest OS and the QEMU/KVM hypervisor. Results show that the (by far) dominant contributors are OS-level components, namely the time needed by QEMU/KVM to plug the `ivshmem` device and the guest OS to recognize it. Instead, the weight of the p-2-p link detector module is negligible, questioning the necessity of a more optimized algorithm.

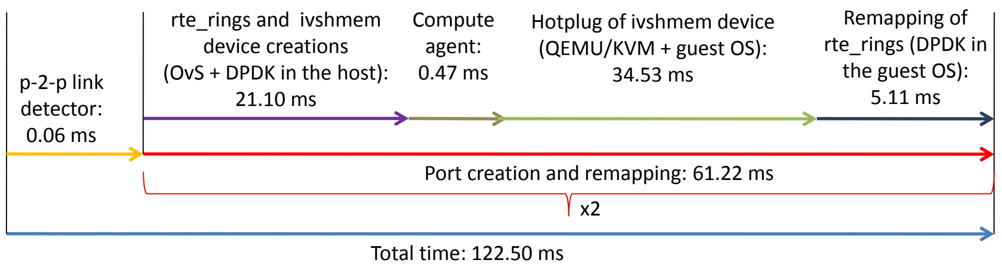

p-2-p link detector: 0.06 ms
rte_rings and ivshmem device creations (OvS + DPDK in the host): 21.10 ms
Compute agent: 0.47 ms
Hotplug of ivshmem device (QEMU/KVM + guest OS): 34.53 ms
Remapping of rte_rings (DPDK in the guest OS): 5.11 ms
Port creation and remapping: 61.22 ms
x2
Total time: 122.50 ms

Figure 6.10: Time required to establish a direct connection.

6.5 Related work

Several works aim at optimizing the communication between VMs executed on the same server, often through the creation of a direct channel between such VMs.

Intel [73, 72] is working on a traffic bypass mechanism based on Virtio [84] that targets mainly non-DPDK applications handling packets through (virtual) NICs.

`ptnetmap` [45] allows netmap-based applications [81] (running in VMs) to transparently use different type of ports; hence they can be connected to physical NICs, to the VALE [82] vSwitch or to a netmap pipe, which is a direct channel between two netmap-based applications.

XenSocket [97] defines a new socket that exploits a shared memory to bypass the network stack, but requires applications to be modified; XWay [58] gets the same result by modifying the internals of the TCP stack and hence supports unmodified applications. This applies also to XenLoop [94], which intercepts packets at the network layer and can send them via shared memory.

Other works [54, 38] focus on High Performance Computing and propose libraries that implement a message-passing paradigm based on memory shared between VMs residing on the same server. Finally, ClickOS [64] (based on VALE and Click [71]), NetVM [55] (based on DPDK) and the shared buffer presented in Chapter 5 optimize the communication between the VMs and the vSwitch.

Although these proposals present similarities with the work presented in this chapter (e.g., most of them create a direct channel between VMs; some are also transparent to applications), important differences exist. Most notably, only our direct VM2VM takes an holistic approach that transparently accelerates DPDK applications using the most common technologies currently used in the NFV world, while retaining the compatibility with the entire ecosystem including SDN controllers.

6.6 Conclusion

This chapter proposes an architecture that is able to optimize inter-VNF communications by bypassing the vSwitch in case of p-2-p connections between VMs. Our architecture can accelerate the packets exchange between the VMs *transparently*, i.e., without modifying the applications and by keeping the compatibility with all the services (e.g., SDN controller) deployed in an NFV environment, and *dynamically*, i.e., it can optimize direct paths when those are detected and revert back to the traditional VM-to-switch-to-VM communication when the optimization is no longer possible. Our extensions have been integrated in a *widespread* vSwitch, bringing the advantages of this technology to a broad set of use cases.

Our tests confirm the goodness of the approach and the possibility to implement this idea by touching a limited number of components, namely OvS, DPDK and (optionally) the compute agent. Future work will explore the possibility to extend the usage of direct paths also when accessing to physical NICs, e.g., through SR-IOV.

Conclusions

This dissertation introduces a number of improvements in the context of Network Functions Virtualization (NFV), both in terms of models and software architectures that enable a multitude of different players (e.g., end users, network operators, etc.) to instantiate their own network services, and in terms of mechanisms used to efficiently exchange packets among the many VNFs instantiated on the same physical server.

The first contribution of the dissertation is FROG, a programmable edge node that gives to the end users directly connected to it (and to other players such as content providers as well), the possibility to deploy their VNFs operating on a portion of the network traffic flowing through the node. In order to scale with the number of end users that may be connected to it, FROG provides to each player a lightweight execution environment (the PEX) running all the VNFs installed by the player himself, and which just operates on the traffic belonging to such a player. In the FROG service model, the PEX of an end user is the first one processing the traffic coming from the end user device, and the last one that handles packets towards the end user terminal; hence, the PEX can be seen as an extension of the TCP/IP stack of the user device moved into the network. Experimental results show that a model that allocates a PEX per user, and which supports until 8000 of active users (each one running their VNFs) at the same time on the same physical machine is feasible.

The dissertation then presents the *service graph*, a new formalism to model generic network services; notably, the service graph definition is completely compliant with the NFV principles of abstract description of a service, but enriches its traditional expressiveness to model legacy networks and services as well (e.g., a DHCP server connected to a LAN). The service graph is transformed in the *forwarding graph* by means of the "lowering process", which leads to the deployment of an optimized service on the operator network. This translation process is in fact capable to adapt the service delivering to available resources of the underlying infrastructure. In order to validate the formalisms, a multilayer network orchestration architecture is then introduced. Such an architecture, starting from a service graph that can be defined by multiple players (e.g., end users, telecom operator), takes care of instantiating it on the physical infrastructure of the network, by exploiting the opportunities offered by both the NFV and Software Defined Networking (SDN) paradigms. Two prototypes of nodes representing the physical infrastructure have then been defined. The first one is the Universal Node, which consists of a single server mainly based on ad hoc components; the latter, called OpenStack-based Node, is instead implemented as a cluster of servers orchestrated by an extended version of the OpenStack framework. Experimental results show that, while the Universal Node has low requirements in terms of memory, its performance are worse than the OpenStack-based Node in almost all the tests carried out.

Moving to the problem of improving performance of virtualized services, this dissertation proposes a number of communication architectures to be used to exchange packets between the virtual switch and the VNFs instantiated on the same physical server. Such communication architectures are designed by considering the typical traffic pattern in NFV, and then they are optimized to work in this context. Among the others, the most relevant proposal is based on a different pair of (lock-free) ring buffers shared between the virtual switch and each VNF, and aims at achieving high performance while at the same time guaranteeing traffic isolation among the (huge number) of VNFs. One of the peculiarity of such an approach is that, through the primary buffer, packets are sent to the VNF and then returned back to the virtual switch for further processing with zero-copy. A form of batching has also been introduced in order to amortize the cost of context switches, while a safeguard mechanism avoids packet starvation in case of VNFs traversed by a limited amount of traffic. Formal verification techniques have been applied, in order to rigorously prove properties such as the absence of deadlocks, livelocks and more. A wide range of experimental tests executed on a prototype designed to characterize the behavior of the algorithm in different conditions provides promising results.

The last contribution of this dissertation is the definition of an architecture that optimizes the inter-VNF communication by bypassing the virtual switch in case of point to point connections between virtual machines (executing applications based on the Intel DPDK framework). In particular, the proposed approach does not require any modification to the VNFs, and keeps compatibility with all the services deployed in an NFV environment, such as the SDN controller. Tests confirm the validity of the idea and the possibility of implementing it by modifying a limited number of components, namely the virtual switch, the DPDK framework, and optionally the compute agent.

Bibliography

[1] Cisco systems. http://homesupport.cisco.com/en-us/support/ccc/PARENTALCONTROLS.

[2] Davide.it. http://www.davide.it/.

[3] Docker. https://www.docker.com/.

[4] Introduction to control groups. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html.

[5] Ivshmem library. http://dpdk.org/doc/guides/prog_guide/ivshmem_lib.html.

[6] Kvm. http://www.linux-kvm.org.

[7] Libvirt. http://libvirt.org/.

[8] Linux containers. https://linuxcontainers.org/.

[9] mmap. http://man7.org/linux/man-pages/man2/mmap.2.html.

[10] Namespaces overwiew. http://lwn.net/Articles/531114/.

[11] Opendaylight. http://www.opendaylight.org/.

[12] Opendns. http://www.opendns.com/parental-controls.

[13] Openstack. http://www.openstack.org/.

[14] Openstack nova. http://docs.openstack.org/developer/nova/.

[15] Performance issue when using netdev-dpdk and multiple pmd threads. http://openvswitch.org/pipermail/dev/2015-December/063885.html.

[16] The SECURED project (SECURity at the network EDge)). http://www.secured-fp7.eu/.

[17] T-nova: Network functions as a service over virtualised infrastructure. http://www.t-nova.eu/.

[18] Unify: unifying cloud and carrier network. http://www.fp7-unify.eu/.

[19] ndpi. http://www.ntop.org/products/ndpi/, Apr 2012.

[20] Openstack blueprints: Neutron service chaining specification. https://review.openstack.org/#/c/93524/, 2014.

[21] Openstack blueprints: Neutron services insertion, chaining, and steering. https://blueprints.launchpad.net/neutron/, 2014.

[22] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin

Vahdat. xomb: extensible open middleboxes with commodity servers. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 49–60, New York, NY, USA, 2012. ACM.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[24] Daniel Barron. Dansguardian. http://dansguardian.org, Aug 2011. Stable release 2.12.0.0.

[25] Berlin Institute for Software Defined Network and Deutsche Telekom Innovation Labs. Your home in your pocket. https://www.opennetworking.org/images/stories/sdn-solution-showcase/BISDN-demo.pdf.

[26] BISDN. The extensible openflow datapath daemon. http://www.xdpd.org.

[27] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper: Software-defined network service chaining. In *Proceedings of the Third European Workshop on Software Defined Networking (EWSDN 2014)*, 2014.

[28] R. Bonafiglia, I. Cerrato, F. Ciaccia, M. Nemirovsky, and F. Risso. Assessing the performance of virtualization technologies for nfv: A preliminary benchmarking. In *2015 Fourth European Workshop on Software Defined Networks*, pages 67–72, Sept 2015.

[29] Zvika Bronstein and Eyal Shraga. Nfv virtualisation of the home environment. In *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, pages 899–904. IEEE, 2014.

[30] I. Cerrato, G. Marchetto, F. Risso, R. Sisto, and M. Virgilio. An efficient data exchange algorithm for chained network functions. In *2014 IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, pages 98–105, July 2014.

[31] I. Cerrato, M. Pramotton, and F. Risso. Moving applications from the host to the network: Experiences, challenges and findings. In *2013 IEEE International Conference on Communications Workshops (ICC)*, pages 744–749, June 2013.

[32] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. Supporting fine-grained network functions through intel dpdk. In *EWSDN*, pages 1–6. IEEE Computer Society, 2014.

[33] Ivano Cerrato, Tobias Jungel, Alex Palesandro, Fulvio Risso, Marc Suñé, and Hagen Woesner. User-specific network service functions in an sdn-enabled network node. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 135–136. IEEE, 2014.

[34] Ivano Cerrato, Guido Marchetto, Fulvio Risso, Riccardo Sisto, and Matteo Virgilio. Shared buffer model. https://github.com/netgroup-polito/shared-buffer.

[35] Ivano Cerrato, Alex Palesandro, Fulvio Risso, Marc Suñé, Vinicio Vercellone, and Hagen Woesner. Toward dynamic virtualized network services in telecom operator networks. *Computer Networks*, 92, Part 2:380 – 395, 2015. Software Defined Networks and Virtualization.

[36] ContentWatch. Net nanny. `http://www.netnanny.com`.

[37] András Császár, Wolfgang John, Mario Kind, Catalin Meirosu, Gergely Pongrácz, Dimitri Staessens, Attila Takács, and Fritz-Joachim Westphal. Unifying cloud and carrier network: Eu fp7 project unify. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC '13)*, UCC '13, pages 452–457, Washington, DC, USA, 2013. IEEE Computer Society.

[38] François Diakhaté, Marc Perache, Raymond Namyst, and Herve Jourdren. Efficient shared memory message passing for inter-vm communications. In *Euro-Par 2008 Workshops-Parallel Processing*, pages 53–62. Springer, 2008.

[39] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *Formal Techniques for Networked and Distributed Systems - FORTE 2004*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer Berlin Heidelberg, 2004.

[40] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *SIGOPS Oper. Syst. Rev.*, 35(4):50–60, October 2001.

[41] European Telecommunication Standards Institute (ETSI). Network Functions Virtualization Industry Specification Group. `http://portal.etsi.org/NFV`.

[42] European Telecommunication Standards Institute (ETSI). Network Functions Virtualization; use cases. `http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf`.

[43] European Telecommunications Standards Institute. Network Functions Virtualisation. White paper, SDN and OpenFlow World Congress, Darmstadt, Germany, Oct. 2012.

[44] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 218–224, New York, NY, USA, 2010. ACM.

[45] Stefano Garzarella, Giuseppe Lettieri, and Luigi Rizzo. Virtual device passthrough for high speed vm networking. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 99–110. IEEE, 2015.

[46] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 302–317, Berlin, Heidelberg, 2010. Springer-Verlag.

[47] Google. Google safe search. http://support.google.com/websearch/bin/answer.py?hl=en&answer=510.

[48] Sriram Govindan, Jeonghwan Choi, Arjun R Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms. *Computers, IEEE Transactions on*, 58(8):1111–1125, Aug 2009.

[49] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010.

[50] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In Eduardo Tovar, Philippas Tsigas, and Hacene Fouchal, editors, *OPODIS*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007.

[51] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual.* Addison-Wesley Professional, first edition, 2003.

[52] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, May 1997.

[53] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[54] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, page 9, 2007.

[55] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, 2014. USENIX Association.

[56] Intel. Intel dpdk - programmer's guide. http://dpdk.org/doc/guides/prog_guide/, 2014.

[57] Internet Engineering Task Force (IETF). Service Functions Chaining (SFC) working group. https://datatracker.ietf.org/wg/sfc/documents/, 2014.

[58] Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 11–20, New York, NY, USA, 2008. ACM.

[59] Patrick P. C. Lee, Tian Bu, and Girish P. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *IPDPS*, pages 1–12, 2010.

[60] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[61] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. Serverswitch: a programmable and high performance platform for data center networks. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[62] Francesco Lucrezia, Guido Marchetto, Fulvio Risso, and Vinicio Vercellone. Introducing network-aware scheduling capabilities in openstack. In *Proceedings of the First IEEE Conference on Network Softwarization (Netsoft 2015)*, Apr 2015.

[63] Cam Macdonell. Nahanni, a shared memory interface for kvm. `http://www.linux-kvm.org/images/e/e8/0.11.Nahanni-CamMacdonell.pdf`.

[64] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, 2014. USENIX Association.

[65] H. Massalin and C. Pu. Threads and input/output in the synthesis kernal. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 191–201, New York, NY, USA, 1989. ACM.

[66] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):108–, April 1992.

[67] McAfee. safeeyes. `http://www.internetsafety.com/safe-eyes-parental-control-software-affiliate.php`.

[68] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[69] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[70] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, August 1997.

[71] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 217–231, New York, NY, USA,

1999. ACM.

[72] Jun Nakajima, Mesut Ergin, Yunhong Jiang, Krishna Murthy, James Tsai, Wei Wang, Huawei Xie, and Yang Zhang. KVM as the NFV hypervisor, kvm forum 2015. http://events.linuxfoundation.org/sites/events/files/slides/Jun_Nakajima_NFV_KVM%202015_final.pdf.

[73] Jun Nakajima, James Tsai, Mesut Ergin, Yang Zhang, and Wei Wang. Extending KVM models towards high-performance NFV, kvm forum 2014. http://www.linux-kvm.org/images/1/1d/01x05-NFV.pdf.

[74] Netgear. Live parental controls. http://www.netgear.com/lpc.

[75] OPNFV. OPNFV - an open platform to accelerate nfv. http://www.opnfv.org, 2015.

[76] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, October 2009.

[77] S. Prakash, Yann Hang Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput.*, 43(5):548–559, May 1994.

[78] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 13–24, Berkeley, CA, USA, 2013. USENIX Association.

[79] F. Risso and I. Cerrato. Customizing data-plane processing in edge routers. In *2012 European Workshop on Software Defined Networking*, pages 114–120, Oct 2012.

[80] F. Risso, A. Manzalini, and M. Nemirovsky. Some controversial opinions on software-defined data plane services. In *Proceedings of the First Workshop on Software Defined Networks for Future Network Services (SDN4FNS)*, SDN4FNS13, pages 1–7. IEEE, November 2013.

[81] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[82] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[83] Julius Rückert, Jeremias Blendin, Nicolai Leymann, Georg Schyguda, and David Hausheer. Software-defined network service chaining. In *Proceedings of the Third European Workshop on Software Defined Networking (EWSDN 2014)*, 2014.

[84] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices.

SIGOPS Oper. Syst. Rev., 42(5):95–103, July 2008.

[85] Fernando Sanchez and David Brazewell. Tethered linux cpe for ip service delivery. In *Proceedings of the First IEEE Conference on Network Softwarization (NetSoft 2015)*, 2015.

[86] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, 2012.

[87] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *SIGCOMM Comput. Commun. Rev.*, 42(4):13–24, August 2012.

[88] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[89] Pontus Sköldström, Balázs Sonkoly, Andrś Gulyás, Felicián Németh, Mario Kind, Fritz-Joachim Westphal, Wolfgang John, Jokin Garay, Eduardo Jacob, Dávid Jocha, János Elek, Róbert Szabó, Wouter Tavernier, George Agapiou, Antonio Manzalini, Matthias Rost, Nadi Sarrar, and Stefan Schmid. Towards unified programmability of cloud and carrier infrastructure. In *Proceedings of the Third European Workshop on Software Defined Networking (EWSDN)*, pages 55–60, 2014.

[90] Joao Soares, Miguel Dias, Jorge Carapinha, Bruno Parreira, and Susana Sargento. Cloud4nfv: A platform for virtual network functions. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 288–293. IEEE, 2014.

[91] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 134–143, New York, NY, USA, 2001. ACM.

[92] Unify Consortium. D5.2 universal node interfaces and software architecture. `https://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables/UNIFY-WP5-D5.2-Universal%20node%20interfaces%20and%20software%20architecture.pdf`, 2014.

[93] Mauricio Vasquez, Ivano Cerrato, and Fulvio Risso. Direct VM2VM optimzation in OvS for DPDK enabled applications. `https://github.com/netgroup-polito/directvm2vm`.

[94] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. Xenloop: A transparent high performance inter-vm network loopback. In *Proceedings of the 17th*

International Symposium on High Performance Distributed Computing, HPDC '08, pages 109–118, New York, NY, USA, 2008. ACM.

[95] Jon Whiteaker, Fabian Schneider, Renata Teixeira, Christophe Diot, Augustin Soule, Fabio Picconi, and Martin May. Expanding home services with advanced gateways. *SIGCOMM Comput. Commun. Rev.*, 42(5):37–43, September 2012.

[96] Yiannis Yiakoumis, Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Slicing home networks. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Home Networks*, HomeNets '11, pages 1–6, New York, NY, USA, 2011. ACM.

[97] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 184–203, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[98] Li Zhao, L.N. Bhuyan, R. Iyer, S. Makineni, and D. Newell. Hardware support for accelerating data movement in server platform. *Computers, IEEE Transactions on*, 56(6):740–753, June 2007.

Appendix
Author publications

Part of the research presented in this dissertation has been previously published in the following papers:

- I. Cerrato, A. Palesandro, F. Risso, M. Sune, V. Vercellone, H. Woesner, "Towards Dynamic Virtualized Network Services in Telecom Operator Networks", in *Elsevier Computer Networks Vol. 92, Part 2*, 9 December 2015. pp. 380-395
- R. Bonafiglia, I. Cerrato, F. Ciaccia, M. Nemirovsky, F. Risso, "Assessing the Performance of Virtualization Technologies for NFV: A Preliminary Benchmarking", in *Proceedings of Fourth European Workshop on Software Defined Networks (EWSDN 2015)*, Bilbao, Spain, 30 Sept. - 2 Oct. 2015. pp. 67-72
- I. Cerrato, A. Palesandro, F. Risso, T. Jungel, M. Sune, H. Woesner, "User-specific Network Service Functions in an SDN-enabled Network Node", in *Proceedings of Third European Workshop on Software Defined Networks (EWSDN 2014)*, Budapest, Hungary, September 2014. pp. 135-136
- I. Cerrato, M. Annarumma, F. Risso, "Supporting Fine-Grained Network Functions through Intel DPDK", in *Proceedings of Third European Workshop on Software Defined Networks (EWSDN 2014)*, Budapest, Hungary, September 2014. pp. 1-6
- I. Cerrato, G. Marchetto, F. Risso, R. Sisto, M. Virgilio M, "*An Efficient Data Exchange Algorithm for Chained Network Functions*", in Proceedings of IEEE 15th International Conference on High Performance Switching and Routing (HPSR 2014), Vancouver, BC, Canada, July 2014. pp. 98-105
- I. Cerrato, M. Pramotton, F. Risso, "Moving Applications from the Host to the Network: Experiences, Challenges and Findings", in *Proceedings of IEEE International Conference on Communications 2013 (IEEE ICC'13) - 1st International Workshop on Mobile Cloud Networking and Services (MCN)*, Budapest, Hungary, June 2013, pp. 744-749

- F. Risso, I. Cerrato, “Customizing Data-plane Processing in Edge Routers”, in *Proceedings of the European Workshop on Software Defined Networks (EWSDN 2012)*, Darmstadt, Germany, October 25-26, 2012, pp. 114-120