On the Design of a Flexible Architecture for Virtualized Network Function Platforms

Vinícius Fulber Garcia*, Leonardo da C. Marcuzzo†, Alexandre Huff*§, Lucas Bondan‡, Jéferson C. Nobre‡, Alberto Schaeffer-Filho‡, Carlos R. P. dos Santos†, Lisandro Z. Granville‡, Elias P. Duarte Junior*

*Federal University of Paraná
{vfgarcia,ahuff,elias}@inf.ufpr.br

†Federal University of Santa Maria
{lmarcuzzo,csantos}@inf.ufsm.br

‡Federal University of Rio Grande do Sul
{lbondan,jcnobre,alberto,granville}@inf.ufrgs.br

§Federal Technological University of Paraná
alexandrehuff@utfpr.edu.br

The proper execution and management of heterogeneous Virtualized Network Functions (VNFs) relies on the employment of efficient and comprehensive VNF platforms. However, current systems are developed without following any standardized reference architecture, thus leading to proprietary and monolithic solutions. Furthermore, those platforms lack support for recent NFV developements, such as VNF Components (VNFC) and the Network Service Header (NSH). In this work, we present an architecture for VNF platforms that is fully compliant with the European Telecommunications Standards Institute (ETSI) NFV architecture, while also enabling the execution of both VNFC and NSH. Through the development of a system prototype called COmprehensive VirtualizEd NF (COVEN) platform, we were able to evaluate the effectiveness of our proposed architecture and to demonstrate the benefits of supporting VNFC and NSH, such as flexibility and efficiency.

I. Introduction

Network Functions Virtualization (NFV) is driving a paradigm shift in telecommunications by introducing a software plane in the core network. This paradigm employs current virtualization techniques to perform different network functions. By moving the network traffic processing from dedicated and proprietary physical appliances to commercial off-the-shelf servers, the NFV paradigm enhances the flexibility and scalability of network services while reducing the CApital and OPerational EXpenditure (CAPEX and OPEX) [1].

To standardize the NFV paradigm, specifications, models, and NFV enablers (systems used to support the execution of Virtualized Network Functions - VNF) are being developed. Standards and recommendations are being specified by organizations such as the European Telecommunications Standards Institute (ETSI) and the Internet Engineering Task Force (IETF), as well as several working groups (*e.g.*, ETSI NFV Industry Specification Groups, IRTF NFV Research Group, and IETF Service Function Chaining). The ultimate goal of these efforts is to encourage the development of solid foundations that support advanced NFV solutions.

The main element of the NFV paradigm is called Virtualized Network Function (VNF), which is responsible for effectively processing the network traffic. A VNF, in turn, can be decomposed in two parts [2]: the VNF platform and the Network Function (NF). In particular, VNF platforms provide the proper environment and required resources for executing and managing network functions. Despite its importance, current VNF platforms (*e.g.*, Click-on-OSv [3] and ClickOS [4]) are being developed without following any standardized reference architecture, thus leading for inflexible and monolithic solutions with severe limitations to support emerging NFV features. Examples of such features are VNF Components (VNFC) [5] – individual elements consisting of some or all of the VNF functionality – and Network Service Header (NSH) – a packet header that enables the creation of dynamic service planes [6].

In this paper, we present a comprehensive architecture for VNF platforms, which is designed to be fully compliant with the ETSI requirements for NFV, while providing support for VNFCs and NSH through native modules. In this context, our main contributions are: i) the design of the core elements of a flexible architecture for VNF platforms that support VNFCs and NSH; ii) the development of a VNF platform prototype, called COmprehensive VirtualizEd NF (COVEN) platform, based on the proposed reference architecture; and iii) the identification of critical attributes provided by both VNFCs and NSH that enable the development of advanced network functions and services.

The remaining of this paper is organized as follows. Section II presents the background on VNF along with its main concepts, requirements, and a brief review of the literature. Section III presents relevant related works. In Section IV, we propose a reference architecture for developing VNF platforms, which enables network operators to achieve improvements in flexibility and management. In Section V, we instantiate the architecture by describing a running prototype platform implemented to support virtualized network functions. In Section VI, we evaluate the performance of our prototype in a case study. Finally, in Section VII, conclusions and perspectives for future work are presented.

II. Virtualized Network Function in a Nutshell

Network Function Virtualization (NFV) is a network paradigm that employs current virtualization technologies (*e.g.*, virtual machines, and containers) to execute softwarized Network Functions (NF) in commodity hardware [1]. The

978-1-7281-0962-6/19/$31.00 ©2019 IEEE

NFV architectural framework is defined by the European Telecommunications Standards Institute (ETSI) [7]. This architecture is composed of three main blocks: NFV Infrastructure (NFVI), NFV Management and Orchestration (NFV MANO), and Virtualized Network Functions (VNF). The NFVI includes the physical and virtualized computing resources that are employed for the execution of network functions. NFV MANO includes standards for virtualized infrastructure management, NF lifecycle control and monitoring, and network service orchestration.

In particular, the Virtualized Network Functions block corresponds to the specific network functions virtual instances. These instances are responsible for network traffic processing, and are the core block of the NFV paradigm. A VNF can process different network layers to execute a variety of functions, such as routing, DHCP, DPI, and IPS. A VNF is composed of two main parts [2]: the VNF platform (*e.g.*, ClickOS [4], OpenNetVM [8], and Click-on-OSv [3]); and the NF software implementation which runs on the platform. In addition, an Element Manager (EM) is employed to configure and monitor a VNF instance, providing the communication interface with the NFV MANO block.

Sophisticated VNFs can be designed through the composition of many VNF Components (VNFC). VNFCs are internal components which implement a subset of the VNF operations [9]. Usually, a standalone VNFC performs a particular operation with lower complexity, but the combination of VNFCs enables the creation of complete network functions. A VNFC is deployed as a virtualized element, but its lifecycle depends on its parent VNF. Also, VNFCs are reusable and specific components can be applied for different network functions (*e.g.*, an application level signature identifier can be used either as part of a deep packet inspector or an L7 load balancer).

Steering traffic through several VNFs in a predefined sequence defines a structure called Service Function Chain (SFC) [10]. An SFC is composed of VNFs, boundary nodes (*e.g.*, ingress and egress data points), and the virtual connections among them. The Internet Engineering Task Force (IETF) RFC 7665 [10] provides a standardized architecture for the SFC implementation. Furthermore, to allow traffic steering through the various possible paths of an SFC, the IETF also proposed the Network Services Header (NSH) [6]. Each VNF can be either NSH-aware or NSH-unaware. An NSH-aware VNF processes and updates the service header during its execution. Otherwise, NSH-unaware VNFs work with a proxy instance that recognizes and processes the service header externally the network function.

Finally, VNF platforms are designed to host and execute NFs and VNFCs. These platforms must observe a collection of requirements in order to be compliant with the ETSI NFV development model (*e.g.*, hardware independence, elasticity, and reliability) [11] and ETSI NFV virtualization assumptions (*e.g.*, portability, performance, integration, management, and scalability) [5]. In a previous work [2], we have shown how a VNF platform can explore the NSH processing through a generic architecture. However, despite the recent research initiatives, VNF platforms are still under-explored. For example, current platforms and architectures do not provide a simple way to deploy, execute, and manage VNFCs. Moreover, the interconnection between platforms internal modules and with other NFV architecture components are not formalized.

III. Related Work

Multiple VNF platforms have been proposed in recent years. These platforms, however, are being developed without following any standardized architecture. Furthermore, they do not entirely meet the VNF requirements described in [5] and do not natively support some essential and recent NFV features (*e.g.*, VNFC and NSH). Next, state-of-the-art VNF platforms currently available are presented and discussed.

ClickOS [4] is a platform based on a paravirtualized tiny operational system (MiniOS) with a netmap-enabled network stack. The ClickOS virtualization is dependent on a modified version of Xen hypervisor with VirtuAl Local Ethernet (VALE) and netmap support. However, the ClickOS platform architecture is inflexible and executes only network functions developed in Click Modular Router. Also, it can only be managed by using the XenStore solution.

Click-on-OSv [3] uses a paravirtualized minimalist operational system, called OSv, to host network functions. This platform employs the Intel Data Plane Development Kit (DPDK) to achieve high throughput on packet processing and provides a custom management agent to enable access to both configuration parameters and runtime performance metrics. However, this platform is built as a monolithic system, thus requiring the operating system to be recompiled in case of major modifications. Finally, Click-on-OSv only allows network functions implemented with the Click Modular Router.

OpenNetVM [8] is a multi-VNF platform based on containers with the DPDK network stack. This platform is deployed in commodity hardware and aims to execute network functions with fast network traffic I/O. In OpenNetVM, the NFs are created within a proprietary framework (NFLib). Its architecture consists of multiple VNFs interconnected through a shared data plane coordinated by an internal manager. Thus, this platform can also create a service chain in a single physical machine.

The OPNFV SampleVNF[1] is the VNF testing platform within the OPNFV project. This platform is based on a Ubuntu virtual machine with DPDK support and does not have any other native internal module. In this way, the network functions must be manually developed and executed. SampleVNF is a very simplified platform, but it should be able to accomplish compatibility and performance testing of NFs before they are deployed on the OPNFV environment.

The presented VNF platforms do not support some important NFV features (*e.g.*, VNFCs and NSH). In general, the solutions are developed as immutable systems and any adaptations, such as replacing or adding packet accelerators

[1] https://wiki.opnfv.org/display/SAM/SampleVNF+-+Home

(network stack) or NF programming languages, must be done by modifying the platform source code. Furthermore, current platforms do not meet all the NFV requirements defined by ETSI. For example, ClickOS depends on Xen hypervisor (integration), the container-based platforms cannot be migrated to different hardware architectures (portability), and platforms that employ Intel DPDK present high CPU overhead due the network stack idle waiting (performance).

IV. A VNF Platform Architecture

There is currently no standard architecture for the design and development of VNF platforms, with features such as NSH processing, VNFC deployment, internal modules dynamic traffic steering, and elastic life cycle management. Flexible and holistic VNF platforms are a fundamental necessity to meet multiple NFV requirements (*e.g.*, portability, performance, integration, management, and scalability) and to accomplish the needs of modern softwarized networks. VNF platforms must also be created with integration in mind. There are several systems (OSS/BSS, Hypervisors) and NFV architecture blocks and elements (NFVI, VNFM, EMS) that must work together with multiple VNFs in order to adequately provide virtualized network services [7].

In this work, we propose a flexible, generic, and comprehensive architecture for VNF platforms. The proposed platform, depicted in Figure 1, consists of six main modules deployed on a VNF core (*i.e.*, a virtualized host, such as virtual machine or container): *(i)* Virtual Network Subsystem, *(ii)* Internal Traffic Forwarder, *(iii)* NSH Processor, *(iv)* Packet Processing Subsystem, *(v)* Management Agent, and *(vi)* Extended Agents. Each module performs specific operations within the VNF and can process network packets from both the data and control planes (depicted with solid lines) and the management plane (depicted with dashed lines). External interfaces to the VNF Core are also defined in the architecture to enable the use of virtualized resources (available in the NFVI) and to support both management and orchestration operations (by using EMS and VNFM systems).

Modules of the architecture are designed to be loosely coupled and with well-defined access interfaces. In this way, each module can be redesigned or replaced as the technology evolves. Modules are described below:

- **Virtual Network Subsystem** (VNS) – This module is responsible for accessing the Virtual Network Interface Controllers (VNICs) – provided by the hypervisor – for sending/receiving network packets. Typically, this operation, when executed by the native network stack in traditional operating systems, is not optimized to support the performance requirements of high-speed networks (*e.g.*, 40GbE/100GbE). To tackle this problem, several packet acceleration tools (*e.g.*, netmap [12], PacketShader [13], Intel DPDK[2], PF_RING/DNA[3], and OpenOnload[4]) have been proposed and extensively evaluated. These systems can replace the traditional L2 Socket approach for traffic steering and are satisfactory solutions for NFV-based networks.

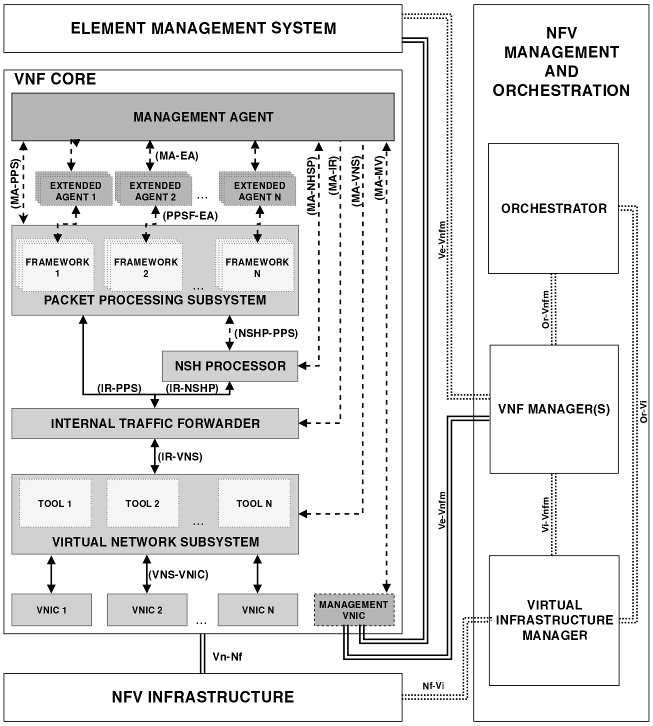

Fig. 1. VNF Platform Architecture

- **Internal Traffic Forwarder** (ITF) – Once the network packets are captured by the VNS, they are forwarded internally to the VNF Core by the Internal Traffic Forwarder. The configuration of this module is a task of the Management Agent (discussed later), which specifies the order of processing among the several VNFCs. Once the Internal Traffic Forwarder is initialized, it uses communication channels (*e.g.*, using shared memory, pipes, sockets) to forward the network packets between the VNS, the Packet Processing Subsystem (PPS), and the NSH Processor (NSHP). The use of an ITF allows VNFCs to be created individually.
- **NSH Processor** (NSHP) – The IETF specified the Network Service Header (NSH) that is inserted in packets/frames to provide service function paths [6]. However, despite its advantages, the use of NSH is optional to steer traffic across multiple VNFs. In order to provide NSH support, we define the NSH Processor, which provides an abstraction for the network functions regarding the existence of NSH packets. Specifically, when this module is activated, the ITF forwards the network traffic to be processed by the NSHP. Alternatively, the network packet is forwarded directly to the corresponding VNFC. NSHP acts on the specific NSH fields that may be modified when traversing a network path (*i.e.,* the Service Index - SI; and Context Header - CH). NSHP provides the

[2] https://www.dpdk.org

[3] https://www.ntop.org/products/packet-capture/pf_ring

[4] https://www.openonload.org

following operations: NSH removal, NSH reinsertion, CH retrieval, and CH update.

- **Packet Processing Subsystem** (PPS) – This module corresponds to the frameworks used to implement and execute network functions. Basically, these frameworks include applications (*e.g.*, Click Modular Router [14] and Vector Packet Processing[5]), programming languages (*e.g.*, C, C++, Python), libraries (*e.g.*, Scapy, libtins, libnet), or even single routines that support the construction and handling of network packets.
- **Management Agent** (MA) – The primary goal of a Management Agent is to monitor and control the execution of VNFs. Furthermore, it is also responsible for coordinating the execution of all internal modules of the VNF platform. MA provides five main operations: request, retrieve, start, stop, and monitor. The request operation receives a VNF Package (VNFP) [15] from the network operator and deploys the specified VNF instance. Once a VNF is executing, retrieve operations can be used to obtain information about the VNF instance (*e.g.*, VNF ID, network interfaces). The start and stop operations are essential for VNF lifecycle management. Finally, the monitoring operation is responsible for measuring performance indicators from the VNF Core (*e.g.*, CPU, memory, and network usage) and providing information retrieved from the extended agents deployed in the VNF platform.
- **Extended Agent** (EA) – This module is controlled by the Management Agent and is used to monitor/control each network function or component. It is supposed to be developed by the creator of the VNF/VNFC, as it acts on the individual management data of those implementations (*e.g.*, number of packets discarded by a firewall). This module must provide at least one standard operation which we call "list". This operation is used by MA to discover all the management data that can be accessed by network operators. To the best of our knowledge, the EA is the first effort to enable customized monitoring of a particular VNFC/NF.

All the modules are controlled by the MA, which upon receiving a VNFP (MA-MV interface), does the validation, extracts the relevant information for deploying the VNF, and configures the internal modules accordingly. For example, VNFP may contain information about the set of VNFCs that compose an NF, which is essential for the ITF to forward the traffic to the proper components. The initial request for VNF deployment must also provide more specific information, such as the extended agents to be instantiated together with the network function components.

The MA also retrieves VNFC source code from the VNFP, requests the PPS to create the associated communication channels, and to start the execution of VNFCs (Figure 1 MA-EA and PPSF-EA). Once this operation completes, PPS returns a success/failure confirmation to the MA (MA-PPS). In case of failures, a rollback mechanism can be employed to abort the instantiation process properly. When NSH is used, the MA starts the NSHP (MA-NSHP). NSHP then creates a communication channel with the PPS (NSHP-PPS) to allow the network functions to access the NSH context header. VNICs are connected to the VNS (VNS-VNIC) based on the original request specified in the VNFP, processed by the MA (MA-VNS).

Finally, the ITF is initiated with two default communication channels: ITF-VNS and ITF-PPS. The former is used by the ITF to retrieve network packets from the VNS, while the latter is used by VNFCs to access the packets to be processed. A third connection labeled ITF-NSHP is used (i) to remove the NSH before it is processed by any VNFC and (ii) to reinsert the NSH after the last VNFC of the path.

The reference architecture described in this section defines the key modules responsible for the deployment of both VNFCs and SFCs [10]. We believe the architecture can provide a valuable reference for the design and development of VNF platforms, working as a guideline to integrate distinct modules in order to create complete solutions.

V. The COVEN Platform

As a proof of concept of the proposed VNF platform architecture we implemented all modules on a prototype platform called COmprehensive VirtualizEd NF (COVEN) [6].

A. *Platform prototype*

The COVEN platform employs Debian Jessie (8.11) as the VNF Core. Debian Jessie is a generic operating system designed for the execution of miscellaneous tasks, thus it allowed both the development of prototype modules and of the network functions that run on the platform, without software compatibility concerns. We note however that production platforms should be based on lightweight virtual machines or containers, such as CoreOS, Alpine, and TinyCore, that properly support NFV requirements (*e.g.*, performance, portability, and integration).

The Virtual Network Subsystem was implemented with L2 sockets. Although we have employed only the L2 sockets tool, packet accelerators can be easily included in the platform through a standard interface template. The Internal Traffic Forwarder communicates with other internal modules of the architecture (*i.e.*, VNS and NSHP) using shared-memory to boost the performance. For the communication with PPS (*i.e.* NFs or VNFCs), L3 Sockets were employed, which may affect the latency and throughput when compared to shared memory. However, L3 Sockets provide greater flexibility for the development of NFs and VNFCs.

The NSH Processor acts as a proxy and is executed on demand if requested by the network operator. The NSHP receives the network traffic as input from the ITF before it is processed by PPS, removes the NSH, updates the service index field and saves it locally. The network traffic is then

[5] https://blogs.cisco.com/tag/vector-packet-processing

[6] https://github.com/ViniGarcia/COVEN

steered to the PPS for packet processing. After that, the traffic is delivered to the NSHP again so that NSH can be reinserted. The NSHP also waits for requests to retrieve or update the context header (which has a fixed-length [6]). For the Packet Processing Subsystem, five frameworks (*i.e.*, Click Modular Router, C, Python 3, Java, and JavaScript) are integrated and natively provided for the network operators.

The Management Agent was developed using the Bottle[7] library and executes all the basic lifecycle operations: [16] start, stop, turn off, monitor, and configure. In particular, for monitoring, we implemented three basic operations: ´´list´´, shows all management and monitoring queries available ; ´´check´´, checks VNFCs heartbeats with NetCat[8]; and ´´request´´, provide the communication between the network operators and the VNFCs Extend Agents. We highlight that these operations can be accessed both by network operators and operational blocks/modules of the NFV architecture (*i.e.*, EMS and VNF Manager).

B. Interconnection between modules

Before the platform is executed, the NFV Orchestrator or a user must provide a configuration file containing the network function and the routines to initialize the modules. The platform receives the corresponding descriptor and configures its internal connections in order to allow the execution of the function.

The (VNS-VNIC) connections can be set up with an L2 socket tool. The (ITF-VNS), (ITF-NSHP), and (ITF-PPS) connections are implemented using inter-process shared memory. The race conditions caused by shared memory write operations is treated using mutexes. The (NSHP-PPS) consists of a REST interface that allows the execution of NSH Context Header operations. All connections related to the management plane (*i.e.*, (MA-PPS), (MA-EA), (PPSF-EA), (MA-NSHP), (MA-ITF), (MA-VNS), and (MA-MV)) are also created using REST interfaces.

Except for inter-VNFC communication, all network packets arriving at the NIC are directly captured by the Virtual Network Subsystem, which forwards those packets to the Internal Traffic Forwarder via shared memory. If NSH processing is being used, the packets are first delivered to the NSH Processor before being processed by the Packet Processing Subsystem. Otherwise, the NSHP module is bypassed and packets are directly delivered to the proper NF/VNFC.

VI. Evaluation

To evaluate the COVEN platform prototype, a case study is presented. The platform was executed on an Intel Core i7-4790K@3.60Ghz server with 8GB RAM DDR4 running Debian 8. The platform was configured to use L2 Sockets as the virtual network tool and NFs can be developed using C, CMR, Java, and Python3 frameworks. All the experiments were repeated 30 times, considering a confidence level of 95%.

The proposed architecture enables the composition of heterogeneous VNFCs into a single VNF. We argue that VNF creators may benefit from choosing multiple VNF development tools to fulfill specific requirements for each component. For example, sophisticated components (*e.g.*, analyzing the payload of ciphered packets) cannot be created using the default elements of CMR, thus requiring more sophisticated programming languages. Furthermore, allowing multiple frameworks to cooperate within a VNF platform also improves the flexibility and reusability of VNFCs, which can be dynamically composed into customized VNFs offered by modern NFV Marketplaces (*e.g.*, FENDE [17]).

The case study, shown in Figure 2, consists of the creation of an L7 Firewall employing VNFC and NSH. The ultimate goal of this network function is to block Skype traffic by using the fingerprint-based approach [18]. The detection process consists of (i) port detection (80 and 443), (ii) payload pattern inspection (first 72 bytes), and (iii) the identification of similar data in different positions of the payload.

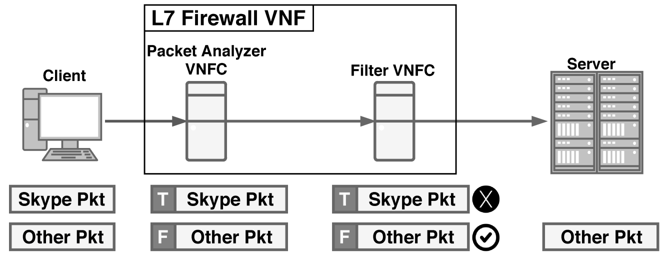

Fig. 2. L7 Firewall Case Study

Two separate components were created: a packet analyzer and a packet filter. The packet analyzer was developed using the Scapy[9] Python3 library, while the packet filter was implemented using C, CMR, Java, and Python3. The packet analyzer receives network traffic and searches for Skype fingerprints. All packets coming from ports 80/443 and presenting Skype patterns in the payload receive a mark in the NSH context header and are then forwarded, while the remaining packets are forwarded without any marking (in-band control). The second component (*i.e.*, the packet filter) checks the context header and discards all the packets that are considered to possibly be Skype traffic. These components were chained to create the L7 Firewall, but they are independent and can be executed alone or be incorporated as part of other complex network functions.

In the case study, we evaluated the RTT between the client and the server by using a combination of VNFCs running within a VNF: the packet analyzer component plus the packet filter component. Figure 3 presents the results. It is possible to conclude that for this case study the combination Python-Python presents the worst results, while the combination Python-C presents the best. The combinations Python-CMR and Python-Java present results that are close to each and neither as bad nor as good as the others.

[7] https://github.com/bottlepy/bottle

[8] http://netcat.sourceforge.net

[9] https://scapy.net

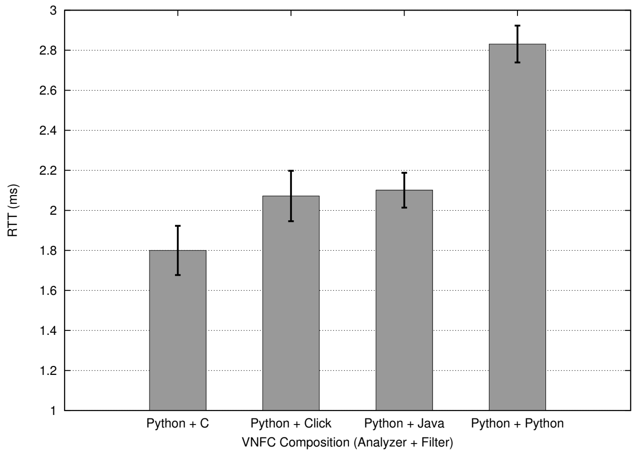

Fig. 3. L7 Firewall RTT

These results can be explained by the fact that Python is interpreted on runtime; Java is bytecode precompiled; CMR involves the translation from a high-level specification language to C++; and C is a low-level and compiled language. It is important to notice, however, that although components implemented in C leverage its optimized packet processing, the development of more sophisticated components can benefit from higher-level languages and libraries available for other frameworks (*e.g.*, Python3 Scapy and Java Pcap4J), even with lower performance. Similar conclusions can be observed in other development scenarios, for example, when embedding assembly code in higher-level programming languages (*e.g.*, C, C++), or when using the Java Native Interface (JNI) framework.

VII. Conclusion

Network Function Virtualization (NFV) has been proposed as a novel network paradigm that migrates the physical network functions (*i.e.*, physical appliances) to a software plane by using virtualization techniques, such as virtual machines and containers. However, despite the recent works and standardization in the field, many research opportunities are still open. Among these opportunities there are several related to VNF platforms. VNF platforms provide resources and support the execution of virtual NFs. Even though VNF platforms perform an important role in the NFV paradigm, there is no *de facto* architecture that specifies their internal operational modules. Furthermore, some important NFV features, such as NSH and VNFC, are not covered by current platforms (*e.g.*, ClickOS, Click-on-OSv, OpenNetVM, and SampleVNF).

In this work we proposed a comprehensive architecture for VNF Platforms. We identified and described the basic operational modules, as well as their interconnections. This architecture ultimately leads to the creation of extensible solutions to support interoperable NFV technology. In addition, we presented a prototype, called the COVEN platform, that implements the proposed architecture. Finally, we have deployed a evaluation scenario that uses the NSH and VNFC technologies to execute an L7 firewall. We observed that creating NFs by using individual components implemented in different languages brings benefits regarding performance and development flexibility. This approach enables, for example, the VNF developer to choose the best mix of languages for his/her specific needs.

Future work includes extending and improving the COVEN platform. Perhaps the first task is to reimplement the prototype using a low-level and high-performance language, the C language is currently the best option. Other improvements include reimplementing all the data plane connections using shared memory in order to reduce the internal delay introduced by the use of L3 sockets. We also plan to allow COVEN to fully support VNFCs, including new features and functionalities such as for example VNFC bottleneck detection and allowing the dynamic composition of NFs. Finally, we hope to get a broader group of users that will provide new case studies to further evaluate the proposed architecture in realistic settings.

References

[1] E. T. S. I. NFV, “Network functions virtualisation – an introduction, benefits, enablers, challenges & call for action,” 2012.

[2] V. Garcia, L. Marcuzzo, G. Souza, L. Bondan, J. Nobre, A. Schaeffer-Filho, C. dos Santos, L. Granville, and E. Duarte, “An nsh-enabled architecture for virtualized network function platforms,” in *Conf. on Advanced Information Networking and Applications*, 2019.

[3] L. Marcuzzo, V. Garcia, V. Cunha, D. Corujo, J. Barraca, R. Aguiar, A. Schaeffer-Filho, L. Granville, and C. dos Santos, “Click-on-osv: A platform for running click-based middleboxes,” in *Symp. on Integrated Network and Service Management*, 2017.

[4] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, “Clickos and the art of network function virtualization,” in *Conf. on Networked Systems Design and Implementation*, 2014.

[5] E. T. S. I. ISG, “Network functions virtualisation (nfv): Virtualisation requirements,” 2013.

[6] P. Quinn, U. Elzur, and C. Pignataro, “Network Service Header (NSH) - RFC 8300,” 2018.

[7] E. T. S. I. GS, “Network functions virtualisation (nfv); architectural framework,” 2014.

[8] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, “Opennetvm: A platform for high performance network service chains,” in *Work. on Hot Topics in Middleboxes and Network Function Virtualization*, 2016.

[9] E. T. S. I. GS, “Network function virtualisation (nfv): Terminology for main concepts in nfv,” 2014.

[10] J. Halpern and C. Pignataro, “Service Function Chaining (SFC) Architecture - RFC 7665,” 2015.

[11] E. T. S. I. G. SWA, “Virtual network functions architecture,” 2014.

[12] L. Rizzo, “netmap: A novel framework for fast packet i/o,” in *Annual Technical Conference*, 2012.

[13] S. Han, K. Jang, K. Park, and S. Moon, “Massively-parallel packet processing with gpus to accelerate software routers,” *Symp. on Networked Systems Design and Implementation*, 2010.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *Trans. on Computer Systems*, vol. 18, no. 3, 2000.

[15] E. IFA, “Vnf descriptor and packaging specification,” 2018.

[16] L. Bondan, C. Santos, and L. Granville, “Management requirements for clickos-based network function virtualization,” in *Conf. on Network and Service Management*, 2014.

[17] L. Bondan, M. Franco, L. Marcuzzo, G. Venancio, R. Santos, R. Pfitscher, E. Scheid, B. Stiller, F. De Turck, E. Duarte, A. Schaeffer-Filho, C. Santos, and L. Granville, “Fende: Marketplace-based distribution, execution, and life cycle management of vnfs,” *Communications Magazine*, vol. 57, no. 1, 2019.

[18] S. Ehlert, S. Petgang, T. Magedanz, and D. Sisalem, “Analysis and signature of skype voip session traffic,” in *Conf. on Communications*, 2006.