Modeling Native Software Components as Virtual Network Functions

Modeling Native Software Components as Virtual Network Functions

Mario Baldi, Roberto Bonafiglia, Fulvio Risso, Amedeo Sapio
Department of Control and Computer Engineering, Politecnico di Torino, Italy
{mario.baldi, roberto.bonafiglia, fulvio.risso, amedeo.sapio}@polito.it

ABSTRACT

Virtual Network Functions (VNFs) are often realized using virtual machines (VMs) because they provide an isolated environment compatible with classical cloud computing technologies. However, VMs are demanding in terms of required resources (CPU and memory) and therefore not suitable for low-cost devices like residential gateways. Such equipment often runs a Linux-based operating system that includes by default a (large) number of common network functions, which can provide some of the services otherwise offered by simple VNFs, but with reduced overhead. In this paper those native software components are made available through a Network Function Virtualization (NFV) platform, thus making their use transparent from the VNF developer point of view.

CCS Concepts

•**Networks** → *Cloud computing; Middle boxes / network appliances;*

Keywords

Network Functions Virtualization; Virtual Network Functions; Service Orchestration

1. INTRODUCTION

While Network Service Providers (NSPs) could benefit from a flexible infrastructure that can rapidly and efficiently provide dedicated, on-demand network services, so far they have not been able to leverage it due to the complexity of deploying middleboxes in the network. Network Functions Virtualization (NFV) has the potential to overcome this by exploiting virtualization techniques, typical of cloud computing, to instantiate Virtualized Network Functions (VNFs) in compute nodes with unprecedented agility. However, current NFV implementations are designed for (centralized) data centers, while NSPs leverage a distributed infrastructure consisting of heterogeneous devices. More specifically, it would be particularly beneficial to offer NFV capability in Customer Premise Equipment (CPE), which is particularly challenging because it is usually based on low-cost hardware. On the other hand, CPE operating systems are often some Linux flavor, which embeds a large number of software-based "native" network functions, such as firewall and NAT (e.g., `iptables`), virtual switch (e.g., `linuxbridge`), and more.

This paper proposes a solution to integrate such functions in an existing NFV infrastructure so that while resource-hungry VNFs are run in the NSP data center, simpler ones are run in the CPE, possibly as **Native Network Functions** (NNFs). Our solution facilitates the provision of services that require Network Functions (NFs) close to the end user (e.g., IPsec terminator), enabling the possibility to execute both VNFs and NNFs, which combines the benefits of flexibility and low execution overhead. In order to support NNF integration, the compute controller in an NFV server is extended with an additional plugin that manages all the NNFs available on the node, along with standard VNFs.

2. ARCHITECTURE

Our proposal is based on the compute node presented in [1], which follows closely the NFV architecture. As shown in Figure 1, a local orchestrator receives a Network Functions Forwarding Graph (NF-FG) and instantiates the required VNFs. For each NF-FG a new software switch, called Logical Switch Instance (LSI), is created in order to steer traffic among the corresponding VNFs in the right order, while a base LSI is in charge of classifying the traffic received by the node and delivering it to the proper NF-FG-specific LSI. VNFs are instantiated and managed by a compute manager through ad-hoc drivers matching the specific VNF support technology (e.g., VM, Docker, DPDK process), while each LSI is managed by its own OpenFlow controller that dynamically inserts the proper rules in flow table(s). All

ACM ISBN XXX.
DOI: XXX

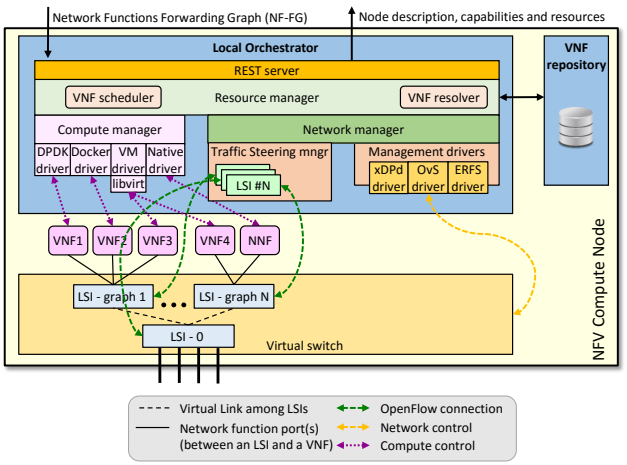

Figure 1: Compute node architecture.

the above drivers must implement a specific abstraction defined by the local orchestrator, which enables multiple drivers to coexist, hence implementing complex services that include VNFs created with different technologies (e.g., VMs and Docker).

Profitably integrating NNFs in an NFV infrastructure requires to identify significant differences with traditional VNFs and translate them into a set of constraints for the local orchestrator when evaluating whether to use NNFs or traditional VNFs. A major one of such differences, is that some NNFs do not allow to concurrently spin up multiple instances. Such NNFs must be "sharable" to have multiple service graphs traversing the same NF. A NNF is "sharable" only if (i) it can use an ad-hoc marking mechanism to distinguish between traffic belonging to different service graphs, hence emulating the execution of different NF instances, and (ii) the NNF can create multiple *internal* paths that are needed to process the above multiple traffic streams in isolation. Moreover, an additional adaptation layer is required to cope with the fact that NNFs may be designed to receive traffic from a single network interface. Such layer attaches the NNF to one port of the switch and configures it to receive the traffic from multiple service graphs, appropriately marked to make it distinguishable.

When a NNF should be used, the compute manager selects a NNF driver developed as part of this work[1]. This NNF driver implements the same abstraction defined for the other compute drivers and dynamically activates the plugin associated to the selected NNF, which is implemented as a collection of `bash` scripts that control the basic lifecycle (create, update, etc.) of the NF.

For each NF in a NF-FG, the orchestrator decides whether to deploy it as VNF or NNF based on its knowledge of the node capability set, the available NNFs and their characteristics (e.g., whether they are sharable), and their status (e.g., already used in another chain). The NNF driver starts the NNF in a new *network* namespace, to provide a basic form of isolation, and configures the NNF with a predefined configuration script. Support for a dynamic configuration mechanism able to translate a generic NF configuration, provided by the orchestrator, in commands appropriate to the specific NNF is not in the scope of this initial implementation and will be targeted by future work.

[1]Source code available at https://github.com/netgroup-polito/un-orchestrator/.

Table 1: Results with IPSec client VNFs

Platform	Through.	RAM	Image size
KVM/QEMU	796 Mbps	390.6 MB	522 MB
Docker	1095 Mbps	24.2 MB	240 MB
Native NF	1094 Mbps	19.4 MB	5 MB

3. VALIDATION

In order to show the benefits offered by NNFs, we present the performance achieved in a simple use case in which a customer activates an IPSec endpoint VNF on his domestic CPE. This NF has been selected because it should be deployed the closest to the user in order to provide him with highest level of security, e.g., on the user home router. We compare the cost of running the Strongswan IPSec endpoint, configured to use the ESP protocol in tunnel mode, as a NNF, a Docker container and a VM using KVM/QEMU as hypervisor. The Strongswan implementation leverages kernel processing to handle packets faster, an expedient (very common among NFs) that highlights the limits of VNFs. The maximum throughput that can be obtained by the three NF flavors has been measured using iPerf and is presented in Table 1, together with the amount of RAM allocated at runtime and the size of the NF image.

Results show that the VM has worst performance, which is due to the additional virtualization layer and to the IPsec functionalities executing in user space (i.e., in the process, within the hypervisor, running the VM). The Docker and Native implementations have comparable performance, since both process packets in the host kernel space. However, Docker requires additional space for storing the Docker image, as well as additional libraries in the Linux operating system, which makes this technology not suitable for resource-constrained devices.

Acknowledgment

This work was conducted within the FP7 UNIFY and SECURED projects, which are partially funded by the Commission of the European Union. We thank Sergio Nuccio for his contribution to an early prototype.

4. REFERENCES

[1] I. Cerrato, T. Jungel, A. Palesandro, F. Risso, M. Sune, and H. 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.