Doctoral Dissertation
Doctoral Program in Computer Engineering (30thcycle)

Improving the Performance of Virtualized Network Services Based on NFV and SDN

By

Roberto Bonafiglia

******

Supervisor(s):
Prof. Fulvio Risso, Politecnico di Torino

Politecnico di Torino
2018

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Roberto Bonafiglia
2018

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my parents that always support me

Acknowledgements

I would like to thank my supervisor Fulvio Risso for the help that he gives me during my PhD career from the point of view of my research and also on a personal level. I would like to thank also the wonderful people that I met during this three years especially the other PhD students Ivano, Matteo, Serena, Fulvio, Francesco, Amedeo, Sebastiano, Gabriele and Jalol.

Abstract

Network Functions Virtualisation (NFV) proposes to move all the traditional network appliances, which require dedicated physical machine, onto virtualised environment (e.g,. Virtual Machine). In this way, many of the current physical devices present in the infrastructure are replaced with standard high volume servers, which could be located in Datacenters, at the edge of the network and in the end user premises. This enables a reduction of the required physical resources thanks to the use of virtualization technologies, already used in cloud computing, and allows services to be more dynamic and scalable. However, differently from traditional cloud applications which are rather demanding in terms of CPU power, network applications are mostly I/O bound, hence the virtualization technologies in use (either standard VM-based or lightweight ones) need to be improved to maximize the network performance.

A series of Virtual Network Functions (VNFs) can be connected to each other thanks to Software-Defined Networks (SDN) technologies (e.g., OpenFlow) to create a Network Function Forwarding Graph (NF-FG) that processes the network traffic in the configured order of the graph. Using NF-FGs it is possible to create arbitrary chains of services, and transparently configure different virtualized network services, which can be dynamically instantiated and rearranges depending on the requested service and its requirements.

However, the above virtualized technologies are rather demanding in terms of hardware resources (mainly CPU and memory), which may have a non-negligible impact on the cost of providing the services according to this paradigm. This thesis will investigate this problem, proposing a set of solutions that enable the novel NFV paradigm to be efficiently used, hence being able to guarantee both flexibility and efficiency in future network services.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network Function Virtualization (NFV) 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.

The European Telecommunication Standard Institute (ETSI) started the Industry Specification Group for NFV [1], with the aim of standardizing the main components of the NFV architecture. Instead, the IETF Service Function Chain [2] 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 can process different packet flows in parallel.

Within this context, the Software Defined Networking (SDN) technology is used mainly to create logical connections between multiple VNFs. The main idea of SDN is to centralize all the control logic of the network on a dedicated server (a.k.a. SDN controller), while the devices are in charge of performing only the packet forwarding

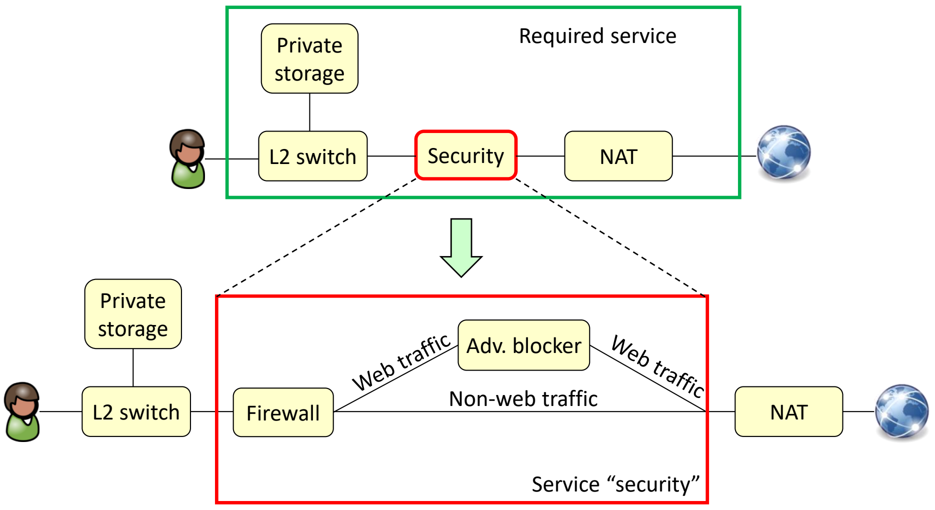


Fig. 1.1 Service graph expansion.

based on the rules configured by this controller. An example and enabler of the SDN concepts is the OpenFlow protocol [3]; in such a scenario, an OpenFlow switch can be used to distinguish the various types of traffic and steer them to the correct network function.

Using these concepts we can define a virtualized network service graph that is a set of functions suitably interconnected to implement a specific service, such as, for example a comprehensive security solution. As illustrated in Figure 1.1, links between functions can be potentially marked with the traffic that has to cross such connections, thus enabling the differentiation of various types of traffic. In addition, according to the recursive functional blocks concept proposed by ETSI [4], each function may, in turn, be defined as a service graph. As an example, the NF security block (e.g., a service) in Figure 1.1 is in fact a service graph comprising a firewall and an advertisement blocker (adv-blocker). Therefore, a service graph covers aspects of both the ETSI network service and the ETSI NF forwarding graph.

The idea of deploying a virtualized network service introduces some issues. Network applications are mostly I/O-intensive, differently from traditional cloud ones which are usually CPU-bounded, hence the the virtualised technologies need to be improved to maximize the network performance. Virtualization techniques may not be appropriate to execute services on edge nodes (e.g., home gateways) that feature limited hardware (e.g., CPU, memory) capabilities; moreover, a complex service may require the cooperation of the edge device and the cloud, with components

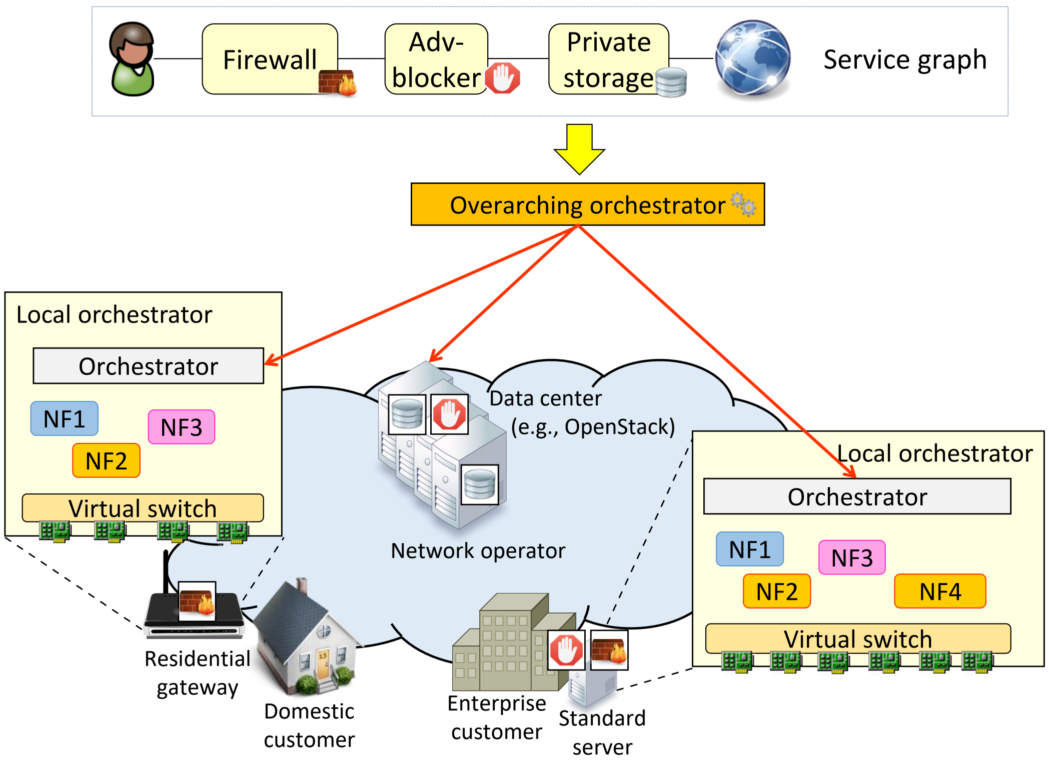


Fig. 1.2 Graph deploying on a NFV distributed architecture.

instantiated on both sides. This may require an overarching layer of orchestration that decides the service location; in this distributed scenario, the service has to be split in several sub-components, instantiated on different nodes, which need to be properly interconnected through logical links. Figure 1.2 provides an high-level view of the overall operations.

The aim of the dissertation is to address the problems that can be encountered on an orchestration layer to optimize the path that the user traffic has to flow on the required network functions. The technologies supported by the different nodes introduces penalties in terms of performance and the orchestration layer has to choose the right implementation. The thesis also addresses how to leverage on resource constrained devices as NFV node that can run types of NF and can be used during the placement decisions of the orhchestrator.

This dissertation addressed the described issues starting from Chapter 2, which provides with an analysis of exiting virtualization technologies for an NFV platform in term of performance with respect to network traffic; subsequently Chapter 3 describes an efficient algorithm for the data forwarding inside a single server.

Chapter 4 describes a native network application that can be deployed on a resource constrained device and leverages the presence of a distributed infrastructure. A lightweight NFV node, on-purpose designed to operate on resource constrained devices exploiting their native capabilities (software that runs on the bare hardware) as VNF, is presented in Chapter 5. However, more complex services can exceed the capabilities of a single node and require a distributed scenario, as VNFs are deployed over the whole network infrastructure, even on different technological domains; to tackle this challenge, Chapter 6 presents the concept of a multi domain orchestration for the deployment of network service graphs. Finally, Chapter 7 concludes the thesis.

Chapter 2

Assessing the performance of virtualization technologies for NFV

Part of the work described in this chapter has been previously published in [5].

2.1 Overview

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), while the paths among VNFs deployed on a server are created through virtual switches (vSwitches). While these technologies have been well tested/evaluated for the cloud computing environment, such a study is still missing in case of NFV services. The rationale is that cloud computing and NFV services differ both in the amount and in the type of traffic that has to be handled by applications and vSwitches, 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.

This chapter describes a preliminary benchmarking of VNFs chains deployed on a single server and based on the most common virtualization components. Particularly, KVM [6] and Docker [7] as execution environments, and Open vSwitch (OvS) [8] without and with DPDK enabled as vSwitches to steer the traffic among them in order to implement the chain(s).

This chapter is organized as follows. Section 2.2 provides an overview of the related works, while Section 2.3 details the technologies considered in our benchmarking. Tests are detailed in Section 2.4, while Section 2.5 concludes the chapter.

2.2 Related work

Several works available in literature evaluate the performance of virtualization components, both in the computing virtualization side (i.e., VMs, containers) and in the network virtualization side (i.e., vSwitches).

From the virtual networking side, [9] and [10] provide a deep evaluation of the behavior of OvS, analyzing how several factors, such as CPU frequency, packets size, number of rules and more, influence the performance of the vSwitch itself. Moreover, [9] measures the throughput obtained when OvS is exploited to interconnect a single VM both with the physical network and with a second VM executed on the same server. However, both the papers do not evaluate OvS when used to cascade several VNFs, as well as they do not consider Docker containers and OvS with DPDK enabled.

In [11], Casoni et al. measure the performance of chains of Linux containers (LXC) [12] interconnected through different technologies: the VALE vSwitch [13], the Linux bridge and the Virtual Ethernet interfaces (`veth`), missing some other widespread technologies such as OvS.

From the computing virtualization side, [14] studies the overhead introduced by the KVM hypervisor [6] when accessing to the disk and to the network. However, it does not evaluate such overhead in a VNF scenario, in which a packet has to potentially traverse several VNFs before leaving a server. This scenario is again not considered in [15], although it provides a comparison between LXC containers and KVM-based VMs. Xavier et al. [16] evaluates networking performance of several virtualization technologies: the Xen hypervisor [17], LXC, OpenVZ [18] and Linux-VServer [19]. However, the tests provided in the paper focus on High Performance Computing workloads, which can be consistently different from those experienced on a server running chains of VNFs.

2.3 Background

This section provides an initial evaluation of the overhead introduced by VNF chains deployed on a single server. Different technologies are considered for what regards both the virtual environment running the VNFs, and the vSwitch interconnecting such functions together and with the physical network, in order to identify their performance and compare such technologies among each other.

VMs are the main virtualization technology used to execute VNFs, since they provide strong isolation among VNFs themselves. However, lightweight containers are gaining momentum, since they still provide a form of isolation while having lower resources demand. As environment to execute VNFs, the section considers KVM-based VMs [6] and Docker [7] containers.

Interconnections among VNFs are implemented through a vSwitch, exploiting its capabilities to perform traffic steering based on multiple criteria such as the port ID and L2-L7 protocol fields. Although several vSwitches are available, the analysis is focused on the most widespread ones, namely Open vSwitch (OvS) [8] and OvS based on the Intel Data Plane Development Kit (DPDK) technology [20].

The remainder of this section provides an overview of the OvS I/O model, as well as it gives some insights on KVM-based VMs and Docker containers, mainly focusing on the way in which they connect to the vSwitch.

2.3.1 KVM-based virtual machines and networking

Figure 2.1 provides an overview of the components involved in the network I/O of VMs running in the KVM hypervisor.

Before detailing how VMs send/receive network packets, it is worth mentioning that KVM is just a kernel module that transforms the Linux kernel into an hypervisor, i.e., it provides to the Linux kernel the capability of running VMs. A VM is then

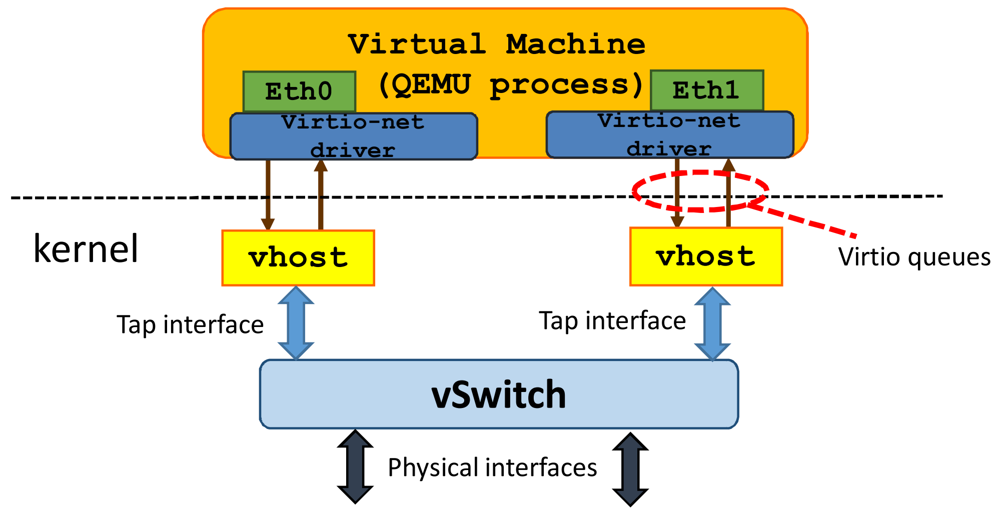


Fig. 2.1 Networking of a KVM-based virtual machine.

executed within QEMU, which is a Linux user-space process that exploits KVM to execute VMs. For instance, the VM memory is part of the memory assigned to the QEMU process, while each virtual CPU (vCPU) assigned to the VM corresponds to a different QEMU thread in the hypervisor.

As shown in Figure 2.1, the guest operating system (i.e., the operating system executed in the VM) accesses to the virtual NICs (vNICs) through the `virtio-net` driver [21], which is a driver optimized for the virtualization context. Each vNIC is associated with two `virtio queues` (used to transmit and receive packets) and a process running in the Linux kernel, namely the `vhost` module in the picture. As shown, `vhost` is connected to the `virtio queues` on one side, and to a `tap` interface on the other side, which is in turn attached to a vSwitch. `vhost` works in interrupt mode; particularly, it waits for notifications from both the VM and the `tap` and then, when such an interrupt arrives, it forwards packets from one side to the other.

As a final remark, the transmission of a batch of packets from a VM causes a *VM exit*; this means that the CPU stops to execute the guest (i.e., the vCPU thread), and runs a piece of code in the hypervisor, which performs the I/O operation on behalf of the guest. The same happens when an interrupt has to be “injected” in the VM, e.g., because `vhost` has to inform the guest that there are packets to be received. These

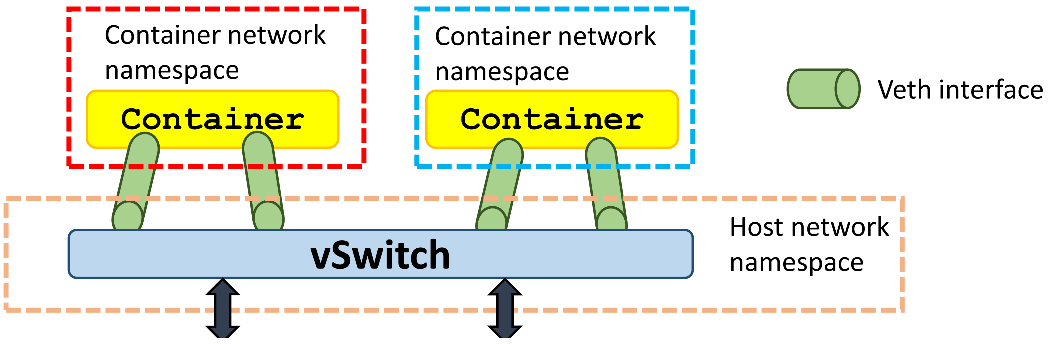


Fig. 2.2 Networking of a Docker container.

VM exits (and the subsequent *VM entries*) are one of the main causes of overhead in network I/O of VMs.

2.3.2 Docker containers and networking

Docker containers are a lightweight virtualization mechanism that, unlike VMs, do not run a complete operating system: all the containers share the host's kernel[1]. Containers represent a way to limit the resources visible by a running userland process; hence, if no process is running in the container, such a container is not associated with any thread in the host. Such this limitation of resources (e.g., CPU, memory) is achieved through the `cgroups` feature of the Linux kernel, while isolation is provided through the Linux namespaces, which give to processes running in the container a limited view of the process trees, networking, file system and more.

As shown in Figure 2.2, each container corresponds to a different network namespace; this means that it is only aware of those interfaces inserted into its own namespace, as well as it has a private network stack. According to the picture, packets can traverse the namespace boundary by means of `veth` pairs, which are in fact a pair of interfaces connected through a pipe: packets inserted in one end are received on the other end. Hence, by putting the two ends of the `veth` in different namespaces, it is possible to move packets from one network namespace to another. According to the picture, a vSwitch connects all the `veth` interfaces in the host namespace among each other and with the physical network.

[1]In other words, on the physical machine there is a single kernel, with a single scheduler, a single memory manager and so on.

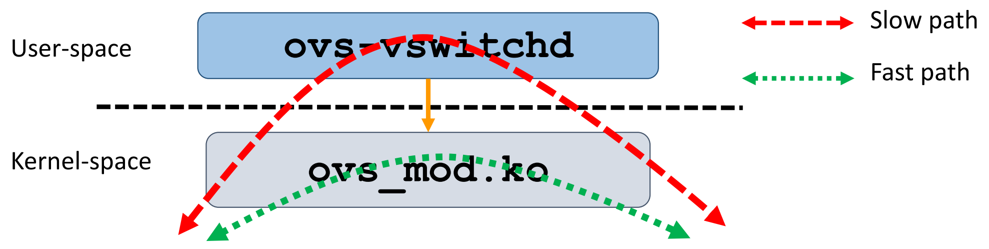


Fig. 2.3 Open vSwitch data-plane architecture.

2.3.3 Open vSwitch I/O model

OvS is a widespread vSwitch, whose data plane architecture is depicted in Figure 2.3; as shown, it consists of a user-space daemon and a kernel-space module, respectively referred as `ovs-vswitchd` and `ovs-mod.ko` in the picture.

The kernel module implements the fast path; it acts in fact as a cache for the user-space component, and includes all the last matched rules for increasing forwarding efficiency. In particular, when a packet enters into the vSwitch, it is first processed in the kernel module, which looks for a cached rule matching such a packet. In case of positive result, the corresponding action is executed, otherwise the packet is provided to the user-space.

The user-space daemon (that implements the slow path) contains in fact the full forwarding table of the vSwitch. By default, it implements the traditional MAC learning algorithm, although it can also be "programmed" by an external controller through the Openflow protocol[2]. Then, after that the first packet of a flow has been handled in user-space, the corresponding rule is offloaded to the kernel module, which will be able to handle all the subsequent packets of the same flow, thus significantly increasing the performance.

Interesting, there are no threads associated with the `ovs-mod.ko` kernel module; the in-kernel code of OvS consists in fact of a callback invoked by the software interrupt raised when a packet is available on a network interface. For instance, the OvS code (in the hypervisor) is executed in the context of the `ksoftirq` kernel thread in case of packets coming from the physical NIC, while it is executed in the context of the `vhost` associated with the "sender" vNIC in case of packets coming

[2]The capability to program the switching table of OvS from an external entity is exploited in NFV to create chains of VNFs.

from the VM (Figure 2.1). In case of packets coming from Docker (Figure 2.2), OvS runs in the context of the process executed in the container. This is possible because a single kernel exists, which is shared among the host and the containers.

2.3.4 OvS with DPDK I/O model

DPDK is a framework proposed by Intel that offers efficient implementations for a wide set of common functions, such as NIC packet input/output, memory allocation and queuing.

Unlike the standard OvS, enabling DPDK simply consists of a user-space process (with a variable number of threads) that continuously iterates over the physical NICs and the `virtio queues`. The `vhost` module shown in Figure 2.1 is in fact integrated in the vSwitch, so that the `tap` interface, which introduces overhead in the data path of packets to/from VMs, is removed. Furthermore, thanks to DPDK: *(i)* the vSwitch accesses to the physical NICs without the intervention of the operating system; *(ii)* packet transfer between the physical NICs and the vSwitch is done with zero-copy.

As a final remark, OvS with DPDK can also exchange packets with the VMs through shared memory: this solution removes the `vhost` at all, and does not require the `virtio-net` driver in the VMs. Although this configuration allows the vSwitch to move packets in a zero-copy fashion among VMs, we decided of not using it in our preliminary benchmarking of service chains. In fact, due to a design choice of DPDK, the same memory would be shared among all the VMs deployed, thus weakening the isolation among VMs themselves.

2.4 Performance characterization

This sections details our benchmarking of VNFs chains implemented on a single server. As already mentioned, we carried out several tests using different technologies for what regards both the virtualization environment used to run VNFs (KVM and Docker), and the vSwitch that properly steers the traffic among them in order to create the chain(s) (OvS and OvS with DPDK enabled).

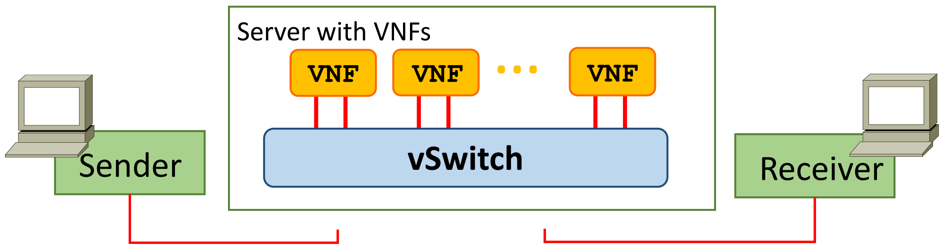


Fig. 2.4 Physical set up used for the tests.

Particularly, the tests focus on measuring the throughput and latency obtained when packets (of different size) flow through a server hosting one or more chains of different length. Each measurement lasted 100 seconds and was repeated 10 times; results are then averaged and reported in the graphs shown in the following of this section. Some graphs are provided with a bars view and a points-based representation of the maximum throughput. The former representation is referred to the left y axis, which provides the throughput in Mpps, while the latter is referred to the right y axis, where the throughput is measured in Gbps.

2.4.1 Hardware and software setup

As shown in Figure 2.4, the test environment includes a server that runs a vSwitch and a variable number of VNFs; the server is connected to a sender and to a receiver machine through two point-to-point 10Gbps Ethernet links. All the physical machines are equipped with an Intel Core i7-4770 @3.40GHz (4 physical cores plus hyper-threading), 32GB of memory, and run Fedora 20 with kernel 3.18.7-100.fc20.x86_64. The physical network interfaces are instead Intel X540-AT2.

As already mentioned, VNFs are executed either in KVM-based VMs[3], or in Docker containers. According to Figure 2.4, each VNF is equipped with two vNICs connected to the vSwitch. In order to steer the traffic among VNFs, the vSwitch is configured with Openflow rules matching the input port of packets, while the action is always the forwarding of packets through a specific port. In case of multiple service chains deployed on the server, the traffic entering from the physical port is split so that some packets enter in one chain, other packets in another chain and so on.

[3]In this case, each VM is associated with a single vCPU, i.e., the whole VM corresponds to a single thread in the hypervisor. Moreover, the guest operating is Ubuntu 14.04 with kernel 3.13.0-49-generic, 64 bit.

Table 2.1 Bare-metal throughput of the applications used in our measurements.

	Linux bridge	**`libpcap`-based bridge**	**DPDK-L2fw**
64B [Mpps (Gbps)]	2.39 (1,14)	1.29 (0.66)	8.78 (4.39)

Traffic splitting is based on the source MAC address of packets, which are properly generated so that they are equally distributed among all the available chains.

Particularly, our VNFs simply receive packets from one interface and forward them on the other; such applications, in a non-virtualized environment (i.e., when used to directly connect two physical interfaces), provide the throughput shown in Table 2.1.

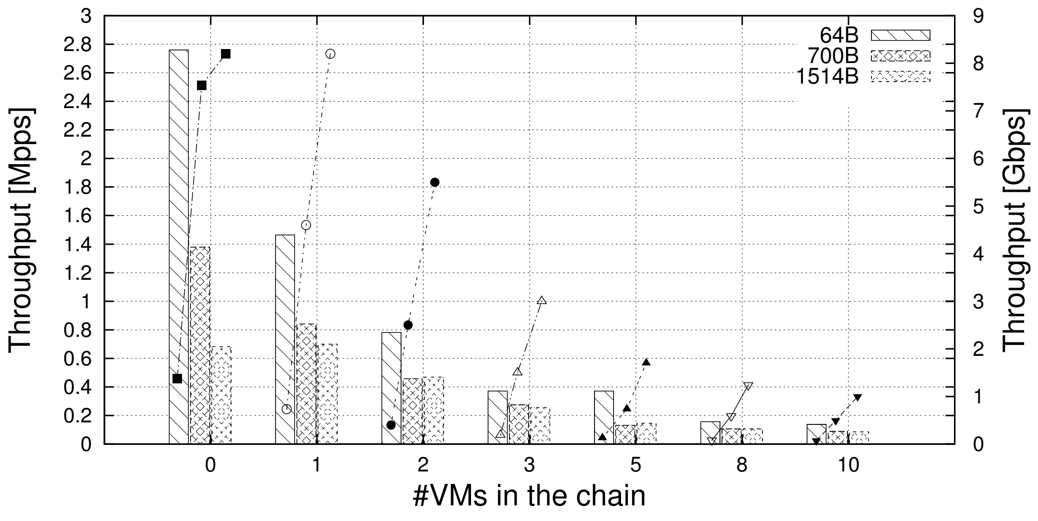
The throughput is measured using unidirectional traffic flowing from the sender to the receiver machine, respectively running a packet generator and a packet receiver based on the `pfring/dna` [22] library, which allows to transmit/receive packets at 10Gbps.

2.4.2 OvS-based chains: virtual machines vs Docker containers

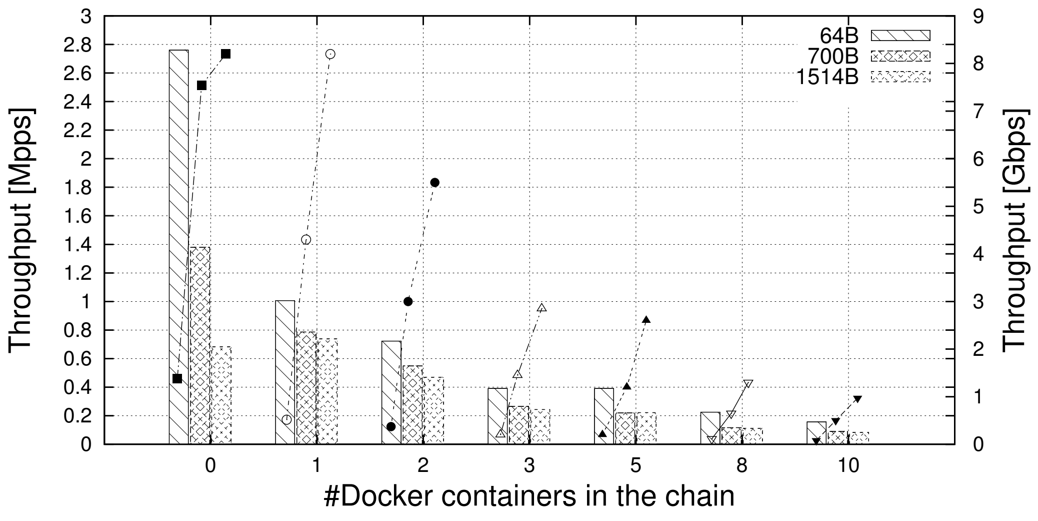
This section evaluates the throughput achieved when a single chain of VNFs is deployed on a server running OvS as vSwitch[4]. Measurements have been repeated with VMs and Docker containers, chains of growing length and packets of different size; results are reported in Figure 2.5. Those graphs have been obtained by executing the Linux bridge within VMs, while each container runs a simple user-space bridge based on `libpcap` [23]. In fact, with the Linux bridge inside containers, we got an unacceptable throughput of 3Mbps with 8 chained VNFs.

This poor result is a consequence of two factors: *(i)* the Linux bridge data plane is in fact a kernel-level callback executed in the context of the thread that provides packets to the vSwitch (similarly to OvS); *(ii)* all Docker containers share the kernel with the host (as detailed in Section 2.3.2). As a consequence, the `ksoftirq` kernel thread that processes the packet received from the physical NIC is also in charge of executing *(i)* the OvS code that forwards the packet in the first container; *(ii)* the data plane part of the Linux bridge launched in the first container (but running, in fact, in

[4]Note that, due to the small number of rules inserted in the vSwitch, the packet processing is entirely done in the kernel module (Section 2.3.3).



(a) KVM-based virtual machines



(b) Docker containers

Fig. 2.5 Throughput of a single chain (of growing length) with VNFs deployed.

the host kernel); *(iii)* again OvS, and so on, until the packet leaves the server through the physical NIC. In other words a single kernel thread executes all the operations associated with the packet, resulting in no parallelization at all and hence in really poor performance, especially in case of long chains.

Instead, in case of VMs, the vCPU thread (Section 2.3.1) executes the guest kernel and hence the Linux bridge, while `vhost` executes the OvS code, thus providing high parallelization in packets processing and then acceptable performance. In case of Docker, such a parallelization can be achieved by running a VNF that is actually a process (e.g, our simple `libpcap`-based bridge). This way, the OvS code between VNFs is executed in the context of this process, and the `ksoftirq` is just involved at the beginning of the service chain.

A comparison between Figure 2.5a and Figure 2.5b shows that chains implemented with the two virtualization technologies (when executing the proper application) present different throughput only when there is a single VNF per chain (and 64B packets), while they are almost equivalent in the other cases. However, according to Table 2.1, the Linux bridge provides nearly twice the throughput of the `libpcap`-based bridge, when executed in a non-virtualized environment. The fact that we got almost the same throughput by running these two applications respectively in VMs and containers, shows how the overhead introduced by full virtualization is higher than the overhead due to lightweight virtualization mechanisms.

Figure 2.5 also shows that the throughput is inversely proportional to the number of chained functions, regardless of the virtualization environment. In fact, more VNFs executed on the same server results in more contention on the CPU resource, as well as on the cache(s) and the Translation Lookaside Buffer (TLB). Furthermore, with longer chains, the probability that a packet is processed on different physical cores is higher, with a consequent increasing in the number of experienced cache misses. Moreover, although OvS is executed on several threads, there is just one instance of the forwarding table, which requires synchronization for example to increment the counters associated with the matched rules.

Figure 2.6 reports instead the throughput obtained when 64B packets are equally distributed among multiple chains of VMs. As evident, more chains result in lower throughput measured on the receiver machine, for the same reasons stated above in case of a single chains of different lengths (i.e., more CPU contention, more cache misses, etc.). Similar results have been achieved with containers.

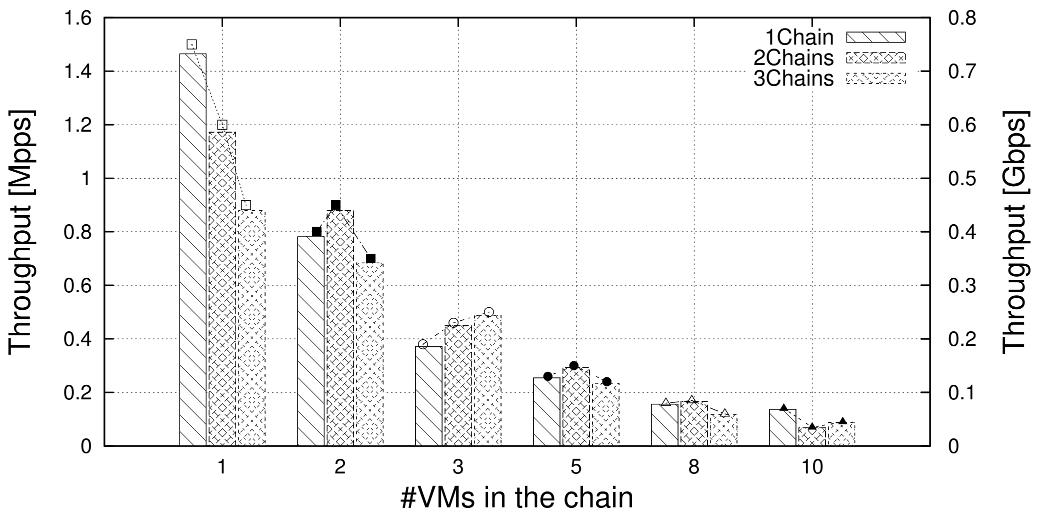


Fig. 2.6 Throughput with several chains implemented with VMs and OvS.

We can then conclude that Docker containers are well suited to run VNFs only in case of applications associated with a specific thread/process. In this case, they provide almost the same performance of KVM-based VMs, with the advantage of requiring less resources (e.g., memory), but with the drawback of providing less isolation.

2.4.3 KVM-based virtual machines: OvS with DPDK enabled and not

This section evaluates the performance of a chain of VNFs interconnected through OvS with DPDK, and compares such results with those achieved in case of OvS without DPDK. Particularly, in order to compare the two switching technologies, we consider VNFs executed in KVM-based VMs. Before analyzing the results, it is worth remembering that , unlike the standard OvS, DPDK is entirely executed in user-space and works in polling, i.e., it continuously iterates over the available NICs. Moreover, the `vhost` module (Figure 2.1) is replaced with a software layer integrated in DPDK, so that VMs directly communicate with the vSwitch.

As a first test, we consider again the Linux bridge as VNFs, obtaining a throughput that is just slightly better than those achieved with OvS and depicted in Figure 2.5a; e.g., 1.66Mpps (0.85Gbps) with 64B packets traversing a singe VNF. In

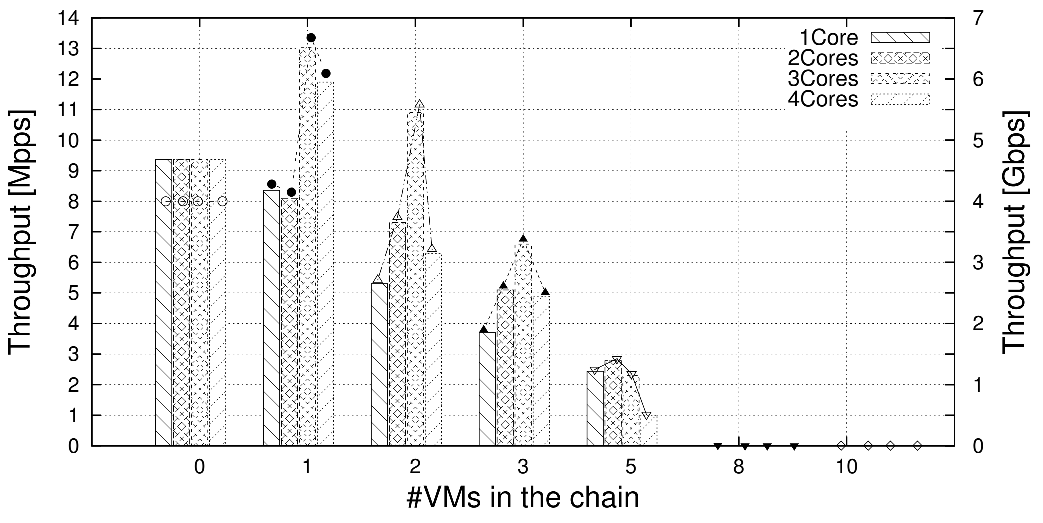


Fig. 2.7 Throughput with VMs chained through OvS with DPDK.

fact, although DPDK provides several enhancements with respect to OvS standard, such improvements are mitigated by the high overhead introduced by VMs.

Then, we installed DPDK in VMs as well, in order to execute DPDK-based VNFs and exploits the DPDK acceleration both in the guest and in the hypervisor. In particular, results shown in Figure 2.7 are gathered using the L2-forwarded application provided with the DPDK library (and 64B packets). The test has then been repeated by changing the number of polling threads (and hence CPU cores) associated with the vSwitch, in order to evaluate its effect on the performance of the chain. Note that, since the L2-forwarder , as any other DPDK-based process, works in polling, during this test we always have to dedicate a CPU core to the vCPU thread associated with each VM.

According to the picture, assigning more cores to the vSwitch results in better performance in case the number of cores required by the whole chain (VMs + vSwitch) does not exceed the number of cores of the server. When this happens, performance degrades until becoming unsustainable with 8 chained VMs. In fact, at this point some cores are shared among many polling threads and, for instance, it may happen that the operating system assigns the CPU to a VNF with no traffic to be processed, penalizing others that would actually have work to do.

For the sake of clarity, Figure 2.7 only reports the throughput obtained with 64B packets. However, our measurements also reveal that a chain implemented with OvS and the DPDK L2-forwarder within VMs, provides a throughput of 10Gbps in case

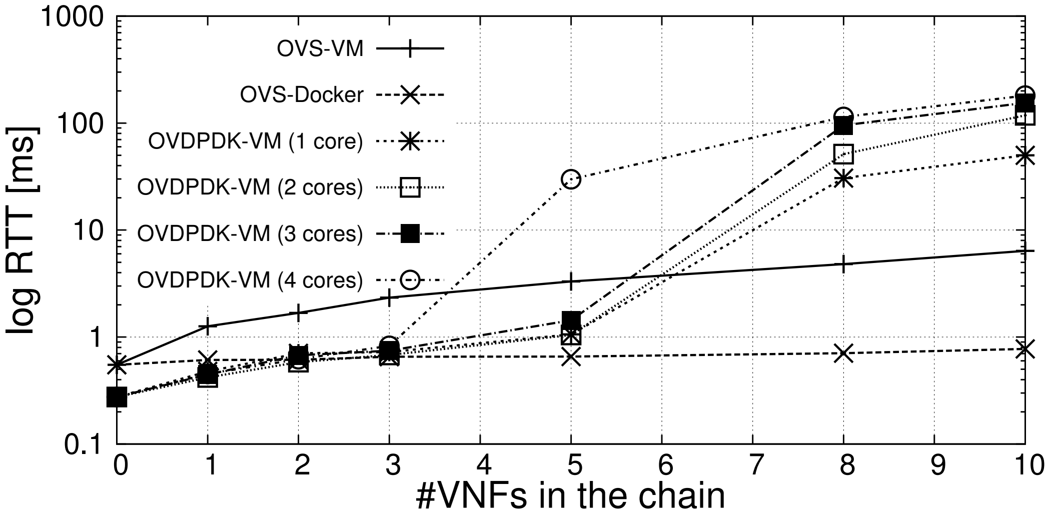


Fig. 2.8 Latency introduced by service chains implemented through different technologies.

of 700B and 1514B packets. Of course, such results are obtained only in case the chain does not require more cores than those available on the server.

A comparison between Figure 2.7 and Figure 2.5a reveals how chains implemented through DPDK-based components (i.e., OvS and VMs running DPDK applications) outperform chains not based on the DPDK library. In fact, DPDK-based modules works in polling and optimize the data transfer among each others, exploiting zero-copy as much as possible. However, as already stated above, the polling working model has the drawback of providing unacceptable throughput when the cores required exceed the number of cores available on the server.

2.4.4 Latency

In networking, the latency introduced by the system is as important as the throughput achieved. Hence, this section reports the latency measured with a single chain implemented with the technologies discussed above. In particular, for each scenario, we executed 100 `ping` between the sender and the receiver machines; results have been averaged and reported in Figure 2.8.

As expected, DPDK-based chains (OvS + DPDK-L2forwarder in VMs) provide better results, provided that the number of cores required does not exceed the number of cores of the physical machine on which such a chain is deployed. At this point, similarly to what already discussed in case of throughput, latency becomes definitely unacceptable.

Figure 2.8 also shows how Docker containers running a user-space process (i.e., the simple `libpcap`-based bridge) introduce smaller latency than VMs. This difference in performance is again a consequence of the higher overhead introduced by full virtualization with respect to lightweight containers.

2.5 Conclusion

This chapter provides a performance analysis of service chains implemented through different virtualization technologies, selected among those representing the state of the art in the NFV domain.

From the several tests carried out, we can draw the following conclusions. First, Docker containers provide acceptable results only in case of VNFs associated with a specific process, while they are definitively unsuitable for VNFs implemented as callbacks to be executed in the kernel. However, due to the low overhead introduced by lightweight virtualization, a user-space program in Docker provides better latency and almost the same throughput of callback-based VNFs (e.g., Linux bridge) run within VMs kernel. Second, OvS with DPDK enabled (when used with VMs running DPDK-based applications) provides much better performance than standard OvS. For instance, this is due to the exploitation of zero-copy when transferring packets to/from physical NICs, and to the polling model implemented by DPDK-based processes. However, the polling model requires at least a dedicated core per VM, hence limiting the usage of DPDK in servers with a reduced number of VNFs.

Given that a *single-technology-fits-all* recipe is not available in this domain, future NFV services need to heavily rely on a smart orchestration system, which should be able to select the implementation to use for the different NFs required to get advantages from them, depending on the specific service required.

Chapter 3

Designing an efficient data exchange algorithm

Part of the work described in this chapter has been previously published in [24].

3.1 Overview

As analyzed on Chapter 2 when VNFs are deployed on a single server, an incoming packet could possibly traverse an arbitrary number of VNFs before leaving the middlebox and this requires each server to include a module (usually referred as *virtual switch* or *vSwitch*) that classifies each packet to determine the path that has to traversed. All this operations cause a drop of performance as shown in Chapter 2.

This chapter proposes and evaluates an efficient algorithm to improve the forwarding of network packets between VNFs consolidated on the same server and the vSwitch, which is based on circular lock-free First-In-First-Out (FIFO) buffers managed by ad-hoc algorithms.

Existing solutions adopted to move packets between VNFs and the vSwitch are usually based on the producer-consumer paradigm. However, since in NFV it is likely that a packet goes from the vSwitch to the VNF and then back to the vSwitch, those approaches require the VNF to copy (almost) each packet from a first receiving queue into a second queue used for sending it back. Instead, the model shown on this chapter has the advantage of allowing VNFs to return back packets without any (expensive) packet copy, with consequent performance improvements.

Particularly, the proposed mechanism is designed to: *(i)* guarantee *traffic isolation* between functions, so that a function can only access the portion of traffic that is expected to flow through it, to limit the potential hazards that a malicious applications could cause providing 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. 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 Chapter is organized as follows. Section 3.2 describes related other solutions able to exchange data among different software components. Section 3.3 details the main operations of the designed algorithm and how it communicates with the different modules. Section 3.4 shows some implementation choices used to increase the performance of the algorithm. Section 3.5 contains a validation of the algorithm both in ideal conditions and in real scenarios. Finally, Section 3.6 concludes the chapter.

3.2 Related work

The efficiency of FIFO queues implemented as *lock-free* has been already analyzed in the past. For instance, [25] and [26] propose different type of lock-free algorithms for FIFO queues that are managed as non-circular linked-lists. Other proposals can be found in [27] and [28], that also require to create a pool of pre-allocated memory slots. However, the related solutions are usually based on uni-directional flows of data according to the *producer-consumer* paradigm, which is not the best solution for managing the bi-directional data flows that is strongly present on the case of virtualized environments. In fact, in these environments, a packet typically goes from the virtual switch to the VNF and then goes back. 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.

A similar solution can be found in the Intel DPDK library [20] and in [29], which contains algorithms designed to operate in contexts where many processes can concurrently do operations on a *shared* buffer. However, those proposals cannot guarantee isolation between VNFs because they use a unique shared buffer. Similar considerations can be made for ClickOS [30, 31] (based on the VALE virtual switch [13]) and NetVM [32], 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 compromising traffic isolation requirement. MCRingBuffer [33], instead, defines an algorithm to exchange data between a producer and a consumer executing on different CPU cores, which is particularly interesting for its efficient implementation of memory access patterns; in fact, part of those techniques were reused on the implementation (Section 3.4).

3.3 The data exchange algorithm

The implementation of an efficient virtual functions chains within a single middlebox requires a fast and efficient mechanism to transfer data between the vSwitch and the VNFs. This requirements is translated into the necessity of a dedicated data dispatching algorithm, which among the ones having the greatest impact on the overall system performance. This section provides a description of the designed algorithm.

In the context of the algorithm, the *Master* is the module that acts as the vSwitch, while the *Workers* are the VNFs. Moreover, a *token* is a generic data unit exchanged between the Master and the Workers. The token can be a packet for the NFV use case, but the proposed algorithm can be actually used to exchange any kind of data, according to the specific use case implemented.

3.3.1 Algorithm overview

As shown in Figure 3.1, the proposed data exchange algorithm is composed of a set of lock-free ring buffers; in particular, the Master shares two different buffers with

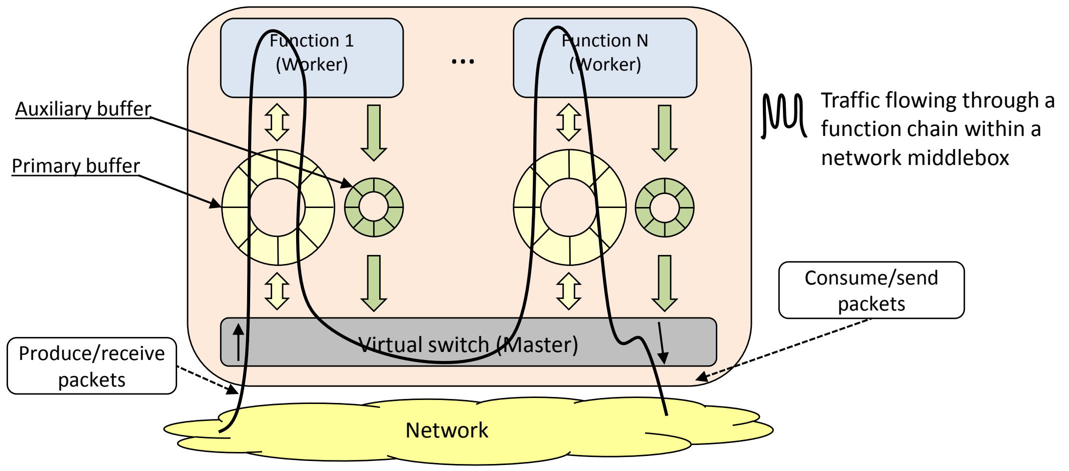


Fig. 3.1 Deployment of the algorithm within a middlebox.

each Worker, which are managed through different (but not independent) parts of the same exchange algorithm.

The intuition behind the algorithm, which is based on the NFV use case, is the following. VNFs are pieces of software operating on the data plane of the network, which mainly process pass-through packets. In fact, VNFs receive packets from the vSwitch and, in the vast majority of cases, forward them back to the vSwitch itself with minimal (or no) changes (functions as NAT or IDS), allowing packets to continue their journey towards the final destination. To efficiently support pass-through data, the model defines the *primary buffer* shown in Figure 3.1, which has the peculiarity of allowing tokens to be moved both from the Master to the Worker, and then back from the Worker to Master, without requiring any (expensive) copy of data in the Worker itself.

Avoiding to copy each packet in each traversed VNF can save in fact many CPU cycles and consequently improve the performance of virtualized functions chains. Notably, in addition to VNFs operating on pass-through data, the primary buffer also supports functions that need to drop packets (e.g., firewall), which should then not be sent back to the vSwitch after their processing in the VNF.

Some network functions may need to generate 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 function in the chain. Similarly, another function may modify 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. Since network applications forward most of the traffic without generating new packets, the primary buffer is designed to be as simple as possible for the sake of speed, and then use a new second lock-free ring buffer, i.e., the *auxiliary buffer* of Figure 3.1, to support Workers that can possibly generate new tokens as a consequence of the data received from the Master. It is worth noting that this second buffer is unidirectional and it is only used by the Worker to provide "new" data to the Master.

Since VNFs may belong to different tenants, a network function must only accesses to the proper portion of network traffic; to guarantee this property the model uses a different pair of buffers per Worker in order to guarantee traffic isolation among them, as this ensures that a Worker can only access packets that are expected to flow through the Worker itself.

Each buffer slot (both primary and auxiliary) includes some flags in addition to the real data, these flags are used to identify the content of the slot itself; more details will be presented in the next sections.

3.3.2 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 middlebox (and then the Master itself) is supposed to be traversed by a huge amount of traffic; hence, a blocking model would be too penalizing requiring an interrupt-like mechanism to start the Master when there are new data. This could significantly drop the performance with high packet rates [34]. In fact, interrupt handling is expensive in modern superscalar processors because they have long pipelines and support out of order and speculative execution [35], 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 is 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.

3.3.3 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 primary 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 processed 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 the NFV use case (with a maximum of N packets per cycle), 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 1 Executing the Master

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

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, we avoid to wake up the Worker for each single token that needs to be processed, which results in performance improvement 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.

Algorithm 2 The Master writing data into the primary buffer

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

Our algorithm avoids the starvation of tokens sent to a Worker (which may happen especially when the system is in underload conditions) thanks to a timeout event that wakes up the Worker even if the above-mentioned 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. This information is carried by the shared variable `workerStatus`, which is set to

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

```
1: **Procedure** master.checkForOldData()
2:
3: **if** buffer.size() $>$ 0 **and** (workerStatus $\neq$ SIGNALED) **and**
   ((now() $-$ timeStamp) $>$ TS_THRESHOLD) **then**
4:     workerStatus $\leftarrow$ SIGNALED
5:     wakeUpWorker()
6: **end if**
```

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 from the primary buffer 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.

Algorithm 4 The Master reading data from the primary buffer

```
1: **Procedure** master.readDataFromBuffer()
2:
3: **if** buffer.size() **then**
4:     **if** M.consIndex $\neq$ W.prodIndex **then**
5:         timeStamp $\leftarrow$ now()
6:         **while** M.consIndex $\neq$ 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**
```

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. 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, although 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 the `W.prodIndex` has still to be updated, 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 3.2 depicts the status of the primary buffer[1] and the indexes used by the algorithm in four different time instants. In Figure 3.2(a) the buffer is empty, and then all the indexes point to the same position. Instead, in Figure 3.2(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 3.2(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 3.2(d) this threshold is passed and the Master has already consumed some data.

[1]For the sake of clarity, the figure represents the shared buffer as an array instead of a circular FIFO queue.

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
```

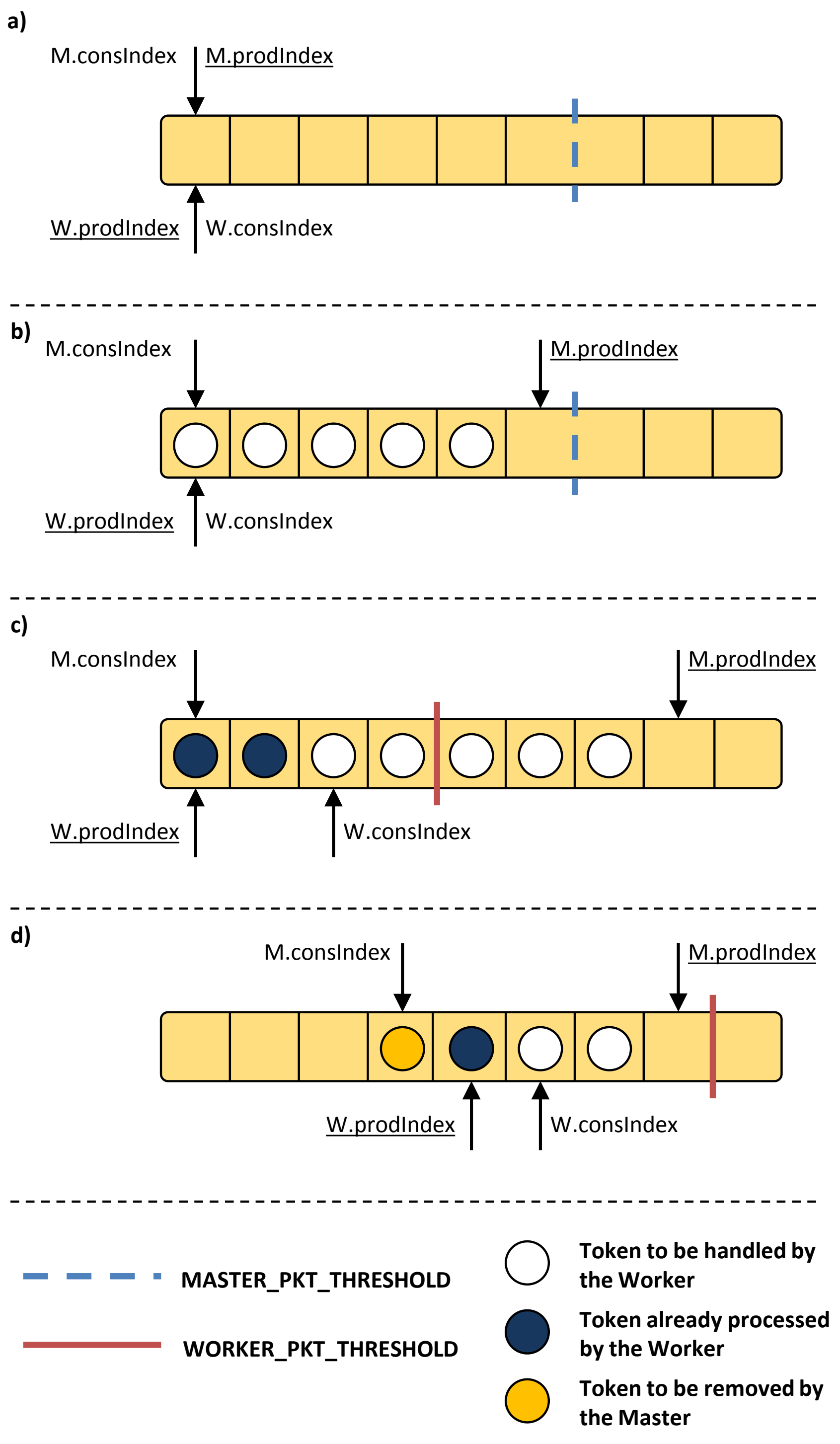


Fig. 3.2 Run-time behavior and indexes of the algorithm.

3.3.4 Extended algorithm: handling worker-generated data

The algorithm also supports Workers that may need to generate *new* token as a consequence of the token just received from the Master; however, 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 and also modify its size) pass-through tokens in the primary buffer or, at most, it can drop these tokens.

Then, in case new data have to be provided to the Master, the Worker can use the auxiliary buffer. 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 to the master 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)
```

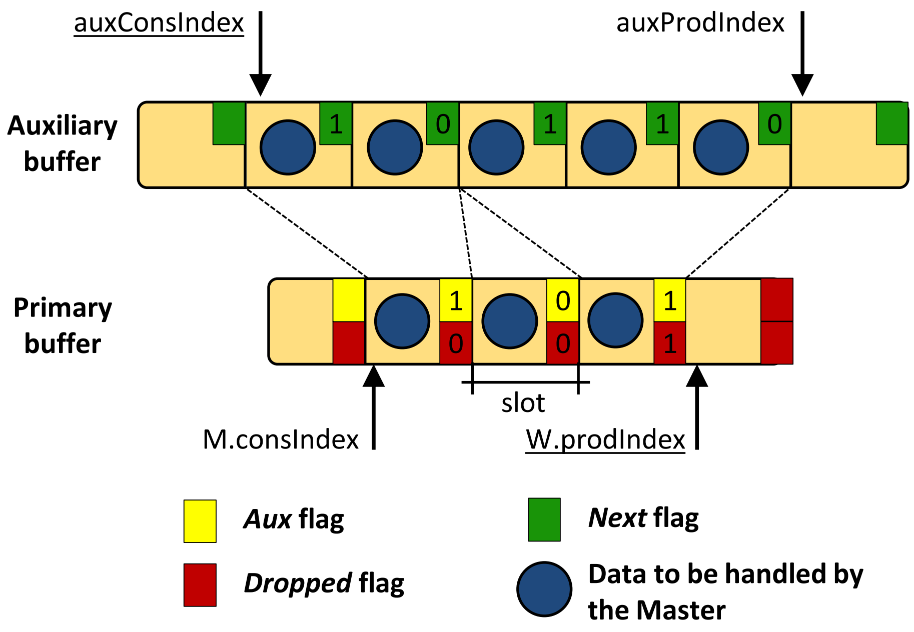


Fig. 3.3 Binding primary buffer - auxiliary buffer.

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.

Algorithm 7 The Master reading data from the auxiliary buffer

```
1:  Procedure master.readDataFromAuxBuffer()
2:
3:  while true do
4:      master.consumeData(auxBuffer.read(auxConsIndex))
5:      if not auxBuffer.next(auxConsIndex) then
6:          auxConsIndex++
7:          break
8:      end if
9:      auxConsIndex++
10: end while
```

Figure 3.3 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 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.

3.4 Implementation choices

This section presents the choices done for the implementation of the prototype to improve the performance and the scalability of the algorithm.

Private copies of shared variables. The presented algorithm, as the many derived from the producer-consumer problem, also needs 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 (the two processes never try to write the same variable at the same time so no contention can occur), but, as shown in MCRingBuffer [33], the actual implementation on real hardware can introduce additional issues. In fact, when two CPU cores works on cached variables if one core modifies the content of a variable that is shared, the entire cache line (64 bytes long on the modern Intel architectures) of the other 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 the algorithm, this problem occurs for `M.prodIndex`, incremented by the Master and read by the Worker, and for `W.prodIndex`, incremented by the Worker and read by the Master. However, the algorithm is robust enough to operate correctly even if those variables are not perfectly aligned. As a consequence, the code of the algorithm implements a cache line separation mechanism (similar to MCRingBuffer [33]), which consists in storing each shared variable (possibly extended with padding bytes) on a different cache line storing the data on a local private variable while the task is being performed.

Alignment with cache lines. When a cache miss occurs, the hardware introduces a noticeable latency related to the necessity to update the cache with the latest data, 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), the 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. So, 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. The implementation emulates the timestamp, which 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, the 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 and data 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 in case at least `MASTER_PACKET_THRESHOLD` packets are queued in the primary buffer, or in case some packets are waiting for long time and then the timeout expired. Although POSIX semaphores are implemented in kernel space, their impact on performance 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 in fact 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[2], by design the 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.

3.5 Experimental results

In order to evaluate performance and scalability of the data exchange algorithm described in Section 3.3, we carried out several tests on our prototype implementation (that follows the implementation choices described in Section 3.4) running on a workstation equipped with an Intel i7-3770 @ 3.40 GHz (four CPU cores plus hyperthreading), 16 GB RAM, 16x PCIe bus, two Silicom dual port 10 Gigabit Ethernet NICs based on the Intel x540 chipset (8x PCIe), and Ubuntu 12.10 OS,

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

kernel 3.5.0-17-generic, 64 bits. In all tests, 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 tests done to evaluate the chain performance are similar to the ones of Chapter 2. 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 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 first present 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. In these testing conditions, Section 3.5.3 compares our algorithm against two existing approaches based on the traditional producer/consumer paradigm, which are typically used to exchange packets between the vSwitch and the network functions consolidated on the same server. Particularly, the comparison shows the advantages deriving by both the absence of data copy in the Worker and the blocking operating mode of the Worker itself. Finally, Section 3.5.6 presents 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 [36] to read/write packets from/to 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.

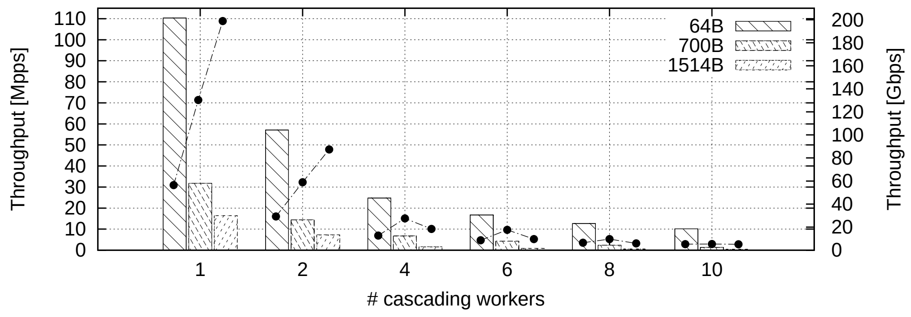
3.5.1 Single chain - Throughput

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, which reports the throughput in Gigabits per second.

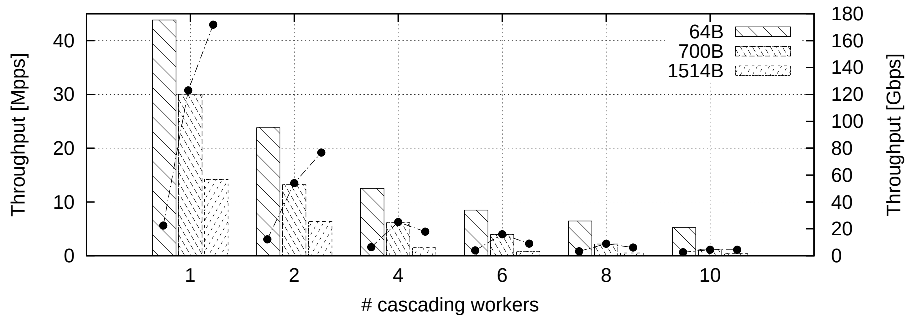
Figure 3.4 shows the throughput offered by the function chain in different conditions. These numbers depend both on the design aspects of our algorithm (e.g., no data copy in the Worker, polling model in the Master, blocking model in the Worker, etc.), as well as on the implementation choices we did when implementing the prototype (e.g., data aligned with cache lines, private copies of shared variables, etc., as detailed in Section 3.4). For instance, 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 of reserving the most part of the CPU power to the Workers, hence limiting the Master to a single CPU core (Section 3.4 - threading model).

Figure 3.4a shows the throughput that could be achieved in ideal conditions, that is: *(i)* with dummy Workers, i.e., Workers 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 3.4b 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 test shows that performance is reduced compared to Figure 3.4a for two reasons: (*i*) the higher number of cache misses generated by the Workers when accessing to the packet content, and (*ii*) the additional processing time spent by the Workers for completing their job.

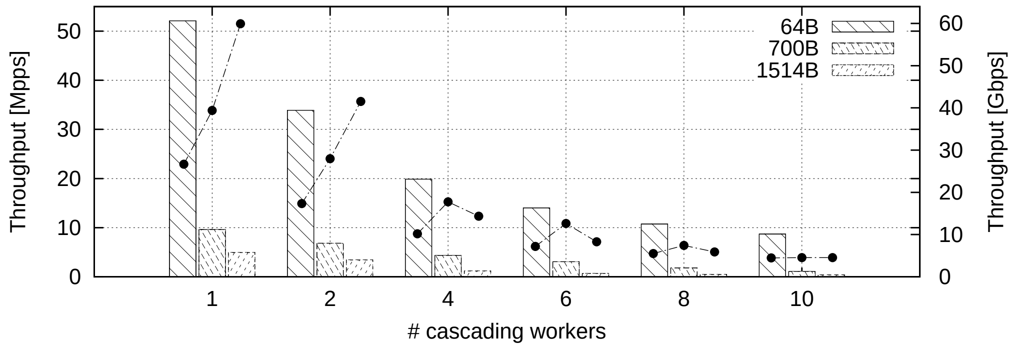
Next tests consider a scenario where the input data for the chain is stored in a buffer containing 1M packets, thus emulating a real middlebox that receives traffic from the network. In particular, Figure 3.4c refers to a scenario with dummy Workers such as in Figure 3.4a and shows how an apparently 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 3.4a, 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.



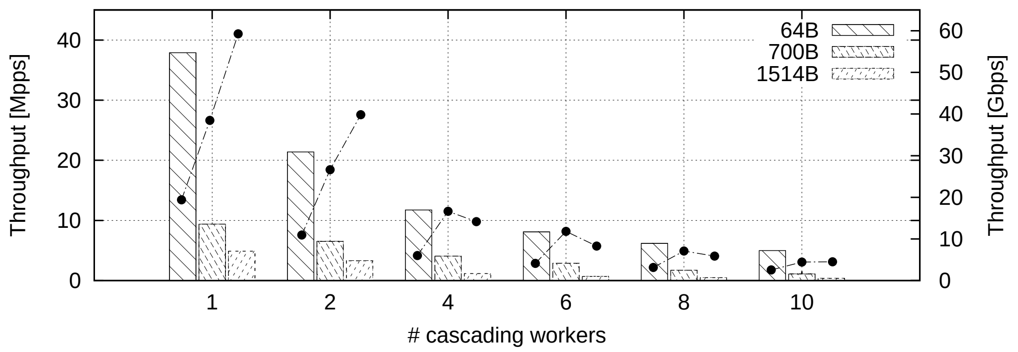
(a) Dummy Workers and a single packet in memory.



(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.

Fig. 3.4 Throughput of a single function chain.

Finally, Figure 3.4d depicts a realistic scenario where Workers access the packet content (such as in Figure 3.4b), 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 3.6 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 3.4d. 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.

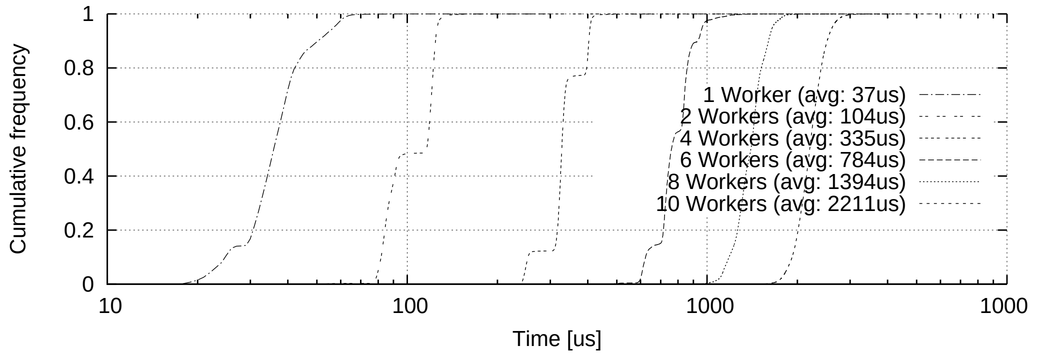
3.5.2 Single chain - Latency

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.

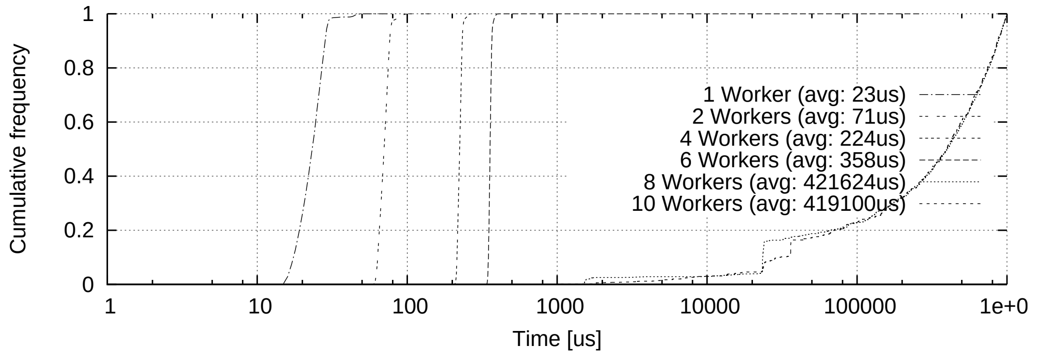
Figure 3.5a shows the latency of 64B packets when traversing a function chain consisting of a growing number of Workers, in case of real Workers and 1M packets in memory. As expected, the latency increases with the length of the chain; however its value is definitely reasonable for most of networking applications, reaching an average value of about 2.2ms in case of 10 cascading Workers, being far less with shorter (and more realistic) chains.

3.5.3 Single chain - Comparison with other algorithms

This section aims at comparing our data exchange algorithm with two other approaches that could be used to exchange packets between the Master and the Workers, and which represent the baseline algorithms used to evaluate the improvements (in terms of performance) brought by our research. Particularly, the comparison aims at



(a) Our algorithm.



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

Fig. 3.5 Latency introduced by the function chain with a growing number of cascading Workers.

validating the advantages of two important aspects of our algorithm: the absence of a data copy in the Worker, and the blocking operating mode of the Worker itself.

In this respect, the algorithm cannot directly be compare with existing prototypes used in NFV such as VALE [13] and OVS [8], 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 we carefully implemented them by using, whenever applicable, the guidelines listed in Section 3.4 (e.g., shared variables on different cache lines, private copies of shared variables, and more).

The first baseline algorithm 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. In this case, similarly to our algorithm, only the Master operates in polling mode, while the Worker wakes up when there are packets to be processed. The second baseline algorithm closely follows the processing model suggested by Intel in the DPDK library [20]. Also in this case two buffers (again 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 memory region and never moved between the components of the functions 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 has been selected as a baseline algorithm to be compared against our proposal because nowadays it represents the “standard” way to move packets in network function chains.

The baseline algorithms are executed in realistic conditions, namely with Workers accessing packets and 1M packets in memory; therefore, obtained results should be compared with numbers reported in Figure 3.4d shown also in Figure 3.7a.

As expected, the throughput of the chain drops of about 30% when unidirectional buffers are used, as shown by comparing Figure 3.7a and Figure 3.7b. 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 the baseline algorithm requires one additional data copy in the Worker.

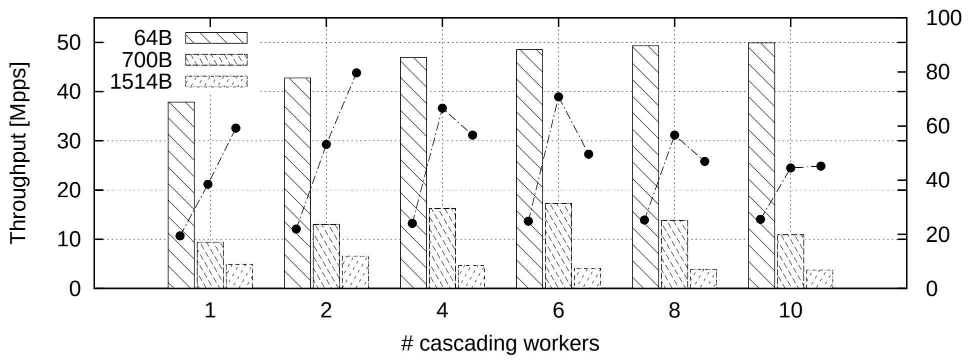


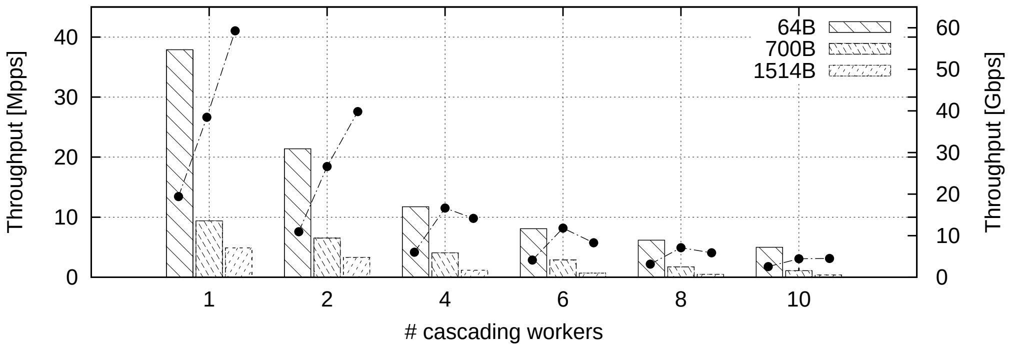
Fig. 3.6 Internal throughput of the function chain, with real Workers and a 1M packets in memory.

Instead, the second baseline algorithm 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 3.7c with Figure 3.7a. 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.

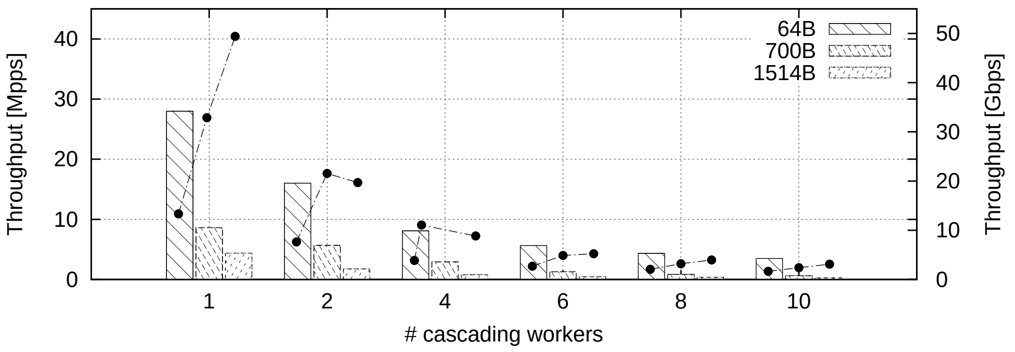
The second baseline algorithm has also been evaluated in terms of latency introduced on the flowing packets. Similarly to what happens for the throughput, it outperforms our proposal when the number of running jobs is less than the number of CPU cores, as evident by comparing Figure 3.5a and Figure 3.5b. For instance, six chained Workers introduce an average latency of 358μs, against the 784μs obtained with our algorithm. Instead, in case of more Workers, the average latency of the baseline algorithm 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.

3.5.4 Single chain - Other tests

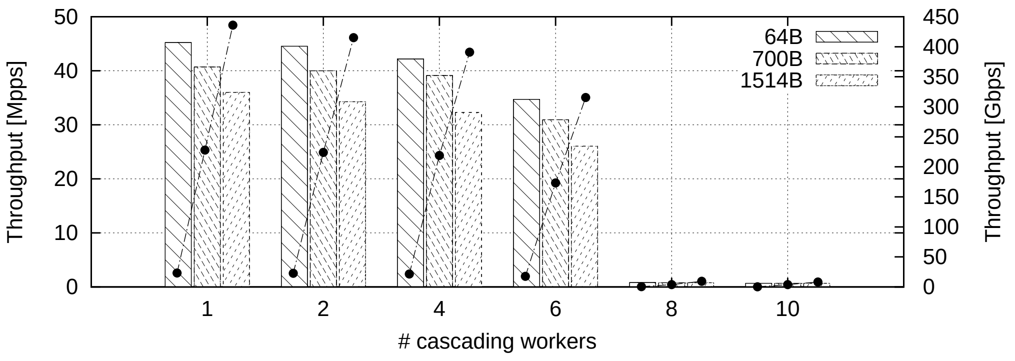
Additional tests have been performed in order to evaluate some other aspects of the system.



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



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



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

Fig. 3.7 Throughput of a single function chain when other data exchange algorithms are used.

Threads vs. Processes

Threads appear more convenient than processes because they share the same virtual memory space, while processes 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 cache efficiency and the TLB behavior and, consequently, on the overall performance of the system. With respect to the former, the same physical memory address shared in two processes lead to two virtual addresses, which requires two entries in the L1/L2 caches[3]; threads, instead, have the same virtual address, hence potentially allow the same cache line to be used by different threads. With respect to the TLB, as the same (virtual) address space is present in many threads, the number of entries in the TLB is reduced as well. 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 NFV node, the Master and all the Workers should be 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

[3]The L3 cache operates with physical addresses.

Workers executing a deep packet inspection algorithm based on a Deterministic Finite Automata (DFA), which requires a huge amount of memory to store 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.

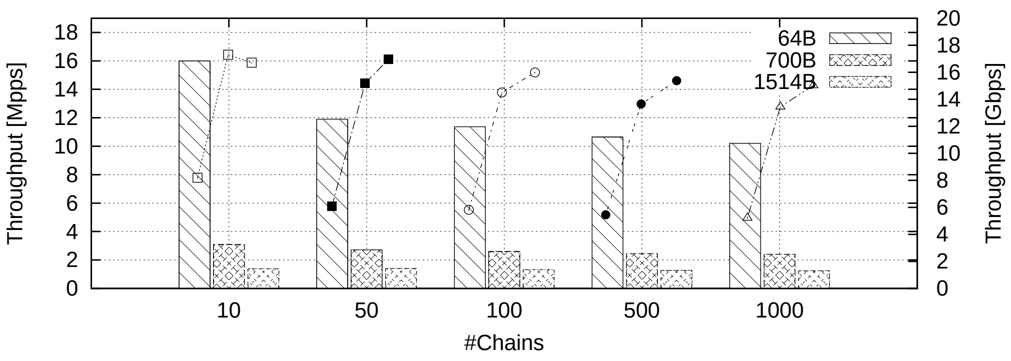
3.5.5 Multiple chains

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

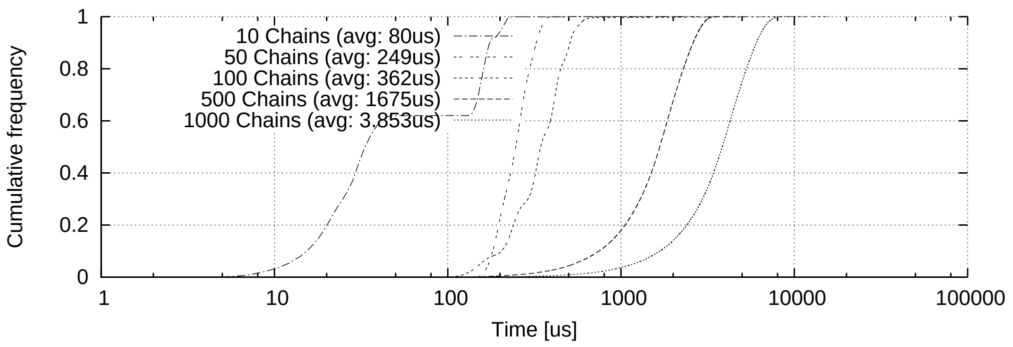
Data read from the initial memory buffer (containing 1M packets) is provided, in a round robin fashion, to a growing number of function chains, each one composed of two Workers. During the tests, each Worker is involved in two chains meaning that, when 1000 Workers are deployed, packets are spread across 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 3.4).

Obtained results are shown in Figure 3.8; as in the previous tests, these numbers are due to the combined effect of the choices we did when designing our algorithm (Section 3.3) and of the implementation guidelines followed to efficiently implement the prototype (Section 3.4).

Figure 3.8a 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 3.8b 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.

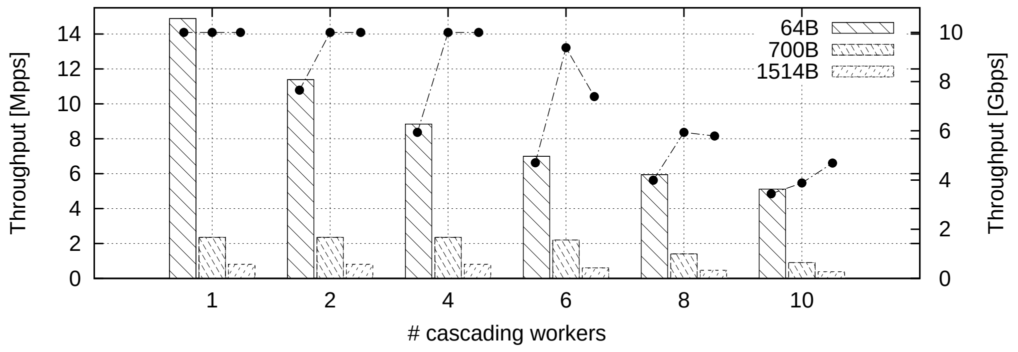


(a) Throughput.

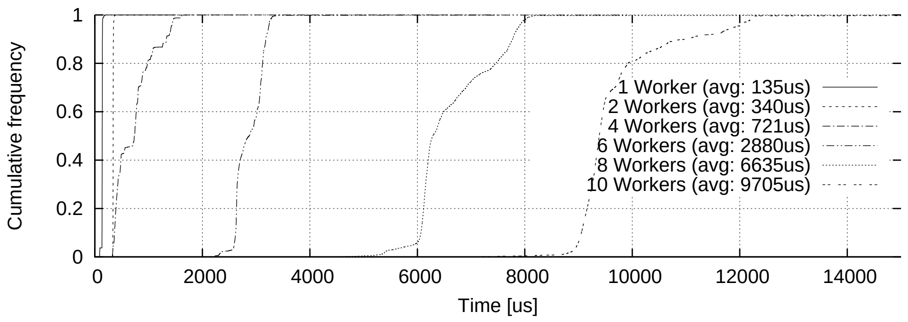


(b) Latency.

Fig. 3.8 Results with a growing number of function chains running in parallel, each one spotting two Workers in cascade.



(a) Throughput.



(b) Latency.

Fig. 3.9 Results with a function chain of growing length, with the Master accessing to the network.

3.5.6 Network tests

This section evaluates our algorithm in a real deployment scenario, i.e., when executed on a workstation that receives/sends traffic from the network. In this case the overall performance of the system depends on the algorithm, on the implementation choices done when developing the prototype, as well as on 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 3.4d, is depicted in Figure 3.9a. 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 is even slightly better than what was obtained in Figure 3.4d 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.

Figure 3.9b shows the cumulative distribution function of the latency introduced by network function chains of different length when traversed by 64B packets. Those numbers, obtained by sending packets at the same rate shown in Figure 3.9a, measure the time between the instant in 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 then 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. Particularly, reported numbers also include the time that the packet spends in the input buffer before being picked up and sent through the chain by the Master, because of its batch-based reading mode. Our measurements demonstrate that the latency, albeit still acceptable, is about 4-5 times higher than in Figure 3.5a.

3.6 Conclusion

This chapter showed an efficient way to move data between 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.

Finally, performance and scalability of the proposed solution have been evaluated by means of a wide range of experiments made on a real implementation, comparing also different possible implementation choices; obtained results confirm the effectiveness of the proposed approach.

Chapter 4

Prototyping a dedicated network function for resource-constrained devices

Part of the work described in this chapter has been previously published in [37].

4.1 Overview

This chapter presents a specific native application for the filtering of HTTP URLs on resource constrained devices like home gateways, which leverages a distributed software architecture.

The architecture of residential gateways is characterized by special purpose hardware chips that forward packets at high speed at the data link layer, and other hardware components, such as CPU and central memory, used for other operations that require more sophisticated processing. Their processing capabilities are often underutilized and they could be leveraged by Internet access service providers to offer additional service to their customers. However, the limited computing and memory resources that residential gateways have by design make the implementation of new features working at wire-speed very challenging, particularly when complex operations such as parsing packets up to the application layer (a.k.a. Deep Packet Inspection (DPI)) are involved.

This is the case for many critical modern filtering applications, such as malware protection, corporate policy enforcement, parental control, advertisement block, that are based on inspection and filtering of Uniform Resource Locators (URLs). In fact, users access and exchange content mostly through mobile apps and web applications, both based on HTTP, which uses URLs to identify data objects to be transferred. Implementing this services on the residential gateway would require matching URLs against large, dynamic blacklists, which far exceeds the limited hardware capabilities of this category of devices. For example, an effective parental control service, which is a valuable offer to residential customers, is based on a very large database of URLs that cannot be stored in the limited memory of common residential gateways (usually in the order of tens of MB). An additional challenge comes from the fact that the database must be frequently updated. Last but not least, URL matching cannot be limited to the hostname, but the entire URL should be considered because the same web server can host both appropriate and inappropriate or malicious pages.

The presented solution, named *U-Filter*, integrates a URL filtering service in a resource constrained device, such as a common residential gateway, leveraging a distributed software architecture. A remote *policy server* in charge of keeping the URL database provides a fast API that can be accessed through the network in order to establish if a request for a specific URL is allowed. It is reasonable that the above mentioned server is operated by a service provider (or the network service provider) and can rely on powerful hardware resources to serve multiple residential gateways with minimal response time. However, this architecture does not necessarily require the network service provider awareness and collaboration. The presented solution greatly alleviates the load on each residential gateway, even though it must still perform a limited form of DPI on outgoing packets to extract the URL from every HTTP request, and afterwards query the server in order to determine the policy that must be applied. We adopt specific techniques to optimize this task and limit the latency introduced by the client-server interaction, striking a balance between the load they introduce and the limited resources available in residential gateways.

The chapter is organized as follows. Section 4.2 presents the architecture of U-Filter, describing the design principles that led to our solution and the optimizations used to provide real-time policy enforcement on resource-constrained devices. We validate U-Filter in Section 4.4 through various experiments showing the impact on the user experience. Section 4.5 presents the state of the art of HTTP-level policy enforcement and Section 4.6 concludes the chapter.

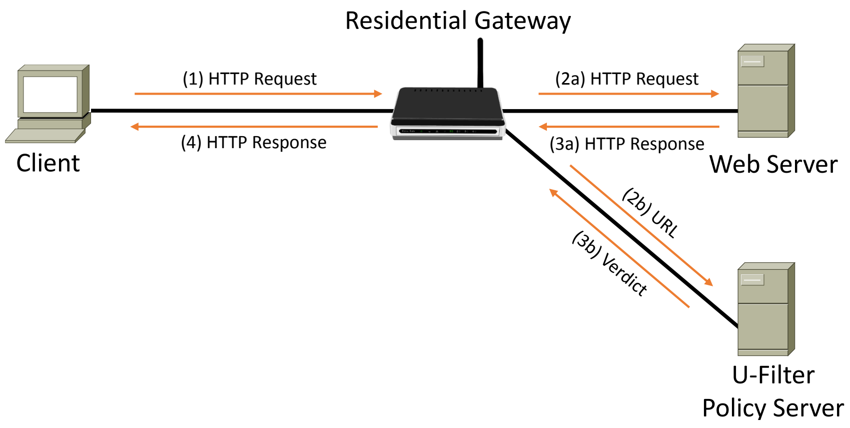


Fig. 4.1 U-Filter workflow.

4.2 Overall architecture

4.2.1 Operating principles

A typical deployment scenario of U-Filter is presented in Figure 4.1. A user surfing the web generates many HTTP requests that transit through her/his residential gateway. These requests are analyzed by U-Filter, which extracts the requested URL through a lightweight DPI algorithm. This allows to process line rate traffic with a small overhead for the residential gateway. Afterwards the HTTP request is released and can continue its journey towards the web server, while the URL is simultaneously sent to the policy server that provides the policy to enforce. This policy is enforced by U-Filter on the packet carrying the HTTP response by either blocking or allowing it.

4.2.2 Architecture overview and design principles

The prototype has been built around three main objectives. First comes **flexibility**, as it is essential to be able to enforce effective protection to end users in a prompt response to newly discovered threats. Second is **efficiency** since the system is targeted to resource-constrained devices. Third, we took care of ensuring an excellent **user experience**, hence limiting the impact of the system in terms of possible additional latency when inspecting traffic to apply filtering policies. The above high-level objectives have translated in the following four design choices.

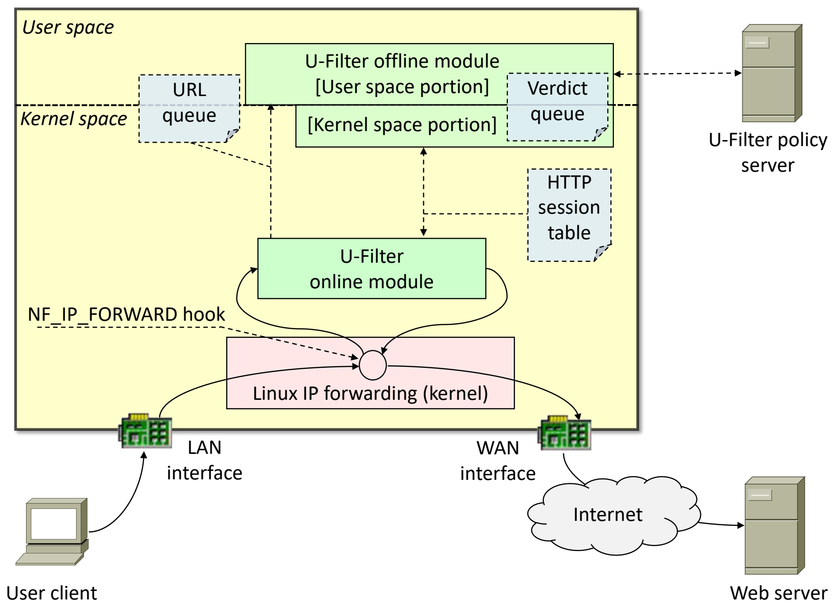


Fig. 4.2 U-Filter architecture.

Three-tier processing architecture

As shown in Figure 4.2, U-Filter includes (*i*) an *online module*, which sits on the data plane of the router and is mainly in charge of identifying (and extracting) requested URLs from network traffic (more details in Section 4.3.2) and apply the policy decisions on the return traffic, (*ii*) an *offline module* that queries a remote policy server to know whether such URL should be allowed or not (described in Section 4.3.3), and (*iii*) a *remote server* that implements the complex protection logic and returns a boolean value with the result of the classification, i.e., if the corresponding HTTP session handled by the online module has to be allowed or the URL is malicious and the response has to be blocked. The first two modules are built with efficiency in mind, while the latter allows to achieve the required flexibility.

The policy server can be executed on a remote host (or on a cluster of hosts for performance reasons), as its only interaction with the rest of the system is through a query/response protocol. A single policy server can be queried by offline modules running on multiple (remotely distributed) residential gateways.

Decoupling policy verification from HTTP operation

As introduced in Section 4.2.1, policy compliance is verified without holding outgoing packets on their ride towards the final destination. This solution makes the system more complicated but much more efficient. In fact, keeping the HTTP request on hold until the arrival of the response from the policy server would add additional delay to the HTTP communication, increasing the Round Trip Time (RTT) of the HTTP connection and hence affecting the user experience. Vice versa, the U-Filter offline module checks the requested URL with the policy server during the normal HTTP RTT. A temporary entry in an HTTP session table is created by the online module in order to possibly hold a response from the web server received *before* the result of the compliance check arrives from policy server. While this allows packets to travel through the Internet also if they are part of a session that shall be stopped, the answer from the web server never reaches the user, effectively preventing possible unwanted data to reach the user's host.

Efficient memory usage

Efficient memory usage is a key problem because of (*i*) the limited amount of memory usually available in current residential gateways, and (*ii*) the bad effects in terms of CPU cache pollution when large memory structures (with sparse access patterns) are used. Several implementation choices have been adopted to ensure that memory is used efficiently. According to the best practice for kernel module development, all the memory used by the online U-Filter module is allocated at startup in order to avoid costly memory allocations at run-time, and the structures that are used for the communication between online and offline modules are shared using the proper primitives for mapping memory between kernel and user space for better memory efficiency. Furthermore, all the helper structures detailed in Section 4.3.1 make use of contiguous memory areas in order to improve data locality and, as a result, CPU cache efficiency, except for the packets that may need to be held temporarily by U-Filter (while waiting for an answer from the policy server), which have been allocated by other portions of the kernel and therefore are not under our control. Finally, the usage of additional memory is kept at minimum: (*a*) the data structure dedicated to the session table defines a "default" behavior that avoids storing accepted sessions, and (*b*) the number of packets held by the router while

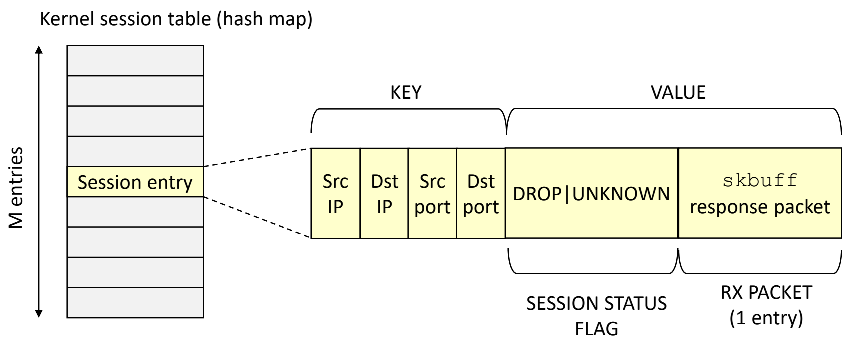


Fig. 4.3 HTTP session table, shared between online and offline modules.

waiting for the answer from the policy server is limited to, at most, *one per session*, hence further reducing memory requirements.

Per-packet operation

This is known to be much more efficient than per-TCP session processing while, at the same time, reducing the latency required to extract application level information (namely URLs). In fact, the former can be based directly on the very efficient packet processing primitives available in the Linux kernel through the `netfilter` framework, instead of requiring a full-blown HTTP proxy, whose complexity is so high to make a kernel implementation problematic. Therefore, an additional overhead is added for moving all packets from kernel to user space, where a proxy is usually located, and then back to kernel for their transmission on the output interface.

4.3 Implementation details

4.3.1 Key data structures

The online and offline modules exchange data using three shared structures, as shown in Figure 4.2: (*i*) a hash map for the status of the policy for a given session, (*ii*) a queue for the URLs that have to be send to the policy server and (*iii*) a queue with the verdict received from the policy server. Each of the data structures is described in detail in the reminder of this section, while their usage will be discussed in the following sections.

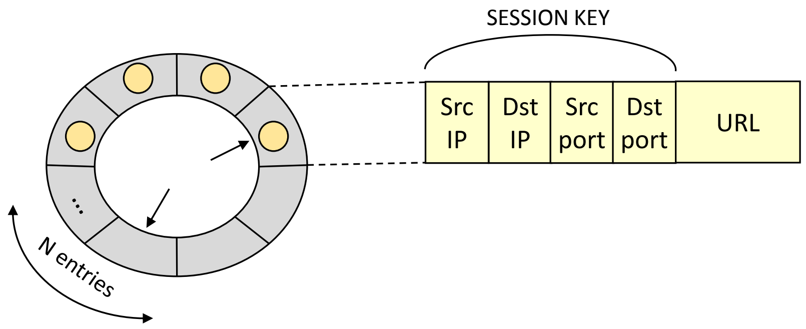


Fig. 4.4 URL queue, shared between the online module and the offline module user space process.

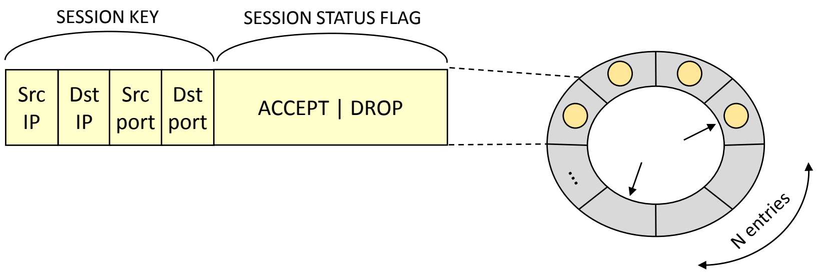


Fig. 4.5 Verdict queue, shared between the offline module kernel thread and user space process.

The *HTTP session table* (shown in Figure 4.3) stores data regarding pending sessions. An HTTP session is considered pending when the HTTP request has been received, but either the HTTP response from the web server or the decision from the policy server are yet to be received. The hash map implementing the HTTP session table is allocated in kernel space and is shared between the online and offline module because the former needs to know (when an HTTP response arrives) whether a decision for an URL has been received, while the latter needs to know, when the verdict is available, whether an HTTP response is already waiting. An entry in the HTTP session table can be deleted as soon as both the HTTP response and the verdict from the policy server have been received.

The *URL queue* (shown in Figure 4.4) is shared between the online module and the offline module user space process, while the *verdict queue* (shown in Figure 4.5) is shared between the kernel thread and the user space process of the offline module. The two queues are managed according to a FIFO policy and the access to each queue is implemented with two pointers, pointing respectively at the first *free* and the first *full* slot.

To correlate data in different data structures, an entry always contains a key made by the 4 tuple identifying the TCP session (later referred as session ID):

$$(Source\,IP, Destination\,IP,\ Source\,TCP\,port,\ Destination\,TCP\,port)$$

The addresses are the ones present in the HTTP request and are inverted in the corresponding HTTP response.

An entry in the URL queue contains also the URL that should be checked with the policy server, while an entry in the verdict queue contains a session status flag that assumes either `ACCEPT` or `DROP`, according to the policy to enforce. The URL is stored in some pre-allocated memory whose size allows containing a full-length HTTP payload (i.e., 1460 bytes), in order to avoid memory allocations at run-time. On the other hand, an entry in the HTTP session table stores as value a session status flag and a void pointer to a packet (`skbuff` structure, allocated by the operating system). The use of this pointer is detailed in Section 4.3.2. Differently from the verdict queue, the session status flag in the HTTP session table can assume either `UNKNOWN` or `DROP`. In fact, entries corresponding to an `ACCEPT` policy are deleted as soon as the verdict is available in order to reduce the size of the hash table. Thus, in the HTTP session table the absence of an entry is considered as an `ACCEPT` policy.

As a further optimization to reduce the allocated memory, in our prototype the TCP session ID uses only the last byte of the source IP address, instead of the entire 4 bytes address, with no impact on the system proper execution. This optimization is correct in our environment, since domestics LANs usually adopt a 24 bits subnet, therefore all the clients have the same value for the first 3 bytes of the IP address. In general this is not valid for every deployment, hence the optimization should be adapted to the specific addressing plan in use.

4.3.2 Online module

The online module sits on the data path by intercepting all the traffic forwarded by the router through a callback registered on the `NF_IP_FORWARD` `netfilter` hook. As shown by the workflow depicted in Figure 4.6, most of the processing occurs when an HTTP *request* or *response* is detected. For each packet, the module first locates the beginning of the TCP payload and then checks if that packet can be considered the *first segment* of an HTTP request or response by matching the beginning of the TCP payload against a few simple text strings, namely an HTTP method (i.e., GET, POST, PUT, etc.) in case of a request or a version string (i.e., HTTP/1.0 or HTTP/1.1) in case of a response. This classification method is far more reliable than checking the transport-layer port number, as investigated in [38]. All other packets, namely HTTP packets that are not the first of the request/response message (hence, do not match the signature), as well as non-HTTP traffic, are left to continue their way as the online module returns `NF_ACCEPT` to `netfilter`. Notably, since *all* TCP packets containing a valid payload are matched against the signature, this algorithm is able to intercept *all* the HTTP requests/responses that are issued within a connection in HTTP 1.1 persistent mode, not only the first one, as well as within HTTP connections terminated on a non-standard TCP port. This algorithm could raise concerns about the cost of inspecting all packets, as general DPI techniques are normally demanding in terms of computing resources. However, our algorithm does not perform a full-blown DPI with full parsing of all protocol headers and their fields. Instead, it performs a lightweight parsing to locate the beginning of the TCP payload and a *string checking* (instead of regular expressions) just on the *initial bytes* of the payload. In fact, our experimental validation (Section 4.4.3, Figure 4.11) confirms that the online module does not introduce noticeable overhead in the traffic processing.

In case of an HTTP request, the URL is extracted and sent to the offline module by pushing a new entry in the (shared) URL queue (Figure 4.4), which includes the TCP session identifier to later match the verdict from the policy server with the corresponding HTTP session. A new entry is also created in the HTTP session table; as shown in Figure 4.3, it includes the TCP session identifier (as a key), a session status flag that is marked as `UNKNOWN`, and an additional field that is left empty. Afterwards the packet is allowed to be forwarded by returning `NF_ACCEPT` to `netfilter`.

When an HTTP response is received, the module checks the status in the HTTP session table and acts according to the three possible scenarios:

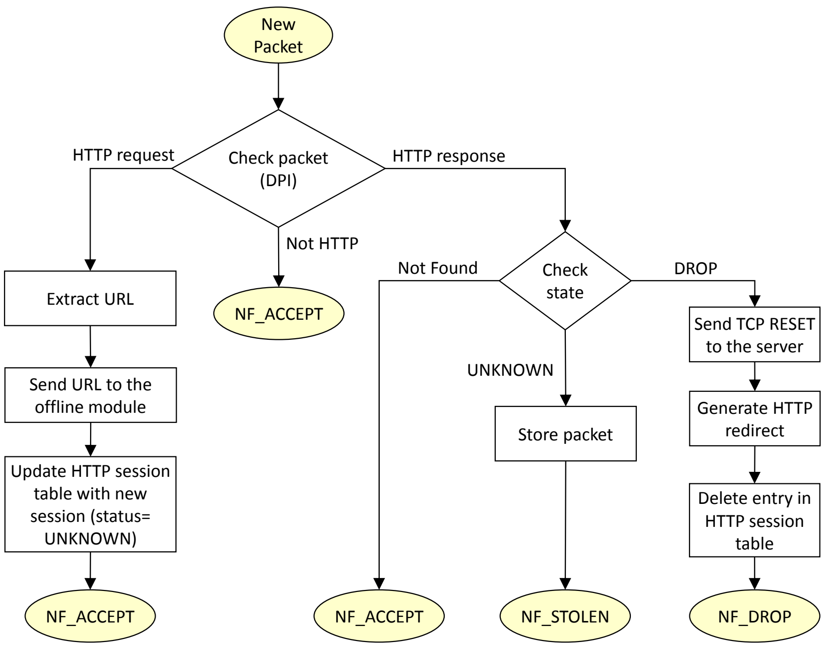


Fig. 4.6 Summarized workflow of the online module.

- The lookup is successful and the requested URL is forbidden (`DROP` in the session status flag). The HTTP response is dropped (i.e., a `NF_DROP` is returned to `netfilter`), and two new packets are generated: (*i*) a TCP RESET message sent to the web server to forcibly close the connection and (*ii*) an HTTP redirect message sent to the client in order to show the user a courtesy web page notifying that the requested web resource was blocked. Moreover the entry is removed by the HTTP session table.

- The lookup is successful but the system is still waiting for the policy server to respond (`UNKNOWN` in the session status flag). This occurs when the response from the web server arrives *before* the one from the policy server. In this case the HTTP response packet is put on hold by returning `NF_STOLEN` to `netfilter` and saved in the proper `skbuff` structure (shown in Figure 4.3) of the HTTP session table entry, waiting for the arrival of the answer from the policy server. This is the only case in which the user experiences an additional delay compared to a scenario where U-Filter is not deployed.

- The lookup is unsuccessful. Our algorithm interprets this condition as the URL being allowed, hence the HTTP response is forwarded to the client. Since in common URL filtering applications most URLs are not to be blocked, this design choice allows considerable space savings in the HTTP session table (Figure 4.3), as we avoid explicit entries for all the sessions that correspond to 'accepted' URLs.

Notably, the algorithm needs to hold (hence, store in the kernel session table) no more than *one* packet per HTTP session. In fact, even if other segments of the HTTP answer are in fact delivered to the destination, the TCP layer on the destination host cannot reconstruct the entire message because of the missing packet, which is the first segment of the HTTP response. This prevents the message to be actually delivered to the application (e.g., web browser) while keeping at minimum the memory storage requirements in the residential gateway. However, this solution also causes the transmission of some duplicated packets, which we analyze in Section 4.4.2 and that are discarded by U-Filter since they are equal to the packet already on hold.

4.3.3 Offline module

As depicted in Figure 4.2, the offline module is split in two portions, the first one operating as a process in user space, while the other operates as a thread in kernel space. The former is in charge of the communication with the policy server, as shown in Figure 4.7, while the latter executes the workflow summarized in Figure 4.8.

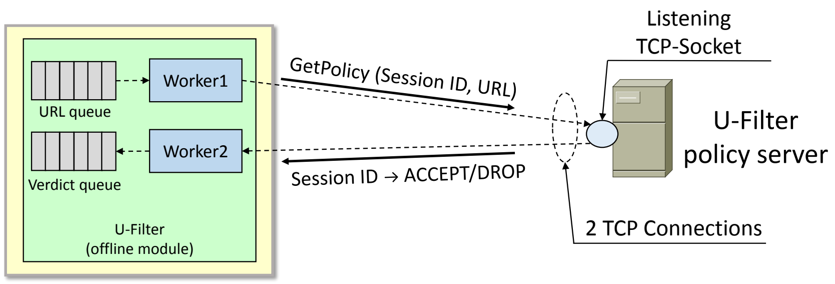


Fig. 4.7 Offline module user space process.

The user space process retrieves URLs from the URL queue and sends them to the policy server, which provides decisions stating whether they are acceptable or to be blocked. These decisions are then pushed in the shared verdict queue, together with the same TCP session identifier that was stored in the corresponding URL queue entry.

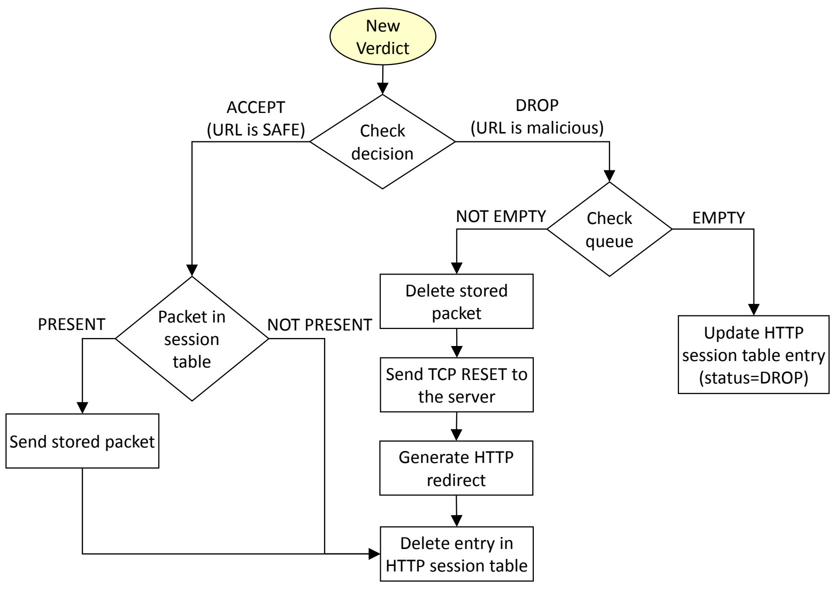


Fig. 4.8 Summarized workflow of the offline module kernel thread.

The entries in the verdict queue are retrieved by the offline module thread in kernel space, which reads the enclosed decision. In case the resource is legitimate (the entry contains the `ACCEPT` flag), it checks whether a packet is stored in the HTTP session table entry corresponding to the TCP session key present in the verdict queue entry. This packet, if present, is injected back into the networking stack of the operating system, exactly in the same point of the `netfilter` chain where it had been stolen, so that the packet is processed by any other software relying on `netfilter` (e.g., NAT). The HTTP session table is then updated by deleting the entry since, as mentioned earlier, the absence of an entry is interpreted as an `ACCEPT` verdict. The `skbuff` structure containing the first packet of the HTTP response is stored in a memory location managed by the operating system, hence the offline module leverages the kernel space thread to access it. In case the resource is not legitimate (the verdict queue entry contains the `DROP` flag), if no packet is found in the HTTP session table entry, the session status flag is updated to `DROP`, thus the online module will drop the response packet when it arrives. If a packet is already stored in the HTTP session queue entry, the offline module performs the same actions previously described for the online module in case of a `DROP` policy. Additionally the packet is dropped, so that the client cannot reassemble the HTTP response.

Additionally, the last *N* URLs verdicts are cached in the offline module. Each URL is first looked up in the ad-hoc verdict cache and, in case of a hit, there is no need to interact with the policy server and, in case of unauthorized result, redirection to the courtesy web page can be immediately implemented, thus reducing the overhead for the module.

4.3.4 Communication with the policy server

The U-Filter offline module exploits two different parallel threads to interact with the policy server, each one using a distinct TCP connection as shown in Figure 4.7. The two threads establish the TCP channels when the system starts, hence enabling the offline module to send immediately a query to the policy server when needed, without the overhead (and the consequent latency) of the TCP handshake[1].

The offline module exploits these threads to implement an asynchronous communication with the policy server, separately processing the requests and the replies without any wait. The first thread cyclically collects every new entry present in the URL queue and sends the URL and the TCP session identifier to the policy server, which replies with a message on the second thread, using the second connection, containing the same Session ID and a single binary information (`ACCEPT`/`DROP`) that is used to push a new entry in the verdict queue. This solution allows to process as fast as possible both new entries in the URL queue and new replies from the policy server. The Session ID sent back and forth is used to correlate the requests with the replies, so that there is no need to share data between the two threads. Since the requests are sent sequentially, the policy server can adopt different techniques to efficiently parallelize the policy checking, such as spawning new threads without the necessity to open a dedicated TCP connection for each of them. It is worth noting that most TCP implementations are designed to use the Nagle algorithm by default, in order to reduce the congestion of the network and increase bandwidth efficiency at the expense of latency [39]. This algorithm buffers application data until all the previously sent packets are acknowledged or the data reach the Maximum Segment Size (MSS). In this way the probability of having small packets in the network (i.e. packets smaller than the MSS) is strongly reduced, thus limiting the

[1]The messages sent to and received from the policy server are not intercepted by the callback of the online module, since they are addressed to the local host and do not cross the `NF_IP_FORWARD` hook, where the callback is registered.

overhead of TCP headers, allowing for a more efficient use of transmission links and reducing the burden on routers in terms of packets per second to be processed. This behavior is particularly harmful for U-Filter, since both the offline module and the policy server always send very small packets, that most of the time would be delayed up to one RTT. It is therefore crucial that the offline module and the policy server disable the Nagle algorithm (typically with the `TCP_NODELAY` socket option) when establishing the two connections.

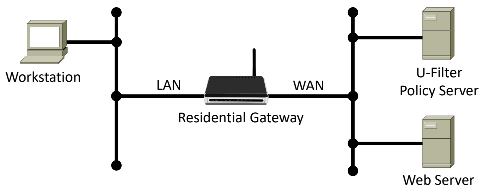
4.4 Experimental validation

In order to validate the proposed solution we conducted a broad range of experiments. Specifically our goal has been to study the interaction between the presented algorithm and TCP, as well as the conditions in which a web page load time is increased, quantifying to what extent the user experience is affected.

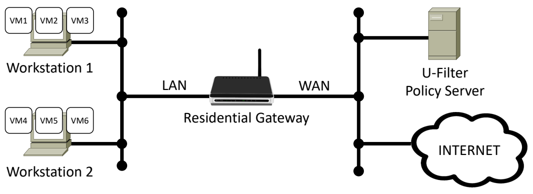
4.4.1 Testbed setup

We deployed U-Filter on a commercial low-cost residential gateway, a TP-Link Archer C7 (single core MIPS32 CPU clocked at 720MHz, 16MB Flash, 128MB RAM) running *OpenWrt* 12.09 [40] with the version 3.3 of the Linux kernel. OpenWrt is an open source operating system specifically optimized for the execution on resource constrained residential gateways. As shown in Figure 4.9, multiple workstations (whose number and setup varies according to the specific test) acting as clients are connected on a Gigabit Ethernet LAN representing the “domestic side” of the residential gateway. Another 1 Gbps interface (“WAN side”) hosts the policy server and the traffic sink of our experiments, which is represented by a web server during TCP interaction and throughput experiments or a vanilla Internet connectivity when evaluating browsing experience. All the workstations and the servers are equipped with an Intel Core i7-4770 CPU and 32GB of main memory in order to guarantee not to become the bottleneck.

Since a production-grade policy server is not in the scope of this work, we use a policy server that gives always a positive verdict, with a customizable delay in order to simulate the processing time. Moreover, in the policy server we use *Linux*



(a) Testbed to analyze the interaction with TCP and to evaluate the maximum throughput.



(b) Testbed to evaluate the browsing experience.

Fig. 4.9 Testbed setup.

Traffic Control (*tc*) to add a custom delay to any outgoing packet in order to simulate various network RTTs.

To generate single HTTP requests we use `curl` and `ab` [41], while for real-life simulations we start multiple VMs on the workstations to emulate multiple end-users. Each VM runs an instance of *WebTrafficGenerator*[2], an automation tool that can drive a web browser to replay a user browsing history. For every entry in the provided browsing history, the browser loads a complete web page (i.e. retrieving the web page with all the associated resources such as images, javascript files, etc.)[3]. In this respect, *WebTrafficGenerator* can also issue HTTPS requests, which happens when a page, appearing in HTTP in the browsing history, includes content that has to be retrieved using an encrypted connection. The time between multiple web page requests, a.k.a. the *Thinking Time*, is randomly selected using a random variable with the same statistical distribution as the actual thinking time of the user as measured from his/her browsing history. A realistic thinking time is required not only to simulate a real user behavior, but also to avoid that web services (e.g. Google) recognize that the client is an automaton and thus provide a different response web page with the intent of testing whether or not the user is human. In the event that a new request must start before the previous web page is completely loaded, the tool creates a different browser window, in order to load multiple web pages in parallel (which simulates multi-tabbing).

[2]https://github.com/netgroup-polito/WebTrafficGenerator

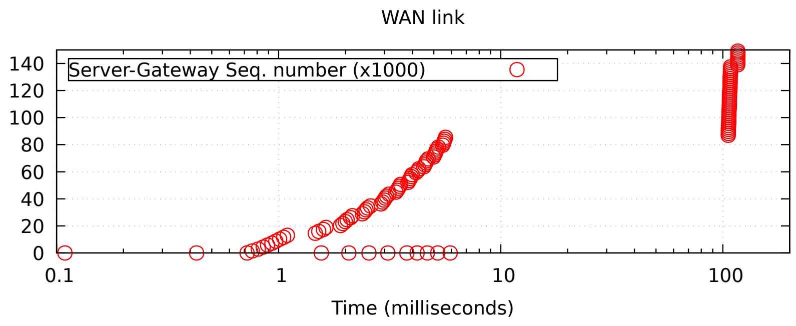
[3]The community have not yet reached a consensus on when a web page should be considered completely loaded. Particularly, *WebTrafficGenerator* considers a page complete when the javascript "`onload`" event is fired on the "`body`" HTML tag.

4.4.2 Interaction with TCP

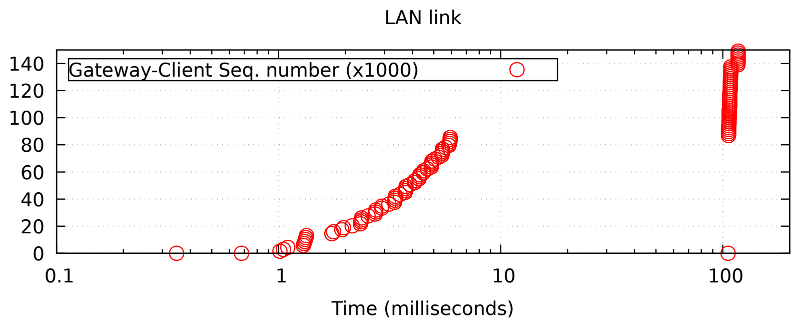
This section shows how the TCP algorithm reacts when one specific packet (the first packet of an HTTP response) is repeatedly lost on its way to the destination, for a certain amount of time. The aim of this analysis is to show that U-Filter has been designed keeping in mind the peculiar characteristics of the TCP protocol, hence our algorithm that possibly delays the first packet of the HTTP response does not cause additional delay in the TCP data exchange.

To reduce external interferences, in this test we use a web server directly connected to the WAN interface of the gateway (as shown in Figure 4.9a) running the *Apache HTTP Server 2.4.7*; T^W measured in this setup is less than 1 ms, thus we can consider $\Delta_{delay} = T^P$. Moreover, in this test the Linux Traffic Control (`tc`) in the policy server is disabled, hence the RTT is negligible and we can consider $T^P = T^P_{proc}$. A client workstation runs `curl` to request a 512 KB web page stored on the webserver. The gateway executes U-Filter with a fixed $T^P_{proc} \approx 100$ ms delay in the policy server response. As detailed in Section 4.3.2 and 4.3.3, only the first packet of any HTTP response is buffered by U-Filter. In the scenario created for these experiments, such packet is eventually forwarded to the client about 100 ms after the HTTP `GET` request traverses the residential gateway. All subsequent packets are forwarded correctly. We capture the traffic on both the LAN and WAN links of the residential gateway and extract the sequence numbers (SEQ) of the TCP segments from the web server to the client and the acknowledgment numbers (ACK) of the ones from the client to the webserver, together with their timestamp. The resulting data are presented in Figure 4.10 (the SEQ and ACK numbers are relative).

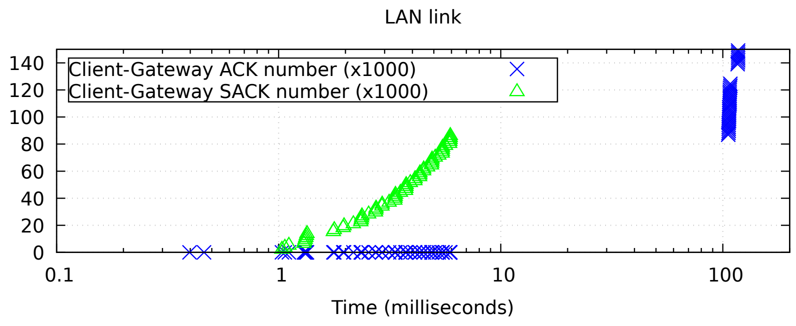
This experiment enables us to observe how a TCP connection progresses during the U-Filter operation. The presented results show that, while the first TCP segment of the HTTP response is blocked, the server TCP endpoint sends the subsequent segments as well as duplicates of the first segment (visible only on the WAN side, in Figure 4.10a), until the TCP window is full. As expected, the TCP receiver repeatedly acknowledges the segment arrived before the one missing (Figure 4.10c); specifically one ACK is sent for each of the subsequent segments received out of sequence. All the modern TCP implementations include the *TCP selective acknowledgment (SACK) option* [42] in the duplicated ACK, which is used to selectively acknowledge correctly received segments logically following the missing one(s). Thanks to the selective acknowledgments, these segments are not re-transmitted, as it happens



(a) Response packets timing on the WAN link



(b) Response packets timing on the LAN link



(c) Acknowledgement packets timing on the LAN link

Fig. 4.10 Progress of a TCP session.

for the blocked segment, as the traditional Go-Back-N algorithm would require. When the blocked packet is released (after 100 ms in our experiment, as shown in Figure 4.10b) and properly delivered, all the previously received segments are cumulatively acknowledged and the transmission can continue from a new segment (Figure 4.10c).

Abiding by *TCP Fast retransmit* [43] algorithm, the web server re-sends the blocked segment for every 3 duplicated acknowledgments. These re-transmitted segments are the only overhead induced by U-Filter. In our test these duplicates amount to 12.8% of the packets sent by the server during Δ_{delay}, and half that number if we consider *all* the packets transmitted during the same interval; however, considering the entire lifespan of the TCP connection, this overhead accounts (on average) no more than 1.6% of all the packets, which can be considered negligible.

From the point of view of the users' experience, selective acknowledgments are particularly beneficial because, even if the policy server replies after the web server (i.e. Δ_{delay} is positive), the actual delay perceived by the user is smaller than Δ_{delay} because several TCP segments are correctly received during the Δ_{delay} interval and are ready to be used to render the web page as soon as the missing segment is delivered.

4.4.3 Browsing experience

This section presents the results of several tests executed in a realistic scenario to show how much a real user browsing experience is affected by U-Filter. Using the testbed in Figure 4.9b, we launched *WebTrafficGenerator* in 6 VMs (running on 2 workstations) in order to simulate 6 users simultaneously browsing the Internet. This number of concurrent users is reasonable for a residential gateway. Moreover, with a large number of users, the browsing experience would be limited by the network speed. As expected, the latency of the policy server proved to be the parameter that has the greater impact on the user-perceived performance of U-Filter.

In every test, a single VM browses 600 web pages collected from the browsing histories of 30 anonymous users (we consider only web pages downloaded using HTTP, since those using HTTPS are irrelevant for U-Filter). In order to use realistic values for the policy server processing time and RTT, we analyzed several traffic traces captured using Tstat [44] during 24 hours in 4 different points of presence

Table 4.1 Inferred RTT values with the policy server in different locations (RTT^P).

Location	Type of measure	RTT
POP	Median	25 ms
	90th percentile	100 ms
Data Center (DC)	Median	45 ms
	90th percentile	200 ms

Table 4.2 Inferred policy server latency values (T^P_{proc}).

Type of measure	Latency
Median	2 ms
90th percentile	80 ms

(POPs) of an Internet Service Provider and extracted the median and 90th percentile values for the RTT of HTTP requests and processing time of web servers. Tstat infers the RTT from the POP to an endpoint by measuring the inter-arrival time of a packet and its acknowledgment and infers a web server processing time by measuring the interval between the arrival of the acknowledgment for the request and the arrival of the first response packet. In fact, a host's operating system usually sends a TCP ACK as soon as a packet is received.

Table 4.1 shows the statistical values for the RTTs from a client to the POP and from a client to the destination server. We use these values in our tests to simulate the RTT in the case that the policy server is either in the POP or in a remote data center, those measures were taken from the university network. Additionally Table 4.2 shows the statistical values of the processing time for web servers. These values are used to simulate the processing time of the policy server: since the operations performed are somewhat similar (parsing of a request, look up in a database, preparation of a response), we assume the complexity to be comparable with (or even lower than) the one of any web server.

At the end of a test, *WebTrafficGenerator* provides a file containing a summary of various aspects of every request. Among the provided values, we are interested in the **complete page** load time (the time needed to load the web page with all its

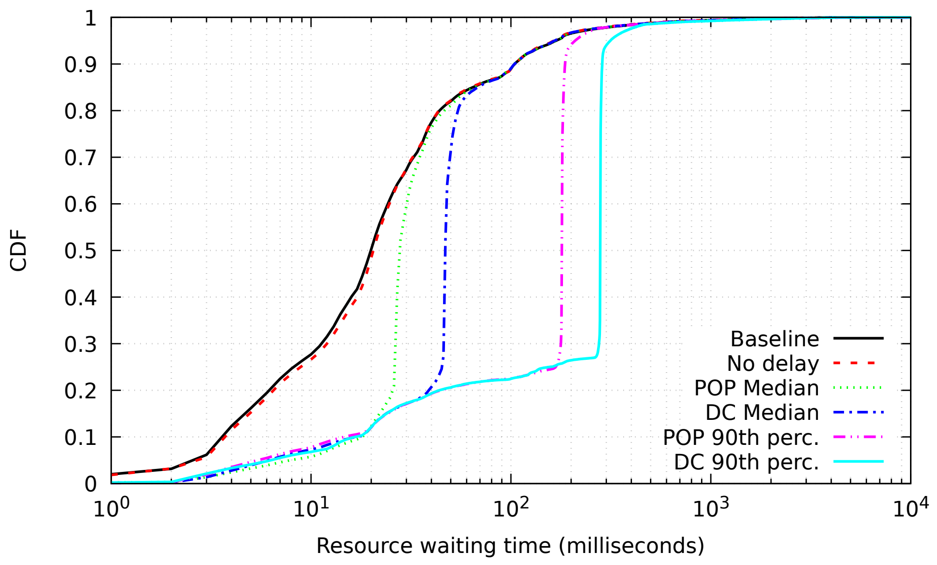


Fig. 4.11 Waiting time for a single HTTP resource - Cumulative distribution function.

resources, such as pictures, libraries, etc.) and the timings of the **individual HTTP requests** issued to get the main HTML page and the associated resources.

Individual HTTP requests

The timing of an HTTP request is the sum of multiple components, such as the queuing time, the DNS resolution time, the connection setup time, etc. The only component that can be affected by U-Filter is the time spent waiting for a response from the server (*waiting time*), equal to $\max\{T^P, T^W\}$, if the RTT between the client and the gateway is negligible. Figure 4.11 shows the cumulative distribution of the waiting time for HTTP requests with different values of RTT and processing time (latency) for the policy server, together with the baseline (i.e., the latency without U-Filter) and the case in which the policy server immediately provides verdicts (in which case the delay T^P is negligible), as if U-Filter and the policy server are on the same LAN. The different measurement from the figure was done using the delays presented on the table shown on the previous section.

These results show that U-Filter adds a negligible delay if the policy server provides an immediate response, therefore proving our claim that the online module does not introduce noticeable overhead in the traffic processing. On the other hand, when the policy server response is received after a certain amount of time, the

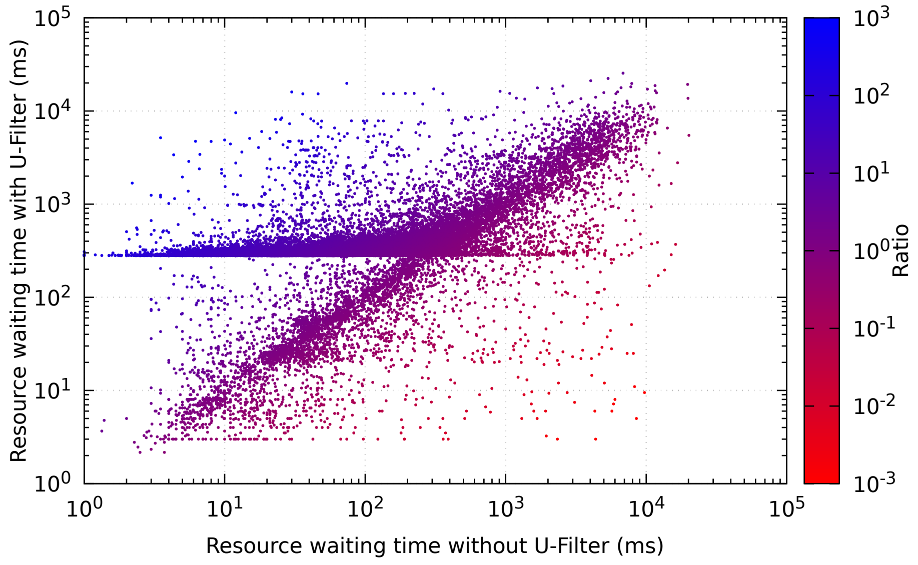


Fig. 4.12 Resource waiting time considering the 90th percentile of the processing time and RTT with the policy server in a data center.

cumulative distribution is shifted toward that value, since all the HTTP responses that arrived earlier are delayed by U-Filter. In summary, the impact of U-Filter on the single resource loading time is highly dependent on the distance from the policy server and its processing time.

Considering only the worst case (i.e., the 90th percentile of the processing time and RTT with the policy server in a data center), we show in Figure 4.12 the waiting time for each requested HTTP resource, with and without U-Filter. The figure shows a cluster of requests on the horizontal line corresponding to the delay T^P, supporting the conclusion that this delay highly influences the loading time of a single resources.

Both figures show that, even with U-Filter, some resources are received before the policy server delay ($T^P \approx RTT^P + T^P_{proc}$). This happens because some resources are retrieved through HTTPS, even if the main HTML page is on HTTP, therefore they do not experience the policy server delay.

Complete pages

Figure 4.13 shows the cumulative distribution function of the complete web page load time, while Figure 4.14 shows for every requested URL the relation between the complete page loading time with and without U-Filter, in the worst conditions

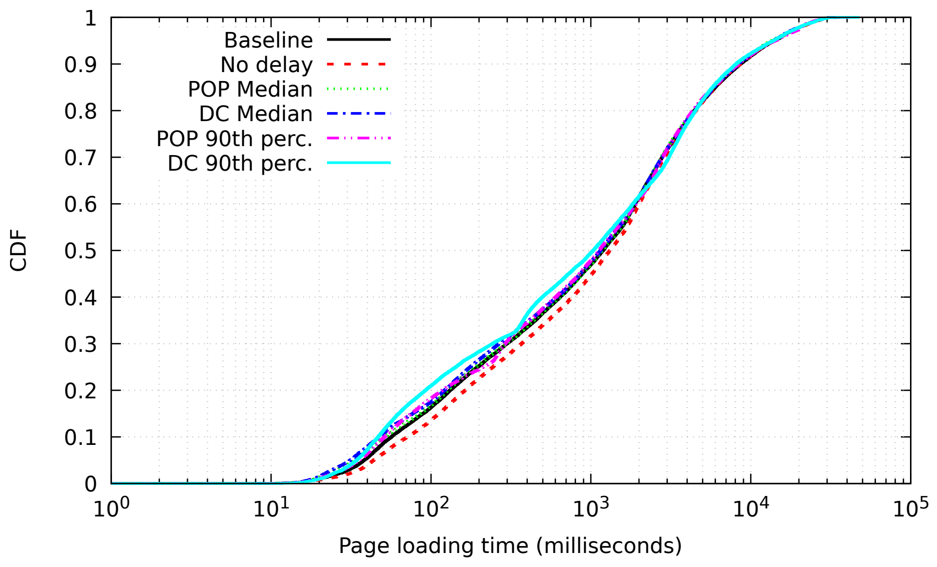


Fig. 4.13 Complete page loading time cumulative distribution.

(policy server in the data center, 90th percentile values for RTT and latency). These results show that the impact caused by the presence of U-Filter is not noticeable, therefore we can assert that the overall page loading time is not affected by U-Filter and also the browsing experience is unaltered.

This is justified by the fact that multiple resources are requested **in parallel** by the browsers, hence the policy server processes all the requests concurrently. As a result, the increase in the overall time for loading the complete web page is not dependent on the number of resources and is, in any case, approximately equal to a single policy server delay T^P. Since the time needed to receive, parse and render the main HTML web page and all its resources is usually an order of magnitude greater than the policy server delay, the added latency (and the impact of U-Filter on the browsing experience) is in effect negligible.

4.4.4 Residential gateway aggregated throughput

In this section we evaluate the overhead introduced by U-Filter by comparing the average aggregated throughput of the residential gateway in 3 scenarios: (*i*) without a URL filtering service in place, (*ii*) with U-Filter and (*iii*) with *Tinyproxy* [45], a URL filtering solution for OpenWrt based on a lightweight HTTP proxy that intercepts and analyzes all the outgoing web traffic and can operate in either explicit or transparent

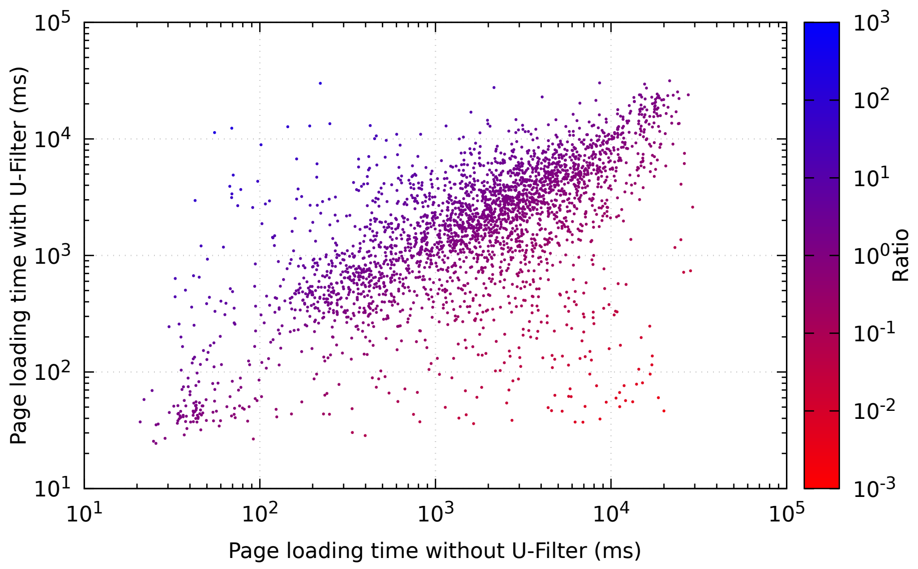


Fig. 4.14 Complete page loading time considering the 90th percentile policy server processing time with the policy server in a data center.

(a.k.a. man-in-the-middle) mode. These experiments assess the impact of U-Filter with respect to the maximum forwarding capabilities of the residential gateway, which is basically limited by the CPU consumption of the on-board software.

These experiments employ the testbed setup depicted in Figure 4.9a; the policy server is configured to simulate a deployment in a data center with the median processing time and RTT, while the web server has the same RTT. The client workstation uses `ab` to request files of different sizes from the web server; each file is requested 100 times. As suggested by the HTTP/1.1 standard [46] with respect to persistent HTTP connections, each client issues two concurrent requests toward the server. The goal of this experiment is to evaluate how much packet inspection and policy checking in the residential gateway affects the download speed and the latency. We show in Figure 4.15 the minimum, maximum and average application-level throughput for the 3 scenarios, while in Figure 4.16 we show the time needed to download the entire file.

These results show that the throughput and the download speed reached with U-Filter are higher than with Tinyproxy for files larger than 8 KB, while for small files the two solutions show the same level of performance. In fact, with very small files, we experience an additional small delay with U-Filter, compared to the baseline. We ascribe this delay to the time needed for the context switch between the online

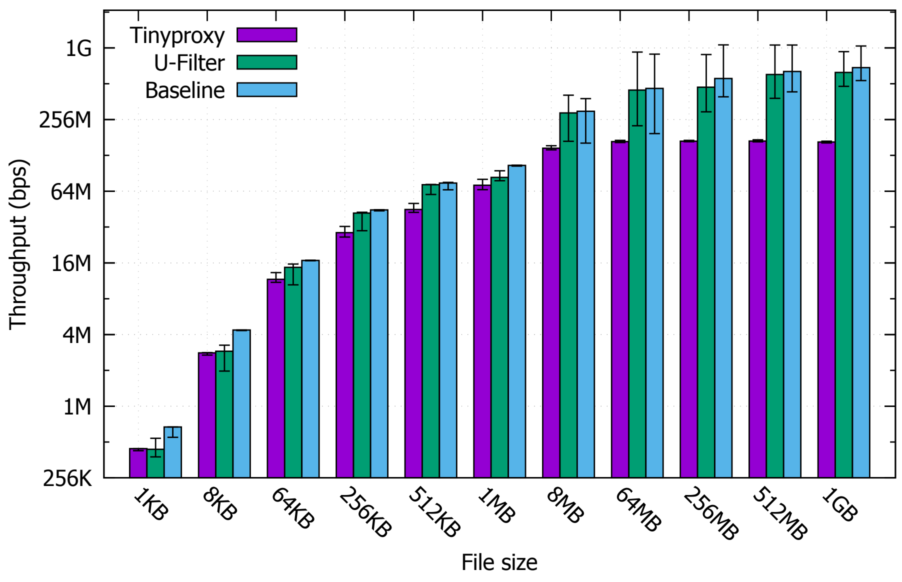


Fig. 4.15 Application-level throughput when downloading files of different sizes.

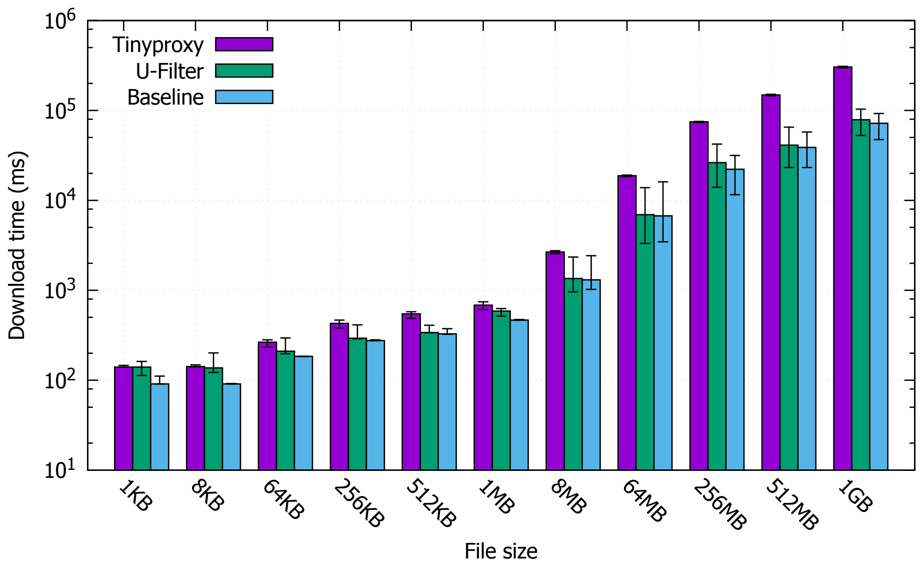


Fig. 4.16 Download time when requesting files of different sizes.

and offline module, given that the residential gateway has a single core. This delay is negligible for larger files, for which U-Filter provides almost the same performance reached without the filtering service in place. We expect that a residential gateway with at least a dual core processor would not experience this delay, therefore U-Filter would provide the same level of performance as the baseline. However, even with a single core gateway, the impact of U-Filter on the download time is only 3% with large files and never exceeds 54%, while Tinyproxy has an overhead ranging from 44% to a remarkable 322%. As an example, the download of a 1 GB file requires approximately 1 minute and 12 seconds without a filtering service, 6 seconds longer with U-Filter and more than 5 minutes with Tinyproxy.

It is worth mentioning that U-Filter can easily implement a whitelist containing the IP addresses of trusted devices whose traffic should not be filtered. This is a useful feature that allows to avoid the additional delay for delay-sensitive clients.

4.4.5 Memory footprint

Given the limitations in terms of available memory in current residential gateways, we extracted the number of pending entries in the HTTP session table every time a new HTTP request was received and plotted the resulting probability distribution in Figure 4.17 in order to assess the impact of U-Filter in terms of memory consumption. The observed values confirm the small memory footprint of U-Filter: even in the worst case, the number of pending entries are always less than a hundred. In the case in which every entry stores a packet (usually 1518 bytes at most), together with IP addresses (8 bytes), TCP ports (4 bytes) and a binary session flag, the HTTP session table requires less than 200 KB of main memory, a value far below the memory size of low-end residential gateways (usually in the order of at least tens of MB).

4.5 Related work

Currently several solutions for filtering traffic based on URLs are available commercially or as open source packages, often used as parental control or ad block. Many are based on software executing on the client machine to control outgoing traffic. Among them, it is worth mentioning *k9 Web Protection* [47], a powerful free software for URL filtering that comes with a large database of URL categorization

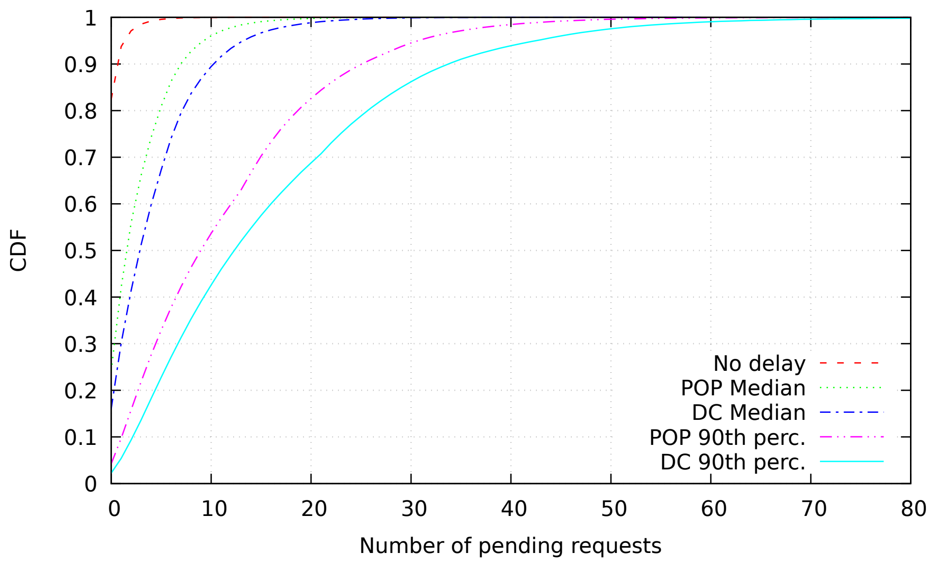


Fig. 4.17 U-Filter load.

data. New websites are categorized in real-time and their information published on a server that is used to update the local database. This software needs to be installed on any device that must be protected and is tuned to run on common PC hardware.

An idea of outsourcing the complex operations from middleboxes to the cloud is presented in [48] that is similar to the NFV concept, reducing the middleboxes operations reduces their costs.

Among existing parental control solutions that do not require execution of a software agent on clients, some are based on applying the filtering policing in the DNS server [49]. While this is a low complexity and efficient solution that enables achieving high performance, it is not effective as it can be easily bypassed choosing a different DNS server. Moreover, filtering is based on server domain names rather than URLs, as required when the same server or name domain can deliver both appropriate and inappropriate content, such as in case of public services like `facebook.com`.

As an alternative approach, filtering policies can be applied by network appliances on the path of the protected client traffic. *Blue Coat WebFilter* [50] is a sophisticated URL filtering solution that runs on business level network appliances and provides policy enforcement on web traffic, blocking malware downloads and web threats. *WebFilter* combines URL filtering and anti-malware technologies, exploiting an engine with a local rule database continuously updated from a remote master database. The engine detects hidden malware and provides reputation and web content categorization based on input from actual users.

None of the above-mentioned solutions is designed to run on resource constrained devices, such as a typical residential gateway, which would not ensure acceptable performance when executing computationally intense tasks. Among the efforts to integrate web filtering service in low-end residential gateways, the ones related to the OpenWrt platform are noteworthy, such as Tinyproxy [45]. Tinyproxy can filter HTTP requests checking their URL against a list of regular expressions contained in a local file, which may be rather big and needs to be frequently updated. A similar technology has been proposed in [51], where an access gateway performs mobile app policy enforcement deploying a transparent HTTPS proxy to gain access to *encrypted* traffic, extract relevant field values, and pass them to an external policy-checking module. However, deployment of an HTTP proxy is critical on resource-constrained devices since it must terminate all the TCP connections, pair them with new TCP connections with the remote endpoint, parse every packet, identify and extract patterns of interest, and match them against a large blacklist. Therefore it becomes easily a bottleneck with high traffic loads, thus impacting user experience.

The work presented in [52] represents an attempt to perform efficient HTTP traffic filtering in *OpenWrt*. The authors propose a two-tier architecture, with a kernel module that intercepts and analyzes HTTP traffic and a user-space process in charge of policy compliance checking. The computational load of the user space module, that performs string matching on URLs, grows with the length of the list of rules, and so does the introduced delay. Consequently, when this approach is implemented on a residential gateway with limited resources, only short lists can be supported without user experience degradation, thus limiting the effectiveness of the policy enforcement system. Moreover, the proposed architecture makes it difficult for a trusted third-party to push real-time updates to the local database in order to ensure prompt detection of newly discovered threats. Finally, the URL analysis is performed by each edge systems in isolation, hence excluding the possibility of a (centralized) cross-correlation mechanism that identifies new threats by analyzing URLs requested from different sources.

Traffic processing in residential gateways has been proposed also in the context of NFV [53, 54]. An existing NFV infrastructure can employ residential gateways to deploy lightweight eBPF data plane programs [55], in order to provide delay-sensitive services to the user, while computation intensive services are hosted in the data center of the service operator. This solution offers flexibility in the type and number of network services that can be provided and represents an interesting target platform for the deployment of U-Filter.

4.6 Conclusion

Leveraging an external policy server and an intelligent combination of kernel and user space processing (and a careful implementation), U-Filter is able to inspect the URL in every HTTP request and block unwanted web pages with a very small memory footprint and processing overhead. This makes U-Filter appropriate for the deployment on resource-constrained devices and also reduces at a minimum the additional delay introduced on page download, which leaves the overall browsing experience of the user practically unaltered.

Since U-Filter operates on a packet-by-packet basis, it assumes that the entire HTTP header is on the same packet. This makes URL extraction easier and avoids to have to store additional information to correlate subsequent packets. Since the maximum size of an IP packet is usually 1500 bytes, this does not represent a problem in a real scenario, as confirmed by [56].

The policy server, where multiple mechanisms and optimizations can be implemented, was purposely kept outside of the scope of this work as it involves a completely different set of challenges and solutions. Similarly, we did not address how providing additional information to the residential gateway can increase its efficiency in caching verdicts, thus reducing the number of interrogations.

In case of HTTPS traffic the URLs matching can not be done because the HTTP traffic results encrypted, using the same idea the module can extract the data from the unencrypted packets during the handshake of the connection and with the `server_name` field on the `CLIENT HELLO` packet the module can know the name of the server that the client wants to contact.

This chapter shows that a lightweight network function can be successfully installed and executed even on low-profile hardware such as domestic CPE. In the context of NFV services, we can point out two main observations. Fists, the main lesson learned from the current work is that even resource-constrained devices can be used to provide powerful services, particularly when the proper collaboration between edge and cloud services can be exploited. However, this requires a careful design of the network function, which should be able to consume a limited amount of variable resources; in this context, starting a VM as VNF is out of question. Second, the limitation of the presented work is the lack of flexibility: U-Filter is a function that is deployed *ahead-of-time* and any modification to the service (or, even, the setup of an additional service) requires a re-flashing of the whole system boot disk. In the context of NFV, a better way to setup services should be envisioned in order to support the *service-on-demand* model that is available in NFV.

This work has deeply inspired the results presented in the next chapter, in which a special-purpose node has been designed to support NFV services, in particularly through a new service deployment model for lightweight virtualized services called *native network functions*.

Chapter 5

Designing a platform for NFV

Part of the work described in this chapter has been previously published in [57] an [58].

5.1 Overview

NFV enables the instantiation of network functions (NFs) across the (possibly heterogeneous) compute resources available in the infrastructure of a network operator, ranging from Customer Premises Equipment (CPE), which are typically based on low-cost hardware, to high-end servers in the operator data centers.

The required service is a list of NFs that need to be deployed. Following the results of Chapter 2, NFs can be implemented with different technologies; so we need to design a software architecture that can support the possibility to have different execution environment, and in particularly that can guarantee the NFV flexibility also when resource-constrained devices are involved.

In order to enable efficient service deployment and delivery, we designed COMPOSER (COMPact Open-source SERvice platform), which offers a high-level abstraction for composing network functions in arbitrary service graphs used to deliver virtualized services. We design COMPOSER so that it is well-suited to run virtualized services on high-volume servers, as one would expect based on the current research and industry efforts. In addition, we demonstrate that COMPOSER brings the power and advantages of NFV on resource-constrained hardware such as home-

/SOHO CPEs, also known as residential gateways, thus enabling carriers to use the same service deployment platform across the entire (heterogeneous) infrastructure available from the end-user sites through the fronthaul/backhaul and transport network functions all the way to their core network.

COMPOSER exploits locally available information to optimize service deployment. For instance, COMPOSER evaluates local resources/constraints to select the best implementation of a required function and binds NF(s) to the most appropriate CPU core. Finally, COMPOSER is able to execute NFs running in different execution environments, according to the different operating contexts. For instance, a resource-constrained CPE can execute NFs on bare metal, while full-fledged virtual machines are more appropriate for “fat” servers. Hence, COMPOSER adds the possibility to use the native applications on home gateways, thus extending the results presented in Chapter 4.

The remainder of the chapter is structured as follows. Section 5.2 highlights the design objectives of COMPOSER, mainly focusing on the unique features of the platform with respect to other proposals in the NFV literature. Section 5.3 presents the overall software architecture of COMPOSER. An extensive evaluation of COMPOSER is provided in Section 5.4, while Section 5.5 analyzes xisting related work. Finally, Section 5.6 concludes the chapter.

5.2 Design objectives

This section summarizes the COMPOSER design objectives, focusing primarily on the unique characteristics of the platform.

5.2.1 Domain-oriented orchestration

COMPOSER can interact with overarching orchestrators operating according to two different models: *domain-oriented orchestration* based on functional capabilities, and what we will refer to as “legacy” orchestration, akin to what is currently used in the cloud computing/data center world based on checking for infrastructure capabilities and available resources. More specifically, the former model abstracts each infrastructure node, that can be considered as a “domain”, with a set of functional capabilities, thus hiding from the overarching orchestrator the internal details of each domain, such as the amount of available resources or the way in which a NF is actually implemented. This enables the upper orchestration layers to request a NF such as `firewall` instead of prescribing a "specific VM" (that, for instance, implements a firewall).

With this model, COMPOSER can *(i)* support different implementations of the same NF and use them transparently from the upper orchestration layer; and *(ii)* flexibly select the best implementation for a given request and specific NF, according to the current state of COMPOSER per se, i.e., number of CPU cores currently available or availability of a NF compatible with a hardware accelerator. Consequently, COMPOSER exposes functional capabilities, while infrastructure capabilities, such as the possibility to execute KVM-based VMs, and resources, such as the amount of CPU/RAM available on the node, are exported only when necessary to maintain compatibility with legacy orchestrators.

5.2.2 Network abstraction

The COMPOSER control plane can interact with different virtual switches (vSwitches) in order to implement paths between functions, each of which may be more appropriate for a specific deployment (e.g., CPE vs. high-volume server). For instance, a vSwitch optimized to exploit hardware acceleration available on the specific node is well suited when such hardware component(s) exist, while another vSwich, tailored to exploit multiple CPU cores available in high-end processors, may be the best choice when COMPOSER is deployed on a high-volume standard server.

5.2.3 Compute abstraction

The COMPOSER control plane is able to interact with different execution engines and therefore to execute NFs in different ways, depending for example on the availability of some hardware accelerator or specific software libraries on the infrastructure node, and the amount of available resources (e.g., CPU, memory). In fact, different execution engines may have different requirements in terms of hardware resources needed to run their NFs (e.g., VMs use more memory than containers), and hence are well suited for particular COMPOSER deployments.

5.2.4 Joint network and compute service graph optimization

The service to be implemented by COMPOSER is specified according to the Network Functions-Forwarding Graph (NF-FG) formalism detailed in Section 5.3.1, which describes both the compute and networking aspects of the service, i.e., the required functions and the interconnections between them. This way, each COMPOSER node has the complete view of the entire service, and can thus optimize, for example, the binding between NFs and CPU cores by considering the way NFs are interconnected with each other in the service graph.

5.2.5 Small footprint

This enables service deployment also on resource-limited hardware already available at the edge of the network, e.g., on residential gateways, that cannot be controlled by existing software platforms such as OpenStack.

5.2.6 Support for Native Network Functions

Existing virtualization engines are quite demanding in terms of resources required to run NFs (e.g., memory, CPU, image size), therefore they may not be appropriate for resource-constrained nodes. However, such devices usually run a Linux-based operating system that includes a number of software modules (e.g., `iptables`) that can be used to implement NFs executed directly on the host, hence providing services with a reduced overhead compared to VMs and containers. Furthermore, CPEs may include some hardware components (e.g., crypto accelerator, L2 switch) that can be exploited to implement NFs as well.

Native Network Functions (NNF) represent a way to exploit these native (software and hardware) modules, delivering efficient NFs implementations *and* reduced overhead. As a consequence, COMPOSER, due to its small footprint and seamless support for NNFs, enables the operator to deploy the service graph on existing residential gateways, which are then integrated in the overall NFV telco infrastructure. This is a design feature that in a real-world deployment enables an overarching orchestrator to optimize NF placement and scheduling. As an illustrative example, NFs required to be close(r) to the end users (e.g., secure tunnel termination, low-latency functions, etc.) can be instantiated directly on the CPE, while other functions of the same service (e.g., network address translation) can be executed in a data center.

Unlike VMs and Docker containers that are well-known virtual execution environments, the COMPOSER concept of Native Network Functions (NNFs) [57] enables the deployment of service graphs on current, resource-constrained CPEs, which are then fully integrated in the programmable NFV carrier infrastructure, as opposed to requiring their own silo for control and management. In fact, as shown later in Section 5.4.4, NNFs have a significantly reduced image size with respect to other virtualization environments, which makes them well suited to be executed on resource-constrained devices.

We define NNF as a data plane processing component that exploits capabilities natively present on the platform, which are instantiated as software or hardware modules. Each NNF is executed directly on the host. In practice, an NNF can be implemented as a `tgz` archive containing a set of bash scripts that are called by the *NNF driver* (see Figure 5.1) to implement the functions defined in Table 5.4, required in the NF lifecycle management.

Obviously, to be able to execute a particular NNF, all required modules, or *dependencies*, must be available on the node. Then, besides all information required for the execution of a generic NF (e.g., number of ports), the template associated with a NNF also includes a list of dependencies, which might refer to software packages (e.g., executable, libraries) that must already be installed and that are required by the NNF to operate. The NF resolver also considers these dependencies when selecting the best implementation for the required function.

Differently from virtualization technologies such as VMs and Docker, which support an isolation model for the instantiated NFs, NNFs rely on scripts executed in the host operating system. As such, the NNF driver needs to explicitly implement a layer that provides some form of isolation of the NNF against the rest of the system. Particularly, the NNF driver creates a *network namespace* before starting the NNF, adds to that namespace the virtual ports (virtual Ethernet (`veth`) interfaces) required to connect the NNF to the logical switching instance (LSI), and then starts the NNF within the namespace.

Launching a NNF, i.e. a script running on bare hardware, offers less protection than starting software in a VM or in a container, which can leverage the additional protection shield provided by the hypervisor or the container execution engine. For instance, little protection exists to limit the resources used by NNFs, e.g., in terms of CPU/memory consumption or the number of occupied CPU cores. Although the impact of the above problems could be limited by turning on some additional Linux mechanism such as `cgroup`, this may complicate the solution to a point in which other alternatives may be more appealing, such as replacing the NNF with a Docker-based implementation.

However, it is worth noting that, in any case, no protection exists that prevents a NF, which is expected to provide a given service (e.g., firewall), to behave differently, e.g., to launch an attack toward a remote host and the current solution is simply to trust the creator of the application or the entity (e.g., app marketplace owner) that markets it. Therefore, although we acknowledge that the problem of determining whether a NF is malicious is emphasized in case of NNF because of their inferior degree of isolation, we feel that the problem is rather general and should require a more generic solution that guarantees, a priori, the “goodness" of the NF.

5.3 COMPOSER architecture

Figure 5.1 illustrates the COMPOSER architecture, which includes the following main building blocks. The COMPOSER orchestrator (*c-orch*) is the main component of the control plane; it receives commands through a northbound interface and takes care of implementing them on the infrastructure node. *c-orch* provides network and compute abstraction, hence it orchestrates compute and network resources within COMPOSER by handling the complete lifecycle of the virtual execution environment(s) and networking primitives (e.g., traffic steering rules). Moreover, *c-orch* facilitates domain-oriented orchestration and enables joint optimization of compute and network resources based on the incoming service graph requests.

c-orch relies on the *NF repository* to select the best NF implementation available that matches the service request. The NF repository may also be deployed on another server and can be contacted by multiple nodes.

The data plane includes a *vSwitch* that controls the traffic paths between the NFs and a number of *compute engines* that can execute NFs implemented with different technologies (e.g., virtual machines, containers, or natively).

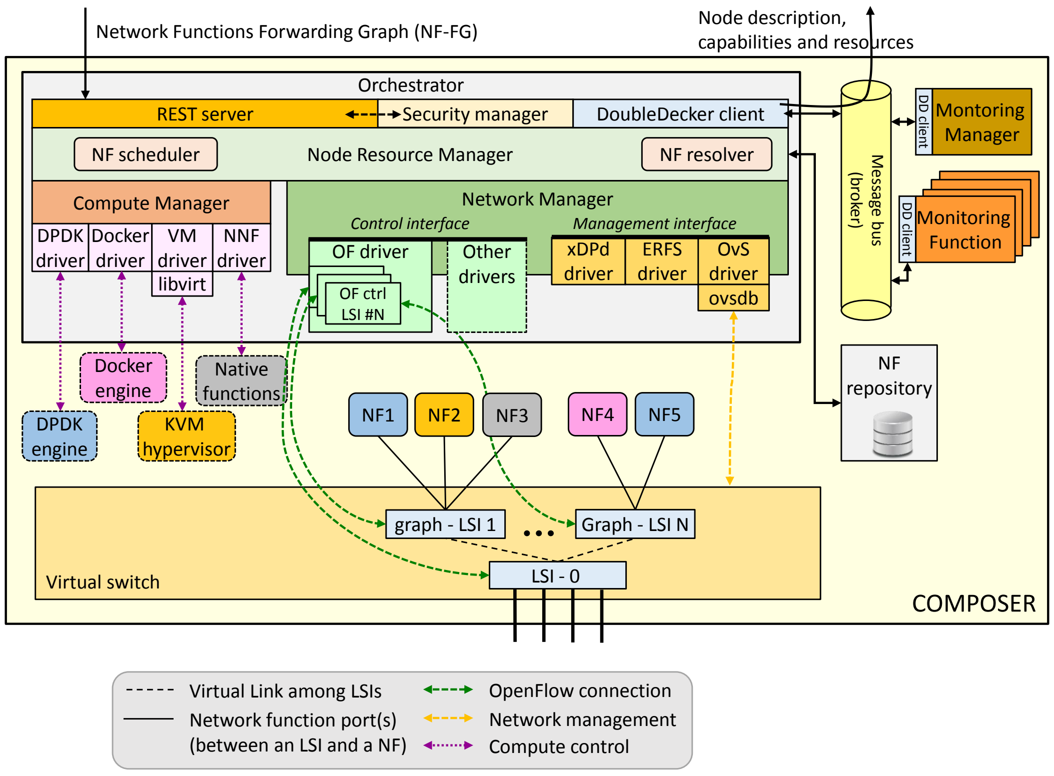


Fig. 5.1 Detailed architecture of COMPOSER.

The remainder of this section details the modules of c-orch (Figure 5.1), together with its northbound interface that is used to interact with the upper orchestrators, and the traffic steering model used to properly implement network paths between NFs.

5.3.1 Northbound interface

c-orch interacts with the overarching orchestrator(s) through a bidirectional northbound interface. Specifically, c-orch receives a service graph, and exports information using an OpenConfig-derived [59] YANG model describing the COMPOSER domain informations.

As shown in Figure 5.1, Create, Read, Update and Destroy (CRUD) commands related to the NF-FG are received through a REST API, while the COMPOSER description is exported through a hierarchical messaging bus based on a publish/-subscribe model. The overarching orchestrator knows exactly the COMPOSER entity (e.g., the IP address of c-orch) on which (part of) the service has to be created/updated/deleted, and hence a REST interface is appropriate for this function. On the other hand, using a publish/subscriber mechanism the c-orch does not need to know the consumer(s) of the description it exports, as there may be several entities interested in such information. All COMPOSERs publish their description using a specific *topic*, enabling all entities interested in such information to subscribe to the corresponding topic. In this case, a pertinent example is the above-mentioned orchestrator, which can use the COMPOSER descriptions to select the node where the service should be deployed.

As depicted in Figure 5.1, the REST server interacts with the *security manager*, a module that manages authentication and checks the permissions of the entities that can send commands to c-orch. For instance, only the network operator may be allowed to deploy NF-FGs, while end users may be only authorized to access the current service graph operating on their own Internet connection in read-only mode.

Network Functions - Forwarding Graph

The *Network Functions - Forwarding Graph (NF-FG)* formalism [60] describes the service to be instantiated with respect to compute (i.e., functions composing the service) and network (i.e., traffic steering rules) primitives.

```
{
  "nffg":
  {
    "id": "0x1",
    "name": "example graph",
    "NFs": [
    {
      "id": "0x1",
      "name": "firewall",
      "ports": [
      {
        "id": "0xa",
        "name":
        "internal port"
      },
      ....
      ]
    },
    ....
    ],
    "saps": [
    {
      "id": "0x1",
      "type": "interface",
      "interface": {
        "if-name": "eth1"
      }
    }
    ....
    ],
    "flowrules": [
    {
      "id": "0x1",
      "priority": 1,
      "match":
      {
        "port_in":
        "access-point:0x1",
        ....
      },
      "actions":
      {
        "output_to_port":
          "nf:0x1:0xa",
        ....
      }
    },
    ....
    ]
  }
}
```

Fig. 5.2 Excerpt of Network Functions - Forwarding Graph (NF-FG).

As shown in Figure 5.2, each NF-FG consists of three main parts. First, the `NFs` section lists the functions that compose the service. Particularly, the NF-FG may require a function without specifying any specific implementation (e.g., `firewall` in the figure). In this case the proper image is selected by c-orch through the interaction with the NF repository. However, the NF-FG can also ask for a specific implementation of a function, by specifying a template that describes it in terms of, e.g., image to be executed, number of CPU cores needed, technology to be used to implement the virtual network interface cards (vNICs), and so on.

The `saps` section describes the *Service Access Points (SAPs)*, namely the ingress/egress points of traffic in the (part of the) service deployed on COMPOSER.

```
{
  "id": "0x2",
  "type": "vlan",
  "vlan":
  {
    "vlan-id": "25",
    "if-name": "eth1"
  }
}
```

(a) *vlan* SAP

```
{
  "id": "0x3",
  "type": "gre",
  "gre":
  {
    "local-ip": "10.0.0.1",
    "remote-ip": "10.0.0.2",
    "gre-key" : "0x1"
  }
}
```

(b) *GRE* SAP

Fig. 5.3 Examples of SAPs in the NF-FG.

In this case, while the *interface* SAPs (Figure 5.2) correspond to physical or virtual interfaces of COMPOSER, a *vlan* SAP only includes traffic belonging to a specific VLAN, although possibly associated with an interface. This means that COMPOSER guarantees that only the traffic with a specific VLAN ID arrives from this SAP (e.g., VLAN ID 25 in Figure 5.3a), and that all traffic sent on such a SAP is tagged (by COMPOSER) with the proper VLAN ID.

The *GRE* SAP (an example is shown in Figure 5.3b) represents the termination of a GRE tunnel. COMPOSER guarantees that only the traffic encapsulated in a specific GRE tunnel enters from this SAP, and that all traffic exiting from such a SAP will be encapsulated in such a GRE tunnel.

Both *vlan GRE* SAPs can be used to connect (i.e., steer the traffic between) parts of the same service that are instantiated in different infrastructure nodes. An example is provided in Figure 5.4, where the overarching orchestrator splits the original service graph into two subgraphs; as shown, the `firewall` and the `adv-blocker` are then attached to GRE SAPs representing a GRE tunnel between the IP addresses `1.1.1.1` and `2.2.2.2`, and identified by the key `0x1`. This way, traffic exiting from the `firewall` is encapsulated, by COMPOSER, into the GRE tunnel and then delivered to the `adv-blocker` by the network infrastructure, and vice versa. Consequently, SAPs enable NFs to operate irrespective of the nature of the network connection, because the actual delivery of the traffic to the next NF in the chain is

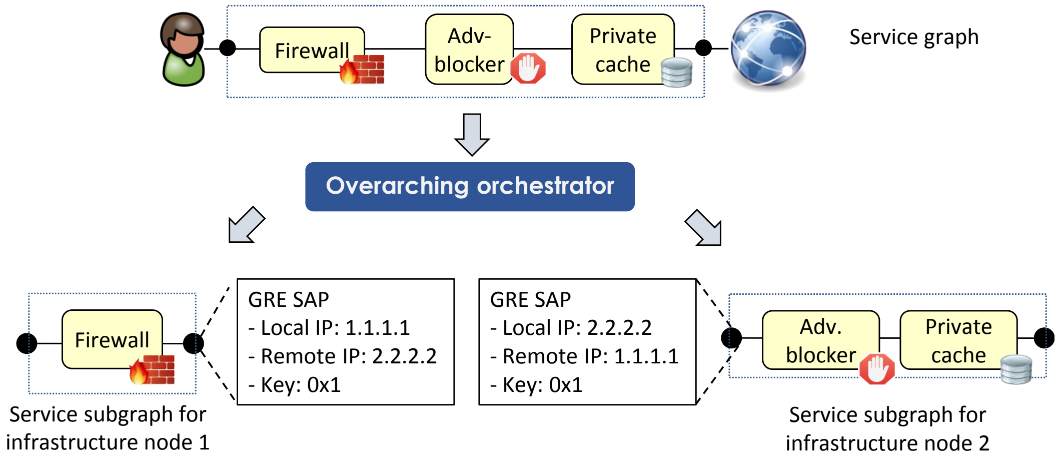


Fig. 5.4 Using *GRE* SAPs to set up traffic steering between subgraphs.

done transparently by the infrastructure node, e.g., by encapsulating packets in a given VLAN or in a tunnel toward a remote destination.

Finally, the `flowrules` section in Figure 5.2 describes the interconnections between NF ports and SAPs. We opted to use semantics akin to OpenFlow whereby each connection is characterized by: *(i)* a priority; *(ii)* a match on a SAP/NF port and potentially on protocol fields (e.g., IP source); *(iii)* an action that forwards packets through a specific SAP/NF port and that potentially modifies the packet content (e.g., decrease the IPv4 TTL).

Domain Description

The COMPOSER domain description published by c-orch includes both network and compute characteristics of the infrastructure node. From the network point of view, COMPOSER is abstracted as a "*big switch*" with a set of endpoints, each one characterized with the following (optional) information: *(i)* neighbor domain, i.e., the identifier of another infrastructure node (e.g., COMPOSER) in case there is a direct connection with it; *(ii)* IP address; *(iii)* support for VLAN traffic: in this case the endpoint indicates the available VLAN IDs; *(iv)* support for GRE tunnels.

From the compute point of view, COMPOSER exports functional and infrastructure-level capabilities, as well as available resources, as mentioned earlier. The multi domain orchestration will be further analyzed on Chapter 6.

5.3.2 Traffic steering model

As shown in Figure 5.1, c-orch implements the network paths between NF ports and SAPs through two LSI layers. The foundation `LSI-0` is overlayed by a set of LSIs (`graph-LSI`), each one in charge of implementing paths between NFs of a different graph. `LSI-0` is created at boot time and dispatches the traffic from the COMPOSER physical interfaces to the graph-LSIs, while additional LSIs (each one created when a new NF-FG has to be deployed) implement the traffic steering paths among the NFs and GRE SAPs that belong to that graph. In fact, while physical interfaces are connected to `LSI-0`, these ports are connected to the associated graph-LSI.

Notably, as each graph-LSI is connected to NFs of a different NF-FG, c-orch can implement multi-tenancy and isolate traffic of different tenants with `LSI-0` being the only LSI traversed by packets belonging to multiple tenants/service graphs.

Moreover, the LSI hierarchy takes care of removing encapsulations used to implement the vlan and the GRE SAPs, in case of traffic arriving from such SAPs. Similarly, NFs are not aware that traffic they transmit will be sent, by the LSI hierarchy, through a vlan/GRE SAP; also in this case, in fact, it is the LSI hierarchy that encapsulates packets in the proper headers, according to the flow rules described in the NF-FG.

In addition, none of the LSIs can be programmed by an external SDN controller, belonging, e.g., to the owner of the service graph. LSIs are in fact exploited by c-orch to implement the network paths described in the NF-FG, and are under the complete control of c-orch (through the network manager, as detailed below).

Finally, LSIs provide to COMPOSER complete control on the NFs networking, hence it can implement network connections as defined in the NF-FG. In contrast, note that the standard networking models offered, e.g., by KVM and Docker, do not provide such full control of networking paths, since they attach VMs/containers to a L2 bridge only.

5.3.3 Node resource manager

The *node resource manager* is the main module of c-orch, as it handles the commands received through the REST API and exports the node description both at boot time as well as each time that something changes in its configuration.

According to the message sequence diagram of Figure 5.5, when c-orch receives a command to create a new NF-FG, the node resource manager: *(i)* interacts with the NF repository in order to select the most appropriate NF image for each function that is part of the service and that is not explicitly associated with an image in the NF-FG; *(ii)* configures the vSwitch to create a new LSI and the ports required to connect it to the NFs to be deployed; *(iii)* deploys and starts the selected NFs; and *(iv)* configures the forwarding table(s) of the LSI(s) according to the required traffic steering rules. Similarly, the node resource manager takes care of updating or destroying a graph, when the corresponding commands are received.

Figure 5.1 shows that c-orch includes, among other modules, the *network manager* and the *compute manager*, which are exploited by the node resource manager to interact respectively with the vSwitch and the execution engines to create the proper paths and start the proper NFs. The *NF resolver* interacts with the NF repository and selects the best implementation for the required functions, according to parameters such as the amount of compute and memory resources available on COMPOSER and constraints associated with the requested NF (e.g., 3 ports). Finally, the *NF scheduler* can optimize the NF/CPU core(s) binding(s) by taking into consideration information such as how a NF interacts with the rest of the NF-FG.

5.3.4 Network manager

The *network manager* is the module that handles the networking part of the service graph. It can interact with different vSwitches in order to create the virtual network infrastructure that implements the paths described in the NF-FG; such an architecture supports the parallel instantiation of multiple service graphs according to the traffic steering model presented in Section 5.3.2.

Setting up the paths described in the NF-FG requires the network manager to interact with the vSwitch through the *management* and *control* interfaces. The former is used to create a new graph-LSI with the required virtual ports that will be later

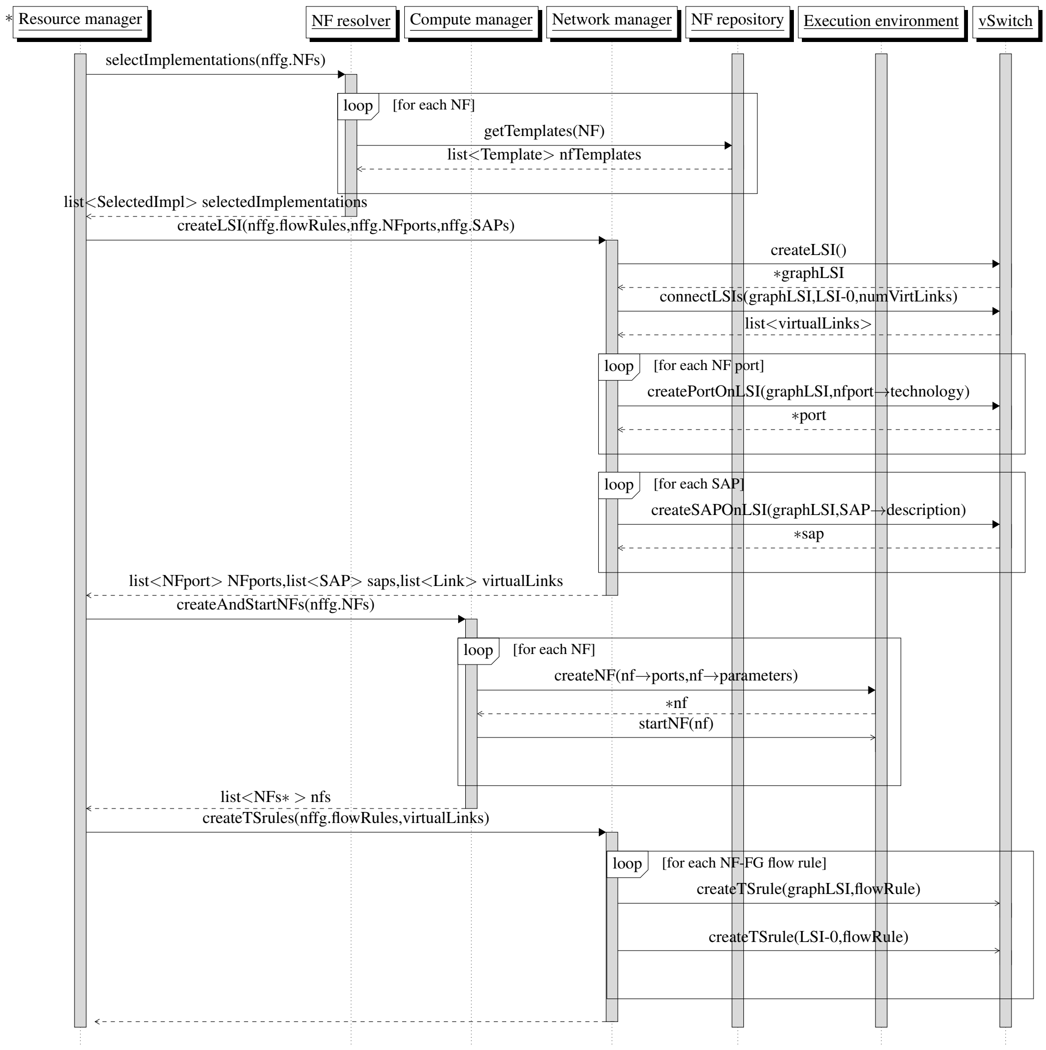

Resource manager
NF resolver
Compute manager
Network manager
NF repository
Execution environment
vSwitch
selectImplementations(nffg.NFs)
loop [for each NF]
getTemplates(NF)
list<Template> nfTemplates
list<SelectedImpl> selectedImplementations
createLSI(nffg.flowRules,nffg.NFports,nffg.SAPs)
createLSI()
*graphLSI
connectLSIs(graphLSI,LSI-0,numVirtLinks)
list<virtualLinks>
loop [for each NF port]
createPortOnLSI(graphLSI,nfport→technology)
*port
loop [for each SAP]
createSAPOnLSI(graphLSI,SAP→description)
*sap
list<NFport> NFports,list<SAP> saps,list<Link> virtualLinks
createAndStartNFs(nffg.NFs)
loop [for each NF]
createNF(nf→ports,nf→parameters)
*nf
startNF(nf)
list<NFs*> nfs
createTSrules(nffg.flowRules,virtualLinks)
loop [for each NF-FG flow rule]
createTSrule(graphLSI,flowRule)
createTSrule(LSI-0,flowRule)

Fig. 5.5 c-orch new NF-FG deployment message sequence diagram.

Table 5.1 Management interface.

Function	Description
lsi ∗createLSI()	Create a new LSI
void destroyLSI(lsi)	Delete a specific LSI
list<∗link> connectLSIs(lsi1,lsi2,N)	Create `N` virtual links between two LSIs
void destroyVlink(link)	Destroy a virtual link between two LSIs
port ∗createPortOnLSI(lsi,technology)	Create a (NF) port with a specific technology on an LSI
void destroyPort(port)	Destroy a specific (NF) port
sap ∗createSAPOnLSI(lsi, description)	Create a SAP on an LSI, according to a specific description
void destroySAP(sap)	Delete a specific SAP

Table 5.2 Control interface.

Function	Description
void createTSRule(lsi,rule)	Insert a traffic steering rule in the LSI
void deleteTSRule(lsi,rule)	Remove a traffic steering rule from the LSI

attached to the NFs[1]. The latter is used to program the forwarding tables of LSI-0 and of the new graph-LSI in order to realize traffic steering.

The management interface, through the set of primitives listed in Table 5.1, enables the network manager to interact with different vSwitches without knowing anything about their switching technology. These primitives are implemented by a set of technology-specific drivers and enable the network manager to *(i)* create/destroy an LSI, *(ii)* create/destroy a port that will be then connected to a NF, *(iii)* create/destroy virtual links between two LSIs, and *(iv)* create/destroy SAPs.

Similarly, the control interface of the network manager enables the configuration of the forwarding table(s) of the LSIs by means of multiple technologies (e.g, OpenFlow [3], eBPF [61], P4 [62]), while at the same time hiding from the network manager the actual technology used. The set of primitives defined by the control interface is listed in Table 5.2 and must be implemented by the technology-specific controllers in order to enable the network manager to insert/remove traffic steering rules in/from a specific LSI.

As shown in Figure 5.1, a different technology-specific controller is created for each LSI, which controls the forwarding table of the LSI itself. Similarly to the management interface, multiple technologies to program the forwarding table(s) of

[1]The technology of the virtual ports depends on the NF image selected. For instance, in the case of a DPDK-enabled process, `dpdkr` ports must be used to interconnect the vSwitch with said NF.

Table 5.3 LSIs involved in implementing flow rules with specific match/action.

		ACTION: output to	
		Interface/vlan SAP	**NF port/GRE SAP /host stack**
MATCH	**Interface/vlan SAP**	only LSI-0 (no vlink needed)	LSI-0 and graph-LSI
	NF port/GRE SAP/host stack	LSI-0 and graph-LSI	only graph-LSI (no vlink needed)

the vSwitch can be supported by writing the corresponding driver implementing the primitives of Table 5.2.

Translating NF-FG flow rules for traffic steering

In order to implement the network paths described in the NF-FG, the network manager has to map the `flowrules` requested by the NF-FG on the traffic steering model defined in Section 5.3.2. Particularly, this model requires that, for each NF-FG to be deployed, the network manager: *(i)* connects the new LSI with `LSI-0` through a number of virtual links, and that, *(ii)* starting from the `flowrules` section of the NF-FG, originates two sets of traffic steering rules to be installed respectively in `LSI-0` and in the new graph-LSI.

According to Table 5.3, some flow rules can be implemented on a single LSI, because both the match and the action involve ports/SAPs that are connected to the same LSI, while other rules must be split in one traffic steering rule for the `LSI-0` and in another for the `graph-LSI`. While the former flow rules do not require any virtual link, as they keep traffic local to one LSI, the latter need virtual links to transfer packets between the two LSIs.

Particularly, as shown in Algorithm 8, we chose to create a different virtual link for each *different* SAP/NF port that appears in the action of rules involving both LSIs (Table 5.3), which is then used to move all traffic that must be sent on that specific SAP/NF port from one LSI to another. According to the pseudocode, in order to minimize the number of virtual links, the same virtual link can be used both to send towards the `graph-LSI` all traffic for a specific NF port/GRE SAP/host stack SAP, and to send towards `LSI-0` all traffic for a specific interface or vlan SAP.

Transforming a NF-FG `flowrule` that can be implemented on a single LSI in a traffic steering rule for such an LSI does not require any operation (an exam-

Algorithm 8 Virtual links creation.

```
 1: procedure createVlinks(first_id,nffg,lsi0,graphLsi)
 2: vlink_to_lsi0 ← vlink_to_graphlsi ← first_id
 3: vlinks ← $\emptyset$
 4: for all r ∈ nffg.flowrules do
 5:     if vlink_needed[r.match.port][r.action.out] and vlinks[r.action.out] ∈ $\emptyset$ then
 6:         if r.action.out ∈ nf_port or r.action.out ∈ gre_sap or r.action.out ∈ hoststack_sap
            then
 7:             vlinks[r.action.out] ← vlink_to_graphlsi
 8:             vlink_to_graphlsi ← vlink_to_graphlsi+1
 9:         else if r.action.out ∈ interface_sap or r.action.out ∈ vlan_sap then
10:             vlinks[r.action.out] ← vlink_to_lsi0
11:             vlink_to_lsi0 ← vlink_to_lsi0+1
12:         end if
13:     end if
14: end for
15: N ← max(vlink_to_lsi0,vlink_to_graphlsi)−first_id
16: return connectLSIs(lsi0,graphLsi,N)
```

ple is provided by the NF-FG `flowrule #3` of Figure 5.6). Algorithm 9 shows how the network manager derives the traffic steering rules corresponding to NF-FG `flowrules` involving both LSIs. After that the NF ports/SAPs have been associated with one of the virtual links just created. Particularly, according to lines `#4-#12` of Algorithm 9, rules whose output port is connected to the `graph-LSI` generate two traffic steering rules as follows. The match of the `LSI-0` rule corresponds to the match of the original rule (line `#7`), while the action differs from the original action only in the output port field; in fact, it forwards packets on the virtual link that transfers to the `graph-LSI` all packets towards the original port (line `#8`). Consequently (lines `#11-#12`) the rule for the `graph-LSI` just matches the proper virtual link and forwards traffic on the output port of the original rule. This behavior can be observed in the NF-FG `flowrule #1` of Figure 5.6, where `port 1` of the `NAT` is associated with the virtual link `vlink1`.

Lines `#13-#21` of Algorithm 9 manage flow rules whose action sends packets on a port connected to `LSI-0`. Unlike in the previous case, now it is the match of the `graph-LSI` rule that corresponds to the match of the original rule (line `#19`), as well as it is the action of the rule on such an LSI that is equal to the original action except for the output port field, that corresponds to the virtual link that brings to `LSI-0` all the packets towards the original output port. Finally, lines `#16-#17`

Algorithm 9 Traffic steering rules creation.

```
 1: procedure splitRules(vlinks,nffg)
 2: for all r ∈ nffg.bigswitch do
 3:     if vlink_needed[r.match.port][r.action.out] then
 4:         if r.action.out ∈ nf_port or r.action.out ∈ gre_sap or r.action.out ∈ hoststack_sap
            then
 5:             {The rule brings traffic from LSI-0 to the graph-LSI}
 6:             {Create the rule for LSI-0}
 7:             rule-LSI0.match ← r.match
 8:             rule-LSI0.action.out ← vlinks[r.action.out]
 9:             rule-LSI0.action.other ← r.action.other
10:             {Create the rule for the graph-LSI}
11:             rule-graphLSI.match.port ← vlink[r.action.out]
12:             rule-graphLSI.action.out ← r.action.out
13:         else if r.action.out ∈ interface_sap or r.action.out ∈ vlan_sap then
14:             {The rule brings traffic from the graph-LSI to LSI-0}
15:             {Create the rule for LSI-0}
16:             rule-LSI0.match.port ← vlink[r.action.out]
17:             rule.LSI0.action.out ← r.action.out
18:             {Create the rule for the graph-LSI}
19:             rule-graphLSI.match ← r.match
20:             rule-graphLSI.action.out ← vlink[r.action.out]
21:             rule-graphLSI.action.other ← r.action.other
22:         end if
23:     end if
24: end for
```

show that the rule created for `LSI-0` just matches the proper virtual link. An example of this procedure is shown for the NF-FG `flowrule #2` in Figure 5.6, where the virtual link used to transfer traffic to `eth1` is the same one used to bring traffic to the port of the NAT.

5.3.5 Compute manager

The *compute manager* interacts with the available execution environments to manage the NF lifecycle, including operations needed to attach NF ports already created on the vSwitch (by the network manager) to the NF itself. The compute manager module can interact with different execution engines, and can thus manage NFs based on different technologies, through the *compute interface* defined in Table 5.4.

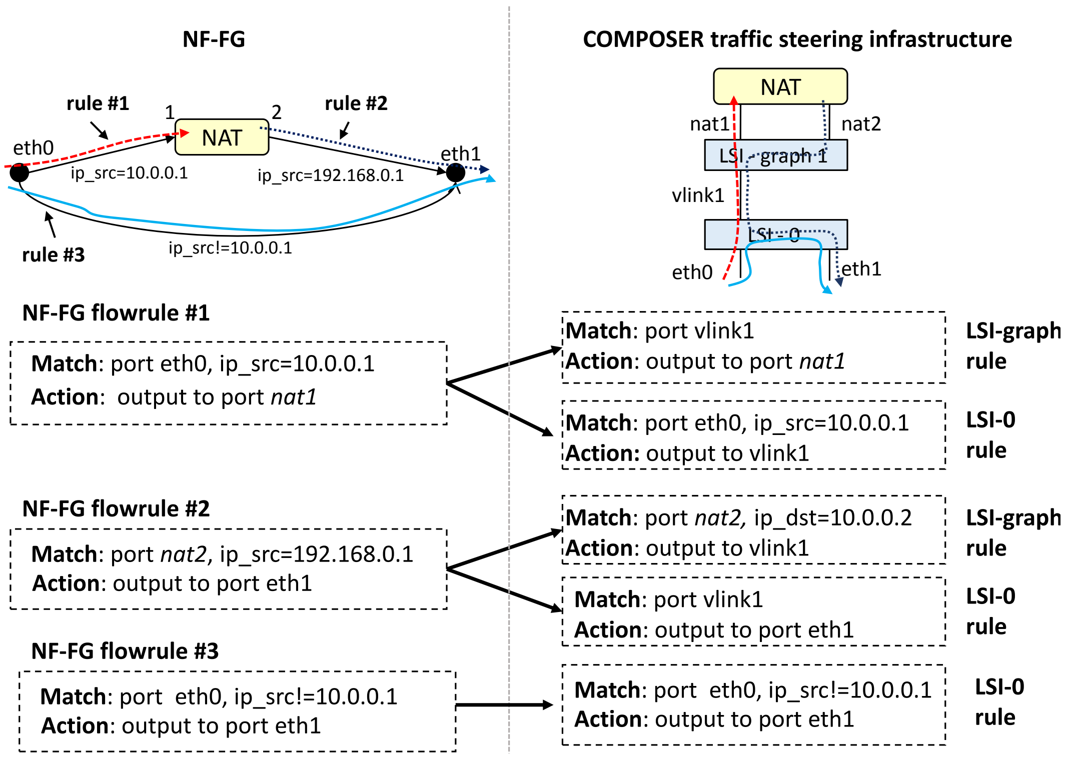

NF-FG
COMPOSER traffic steering infrastructure
rule #1
rule #2
rule #3
1
2
NAT
eth0
eth1
ip_src=10.0.0.1
ip_src=192.168.0.1
ip_src!=10.0.0.1
nat1
nat2
LSI - graph 1
vlink1
LSI - 0
NF-FG flowrule #1
Match: port eth0, ip_src=10.0.0.1
Action: output to port nat1
Match: port vlink1
Action: output to port nat1
LSI-graph rule
Match: port eth0, ip_src=10.0.0.1
Action: output to vlink1
LSI-0 rule
NF-FG flowrule #2
Match: port nat2, ip_src=192.168.0.1
Action: output to port eth1
Match: port nat2, ip_dst=10.0.0.2
Action: output to vlink1
LSI-graph rule
Match: port vlink1
Action: output to port eth1
LSI-0 rule
NF-FG flowrule #3
Match: port eth0, ip_src!=10.0.0.1
Action: output to port eth1
Match: port eth0, ip_src!=10.0.0.1
Action: output to port eth1
LSI-0 rule
Fig. 5.6 Example of transformation of NF-FG `flowrules` in traffic steering rules.

Table 5.4 Compute interface.

Function	Description
nf ∗createNF(ports,other parameters)	Allocate the resources needed by a NF; create a local copy of/download the NF image
void destroyNF(nf)	Release the resources allocated to the NF
void startNF(nf)	Start a NF previously created
void stopNF(nf)	Stop a NF, without deallocating resources
void updateNF(nf,. . .)	Update a running NF (e.g., remove/add network interfaces)
void pause(nf)	Suspend the execution of the NF (e.g., for a possible migration)

As shown in Figure 5.1, this abstraction is implemented by a set of drivers, each one in charge of a specific execution environment technology. The COMPOSER can support QEMU/KVM hypervisor, Docker containers, processes based on the DPDK framework [20], and native network functions. Multiple technologies are supported at the same time. For instance, c-orch can deploy a service including a first NF executed in a Docker container and a second NF running inside a VM.

5.3.6 NF resolver

The *NF resolver* is the part of c-orch that interacts with the NF repository in order to select the NF images to be instantiated. The NF resolver is used when the NF-FG potentially allows multiple NF images to fulfill the requirements (e.g., *firewall with 3 ports*). In this case, it selects the best image through the following steps. First, it asks to the NF repository the templates of all NFs implementing the function (e.g., firewall). Subsequently, it selects the best NF that, according to the template, satisfies the constraints/attributes associated with the function in the NF-FG (e.g., 3 ports), matches an execution environment supported by COMPOSER and requires resource levels (e.g., RAM, CPU) available in COMPOSER.

Notably, the selected NF may consist of a single image or it can be a new service graph composed of a number of other functions arbitrarily connected. In other words, the template may describe the NF as another NF-FG, according to the principle of recursive decomposition. In this case, the NF resolver recursively repeats the operations described above for each function that is part of the new *sub*-NF-FG, until all required NF images have been selected.

NF repository

The NF repository contains the templates and images of the available NFs. The *NF template* describes a specific NF image in terms of functionality implemented (e.g., firewall, NAT), amount of physical resources required on the node in order to execute such an image (e.g., CPU, memory), required execution environment (e.g., KVM hypervisor, Docker engine, etc.), number of virtual interfaces and associated technology, and more. The *NF image* varies according to the technology implementing the NF. For instance, it is the VM disk in case of virtual machines, a tgz archive in case of NNFs, and so on.

5.3.7 Internal message bus

As shown in Figure 5.1, COMPOSER includes an internal message bus. Although in the picture only c-orch, the monitoring manager and the monitoring functions (Section 5.3.8) are connected to such a bus, NFs may be connected to the bus as well, for instance in order to receive monitoring alarms or configuration parameters.

5.3.8 Monitoring manager

The *monitoring manager* is in charge of managing the modules that *(i)* measure some metrics of the deployed service (e.g., CPU/memory consumed by NFs), and *(ii)* generate alarms when specific events occur or thresholds are exceeded. The monitoring manager can be configured in order to measure specific metrics through an instruction string specified in the NF-FG and written according to the MEASURE monitoring language [63].

After the deployment of the NF-FG, the monitoring manager instantiates and configures the proper NFs (e.g., *Google cAdvisor* [64] and *Ramon* [65]) that monitor the required metrics and generate alarms on the COMPOSER internal bus.

The monitoring manager, in addition to starting and configuring the proper NFs for monitoring, it receives alarms, aggregates the received information as required by the MEASURE instructions, and propagates them again. This way, the aggregated information can reach the interested NFs. For instance, a NF may exploit the monitoring results to require an update of the NF-FG, so that it can properly react to the received event. The consumer of the alarm can be one (or some of the NFs) that are part of the NF-FG.

5.4 Experimental evaluation

COMPOSER was used as a validation platform for the FP7 projects UNIFY [66] and SECURED [67]; in addition, we carried out an extensive set of tests that are reported in the reminder of this section. In particular, we will first present the many hardware platforms that we used to validate our proposal, by highlighting the limitations found and the issues we had to solve to run the COMPOSER software and NFs on them. Then we will show the resource consumption (memory and CPU) of the

Table 5.5 Machines used in the validation.

Machine	Specification
Residential gateway #1	Netgear R6300v2, 800 MHz dual-core ARM Cortex A9 CPU, 128 MB flash 256 MB RAM, 4GbE LAN ports, IEEE 802.11 b/g/n 2.4GHz, IEEE 802.11 a/n/ac5.0GHz, 1 GbE WAN port
Residential gateway #2	Banana Pi R1, A20ARM Cortex-A7 dual-core CPU, 1 GB RAM, 5 GbE ports
Professional CPE #1	Hawkeye HK-0910, Freescale QorIQ T1040, 1.2GHz (quad e5500 cores), 64MB NORFlash, 2GB RAM DDR3L-1600
Professional CPE #2	Tiesse Imola 5, Ikanos Fusiv Core Vx185, single core MIPS 34Kc V5.4 CPU @ 500MHz, 256MB RAM, 256MB flash memory, xDSL acceleration
Mid-range server #1	Intel Core i5-3450S @ 2.8 GHz, 8GB RAM, 200GB SSD
Mid-range server #2	Intel Core i7-4770 @ 3.40 GHz (4 cores + hyperthreading), 32GB RAM, 500GB HD
High-end server	Intel Xeon E5-2690 v2 @ 3 GHz (10 cores + hyperthreading), 64 GB RAM, two 10G Intel 82599ES NICs

COMPOSER software, including also the vSwitch and the execution engine(s); such results are compared with OpenStack, as it is the platform typically used to execute NFs. We will then demonstrate how COMPOSER can deploy network services on a residential CPE thanks to the NNFs, which are well suited for resource constrained environments.

5.4.1 Hardware platforms

The portability of COMPOSER has been demonstrated by installing and running the software on the hardware platforms shown in Table 5.5, which feature very different characteristics (also in terms of supported operating system and software build mechanisms) and represent the huge variety of hardware running in carrier network deployments.

For instance, a residential gateway (Netgear R6300v2) has been demonstrated to execute COMPOSER compiled for the OpenWrt operating system. Given the limited hardware capabilities of this box, at the time of this writing c-orch was able to launch only NNFs with OvS used as a vSwitch. In addition, service access points such as tunnels (e.g., GRE) and VLANs were supported, hence enabling to connect COMPOSER to external domains and to create complex services requiring the stitching of multiple sub-graphs spanning across multiple domains [68]. Another example is the use of COMPOSER on CarOS, an embedded Linux distribution targeting carrier networks, running on a Banana Pi R1.

Two professional CPEs were used as well. First, we targeted the Freescale Hawkeye HK-0910, featuring also IPsec and L2 switch hardware acceleration. The software environment was based on the Linux Yocto project, which uses *recipes* to assemble together the required packages and create the software image that will be executed on the hardware platform. The overall software setup was very similar to the previous boxes, hence only NNFs are enabled, although the platform could support also virtualization.

The second professional CPE was a Tiesse Imola 5 with customized OpenWrt as operating system. However we had to use an old version of the Linux kernel (3.10.49) because it was the latest version supported by the drivers needed to control the xDSL interface. This prevented us from terminating GRE tunnels in OvS, due to a known incompatibility with that kernel version. The workaround solution consisted of modifying the OvS driver in the network manager so that, when executed on this platform, it first creates a GRE tunnel port through the Linux command `ip link`, and then adds this port to the proper graph-LSI.

In addition to the aforementioned limited-resource boxes, COMPOSER was tested on several standard Intel servers, ranging from single CPU i5 machines to dual-processor Xeon platforms running Ubuntu 14.04 LTS. All features, including the several supported vSwitches, were turned on and tested.

5.4.2 Empirical evaluation of resource consumption

This section presents the amount of RAM and disk consumed by different COMPOSER deployments and compares them with the requirements of an OpenStack compute node supporting VMs, as it the most widespread technology for running NFs as of this writing.

Starting from a clean installation of Ubuntu server 14.04 LTS with default settings on the high-end server (Table 5.5), we set up, one at a time, the configurations shown in Table 5.6, where each line reports the additional resources required with respect to the case with the clean operating system running. Note that the reported numbers consider all components needed to execute NFs, including the vSwitch; particularly, OvS was used for the tests reported in this section.

Table 5.6 Resources consumed by COMPOSER and OpenStack.

Configuration	**RAM (MB)**	**Disk (MB)**
COMPOSER- only NNF enabled	31	71
COMPOSER- only Docker enabled	46	189
COMPOSER- only KVM enabled	44	131
COMPOSER- all env. enabled	63	249
OpenStack compute node	160	494

As reported in Table 5.6, COMPOSER compiled only with support for NNFs represents the lightest configuration, requiring less than 20% of RAM and less than 15% of disk space than an OpenStack compute node, as NNFs are executed natively by a compute manager which launches shell scripts. KVM and Docker are more resource demanding than NNFs as expected because they need to install and run the respective execution engines (KVM, QEMU and Libvirt in the first case, Docker engine in the latter). COMPOSER with Docker only enabled, for example, uses less that 29% RAM and less than 38% of the disk space required by our benchmark OpenStack compute node. As expected, the most resource consuming COMPOSER configuration is when all the (currently) supported execution environments are enabled. Still, even in this case, COMPOSER requires less than 40% of RAM and about half of the disk space that an OpenStack compute node requires.

Our empirical measurement of actual resources used confirm the advantages, in terms of resource requirements, for COMPOSER when compared to OpenStack; moreover, they also show how NNFs are well-suited for resource constrained environments such as CPEs.

5.4.3 Service deployment time

This section compares the time required by COMPOSER and OpenStack to deploy the service graph shown in Figure 5.7(a), which deploys the firewall NF with a dedicated specific VM. Notably, the two SAPs of the graph (i.e., `ext0` and `ext1`) correspond to physical interfaces in COMPOSER, and to external networks in case of OpenStack.

Tests are executed on the mid-range server (Intel i7) (Table 5.5) with a clean Ubuntu server 14.04 LTS image, where we added all the components involved in the service deployment, namely: *(i)* the COMPOSER software and the NF repository

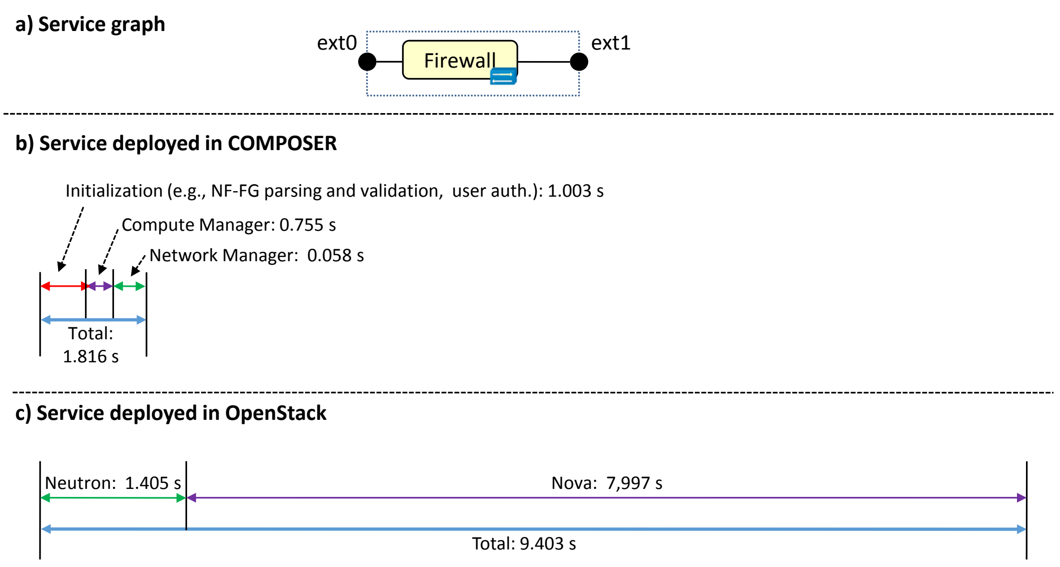


Fig. 5.7 Service deployment time: *(a)* used service graph; *(b)* results with COMPOSER; *(c)* results with OpenStack.

in case of COMPOSER; the Nova, Neutron and Glance (i.e., the compute/network services, and VM repository) servers, and the compute node actually running the VMs in case of OpenStack. Both COMPOSER and OpenStack use OvS as vSwitch; moreover, the VM image is already cached by the OpenStack software in order to avoid downloading the VM image.[2]

Tests are repeated 10 times and averaged; results are reported in Figures 5.7(b) and 5.7(c). In case of COMPOSER (Figure 5.7(b)), the picture reports the total time required by c-orch to serve the request, which is then broken into: *(i)* the time used by the network manager to create the new LSI and the NF ports, and to set up the traffic steering rules into the vSwitch; *(ii)* the time required by the compute manager to create and start the NF as a VM. In the case of OpenStack, we report the total time needed for the service deployment, which includes the time spent to interact with Neutron and Nova so that they fulfill the service request. It is worth mentioning that the results do not include the time needed by the application in the VM (i.e., the firewall) to actually start, which depends from the software executed in the VM itself and not from the orchestration framework and is anyway equivalent in the two tests, given that we used exactly the same VM for the NF.

[2]Installing the VM repository and the compute node on the same physical machine does not prevent OpenStack to download the VM image through the repository REST API, in case it is not in cache yet; this would have a huge (negative) impact on the overall service deployment time.

As shown, the time required by COMPOSER is nearly an order of magnitude lower than the one needed by OpenStack to deploy the same service graph; this is due to the interactions between the various OpenStack components involved in the service deployment (i.e., Nova/Neutron servers, compute/network agents), which employ either REST or distributed message bus (RabbitMQ) calls.

5.4.4 Native network functions

This section validates the NNF idea from the point of view of throughput, CPU load, image size and time required to start the NF. To this purpose, we consider a transparent VPN access use case in which a user client located in a trusted local network (e.g., home) needs to connect to its corporate VPN server. In order to avoid to install the VPN client software on all his devices (e.g., laptop, smartphone, etc.), the user deploys the VPN client as a NF on the CPE, which provides secure access to the corporate network independently of the specific user terminal.

Our testbed, shown in Figure 5.8, includes two devices acting both as traffic source and sink, a CPE executing the IPsec client NF in charge of encrypting/decrypting the traffic, and a VPN server with the corresponding duty. All four boxes are connected with point-to-point full-duplex 1Gbps Ethernet links; faster speed were not available due to the limitations of the current hardware. Three powerful workstations were used respectively as traffic source/sink (two machines) and VPN server, in order to avoid those machines to become a bottleneck in our test setup, while different flavors of CPEs were used, namely a mid-range server (Intel i5), a professional CPE based on the Freescale T1040 and a residential gateway (Netgear) (Table 5.5), all with the same COMPOSER version compiled for the respective platform. The use of different hardware platforms was coupled with different implementations of the same NF, whenever possible, as shown in Table 5.7.

The experiments leveraged the `iperf` tool installed on the two source/sink machines, each one configured to generate unidirectional TCP streams at maximum speed; all experiments were repeated 10 times and averaged. According to Table 5.7, NNFs and Docker bring significant performance improvements compared to VMs because of the simplified architecture that requires neither the hypervisor nor the guest OS. As expected, NNFs and Docker show the same level of performance,

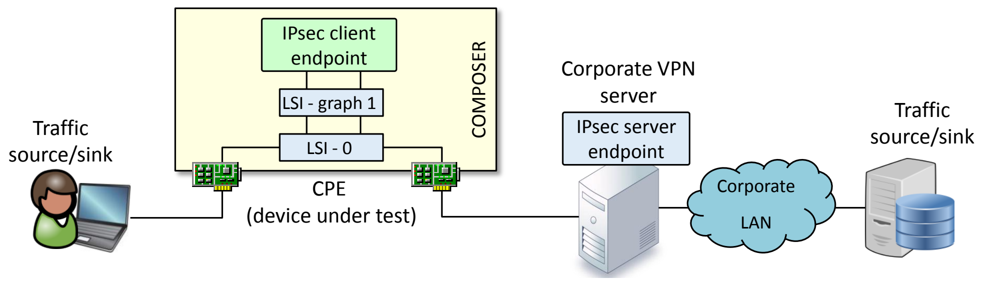


Fig. 5.8 Testbed to validate the COMPOSER when running NNFs.

Table 5.7 Different implementations of the *IPSec client* NF.

IPsec client NF implementation	**Bidirectional Throughput (Mb/s)**	**CPU Load**	**NF Image Size (MB)**
Mid-range Server #1 - KVM/QEMU	796	100%	522
Mid-range Server #1 - Docker	1095	80%	240
Mid-range Server #1 - NNF	1094	80%	5
Residential gateway #1 - NNF	57.2	100%	2
Professional CPE #1 - NNF	617	90%	3.7

because they are based on the same technology (i.e., kernel-based processing in the host plus namespaces).

The last column of Table 5.7 reports the NF image size[3], which confirms the advantages of the NNF approach in resource-constrained environments. In fact, the reason for not testing VMs and Docker on the residential gateway and on the professional CPE is the disk size limitation of these platforms. Moreover, the NF image size also impacts on the time required to download the NF from a remote location, which is critical when the CPE is connected to the Internet through slow links (e.g., xDSL).

Finally, we measured the time to make the IPsec client fully operational on the mid-range server, being the only environment supporting all NF types. Averaged results show 3016 ms with VM (which requires to start the entire VM), 350 ms with Docker, and 727 ms with NNF (i.e., with the IPsec client running in a separate network namespace); the baseline, i.e., the time required to launch the IPsec client on the base system without wrapping it in any virtualization environment, was 154 ms. The (relatively) high number of the NNF is due to some implementation-dependent

[3]In the VM case, we created a guest OS with the default installation of a Ubuntu server 14.04 LTS plus the only packages required for our NF to work.

timeouts required to attach the network ports to the NNF, which still need to be optimized.

5.5 Related work

Based on the ETSI architecture, both industry and academia introduced several prototypes and proofs of concept (PoCs) to deploy network services and functions. However, while COMPOSER defines the software architecture of an infrastructure node that also includes orchestration and VIM functionalities in addition to a number of VNFMs that can be instantiated as NFs and are part of a service graph, most of said earlier works define the architecture of the NFVO that sits on top of many infrastructure nodes and deploys network services through OpenStack [69]. In these cases, OpenStack acts as a VIM, exploited by the NFVO to properly instantiate the service on the OpenStack physical infrastructure.

Proposals employing OpenStack as a VIM and VNFI include OpenStack Tacker [70] which implements the NFVO and a generic VNFM, and uses the *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [71] as a formalism to describe the various aspects of the service to be deployed, which is an implementation of the ETSI descriptors. Open Baton [72] defines a NFVO and a generic VNFM and can be installed on top of existing cloud infrastructures like OpenStack. OpenMANO [73] implements instead its own VIM (in addition to an NFVO sitting on the top of many infrastructure node), although it supports OpenStack as well.

It is worth noting that these proposals are orthogonal to our work on COMPOSER, as they mainly operate on top of the infrastructure nodes and can be extended to interact with c-orch instead of the OpenStack environment.

Cloud4NFV [74, 75] is a platform for managing network services in a cloud environment, which covers both the NFVO and the VIM functionalities defined in the ETSI architecture. The latter includes both a data center controller (OpenStack) and a WAN controller (OpenDaylight) to interconnect parts of the service deployed in different data centers. OpenStack and OpenDaylight are used as VIM also by vConductor [76], while SONATA [77] can support different VIMs by means of adapters; moreover, SONATA supports recursion at the orchestrator layer. Recursion was considered also in the Unify project [66], where two orchestrators [78], [60] that can sit on top of different infrastructures have been defined.

Compared to COMPOSER, OpenStack presents several limitations when used as VIM and NFVI. For instance, it does not have the concept of network service, which means that each NF is just seen as a VM (or container) to be executed independently from the others on one of the servers forming an OpenStack cluster. Hence, VMs are allocated to server/CPU cores without taking into consideration connections between NFs in the service to be deployed, which may result in poor performance for the whole service. An attempt to introduce the concept of service in OpenStack is described in [79], which also highlights how the so-called *network-aware scheduling* is hard to be implemented in such an environment. Moreover, OpenStack is not suitable for resource-constrained environments such as CPEs, as shown in Table 5.7. Instead, OpenStack can be used to implement a virtual CPE such as in [80], in which the traditional CPE functions are implemented as VNFs running in an OpenStack-based data center.

Literature on NFV also includes some proposals of software architectures of network infrastructure nodes that, similarly to COMPOSER, can execute network functions. NetVM [32] is a platform designed to efficiently transfer packets between NFs running inside virtual machines, which mainly focuses on the data plane and marginally considers control and orchestration aspects. NetVM defines its own virtual switch based on the DPDK framework, which can transfer packets with zero-copy between *trusted* VMs, while a copy is required to transfer packets between untrusted VMs. Moreover, existing network applications are not supported by NetVM as they must use a library that hides the communication with the NetVM framework. The NetVM architecture includes a NetVM manager that can talk with an overarching orchestrator by means of a message based protocol similar to OpenFlow, although no more information is provided in [32].

OpenNetVM [81, 82] and SDNFV [83] are platforms built on top of NetVM, which execute ad-hoc DPDK-based NFs within Docker containers and provide a high-level abstraction to compose NFs in service chains, control packet flows, and manage NF resources. Particularly, the NFs that have to process a packet can be selected both by an SDN controller (not part of the OpenNetVM framework) and by NFs, which can program the vSwitch forwarding table without the necessity to interact with said controller. Notably, SDNFV [83] also focuses on creating paths among VNFs deployed on multiple hosts.

ClickOS [30] presents a platform for high-performance NFV. Particularly, it uses the VALE vSwitch [13] to provide packets to ClickOS virtual machines, i.e., Xen-based [17] VMs executing a Click [84] program running on top of a minimal operating system. Unlike COMPOSER, which supports many execution environments and focuses on control and orchestration aspects (in addition to exploiting optimized data paths), ClickOS only focuses on performance of the data plane as it solves bottlenecks in the network I/O of the Xen hypervisor, and runs applications explicitly designed for the ClickOS environment.

nf.io [85] is a platform that employs the Linux file system as an interface to express NFV management and orchestration operations and acts as an API towards the NFVO. Particularly, nf.io defines the semantics of files and directory structures to perform operations such as VNF deployment, configuration, chaining and monitoring. Similarly to COMPOSER, nf.io can execute NFs as processes on physical machines, VMs, Docker and LXC containers. Forwarding rules can be configured both with the `iptables` Linux facility or with OpenFlow for traffic paths implemented using OvS.

GNFC [86] and GLANF [87, 88] are frameworks to deploy VNFs. In addition to a manager that allocates VNFs to servers, and a network controller that configures traffic steering on the NFV servers and among different servers, these frameworks define an agent per server responsible of managing (e.g., start, stop, connect them to OvS) VNFs. However, this agent simply manages Docker containers and creates ports on OvS, while COMPOSER includes an orchestrator on each server which can further optimize service deployment.

COMPOSER is well suited for resource constrained environments such as CPEs; among the other projects focusing on deploying network applications on CPEs we can cite the tethered Linux CPE [89], which has the limitation of being able to only run NFs implemented as eBPF programs loaded into the Linux kernel.

Before concluding this section, we analyze projects that are orthogonal to COMPOSER, as they do not cover all aspects of our proposal, have different targets and design goals, and they may interact with/be exploited by COMPOSER for some tasks.

Mantl [90] is a Cisco proposal for deploying and managing microservices through a number of consolidated technologies for cloud provisioning, service discovering, resource management, load balancing, orchestration and scheduling. Mantl supports several public cloud infrastructures (e.g., OpenStack, Google Compute Engine, Amazon Web Services), and is oriented to the deployment of applications in Docker containers.

Docker Datacenter [91] is a framework oriented to the deployment, management and monitoring of applications packaged as one or more Docker containers. One of its main components is the Docker Universal Control Plane (UCP) [92], which supports both private infrastructure and public clouds such as Amazon Web Services and Microsoft Azure. It exploits Docker Swarm as a container scheduler and infrastructure clustering, and Docker Compose to create multi-container applications (that can be deployed on multiple nodes). Unlike COMPOSER, these projects are explicitly designed for Dockerized applications; moreover, they do not focus on the architectures of the nodes running the containers.

5.6 Conclusion

The chapter presented COMPOSER, a versatile and high-performance service platform for Network Functions Virtualization that can execute several types of NFs on multiple hardware architectures and virtualization/execution environments at high speed.

COMPOSER takes advantage of a wide range of hardware and software combinations, including low-cost equipment, covering the entire spectrum from subscriber premises to carrier-grade data centers across the entire network deployment. COMPOSER has been demonstrated to run efficiently on hardware platforms such as ARM and x86, but there are no real limitations that prevent the COMPOSER software to be executed even on more specialized hardware platforms. Moreover, COMPOSER can run functions in multiple environments, ranging from “native” execution on bare metal to fully-fledged virtual machines — with the full set of virtual ports and links to fine-tune performance as needed.

Finally, from a performance point of view, COMPOSER is superior to other current NFV solutions in part because of the possibility to seamlessly employ, for instance, a high-performance underlying software switches as real-world deployment needs dictate, while at the same time retaining all the benefits of domain-oriented orchestration. This type of orchestration is required in case of services that exceed the capability of the single node, which forces the overarching orchestrator to split the service on multiple nodes with different technologies and create interconnection points such as SAPs presented in Section 5.3.1. This topic will be broadly investigated in the next chapter.

Chapter 6

An orchestration architecture supporting multiple heterogeneous domains

Part of the work described in this chapter has been previously published in [93].

6.1 Introduction

Chapter 5 introduces the concept of domain oriented orchestration that usually involves two levels of orchestration. As shown in Figure 6.1, an *overarching orchestrator* (OO) sits on top of many possible heterogeneous technological domains and receives service graph, which defines the involved VNFs and their interconnections. This component is responsible of *(i)* selecting the domain(s) involved in the service deployment (e.g., where NFs have to be executed), *(ii)* deciding the network parameters to be used to create the proper traffic steering links among the domains, and *(iii)* creating the service *subgraphs* to be actually instantiated in above domains. The bottom orchestration level includes a set of *Domain Orchestrators* (DO), each one handling a specific technological domain and interacting with the infrastructure controller (e.g., the OpenStack [69] cloud toolkit in data centers, the ONOS [94] or OpenDaylight (ODL) [95] controller in SDN networks) to actually instantiate the service subgraph in the underlying infrastructure. In addition, DOs export a summary of the computing and networking characteristics of the domain,

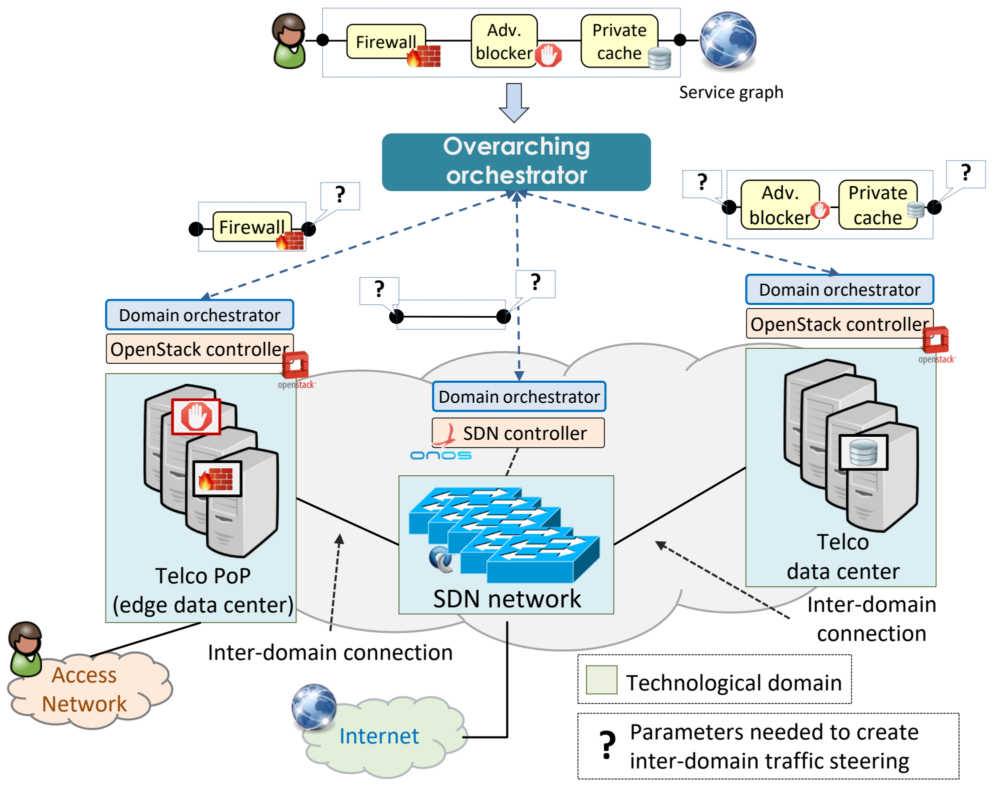


Fig. 6.1 Service graph deployment in a multi-domain environment.

used by the OO to execute its own tasks. Notably, DOs simplify the integration of existing infrastructure controllers in the orchestration framework, because any possible missing feature is implemented in the DO itself while the infrastructure controllers are kept unchanged.

Existing orchestration frameworks do not take care of automatically configuring the inter-domain traffic steering to interconnect portions of the service graph deployed on different domains. For instance, this would require to properly characterize subgraphs endpoints (called Service Access Point, or SAPs) with the proper network parameters, thus replacing the question marks in the subgraphs shown in Figure 6.1 with the proper information such as VLAN IDs, GRE keys and more, based on the capabilities of the underlying infrastructure.

The orchestration framework proposed (*i*) can transparently instantiate NFs wherever they are available (e.g., either on cloud computing or SDN domains), and (*ii*) that enables the OO to enrich the service subgraphs with information needed for DOs to automatically set up the inter-domain traffic steering.

6.2 Related work

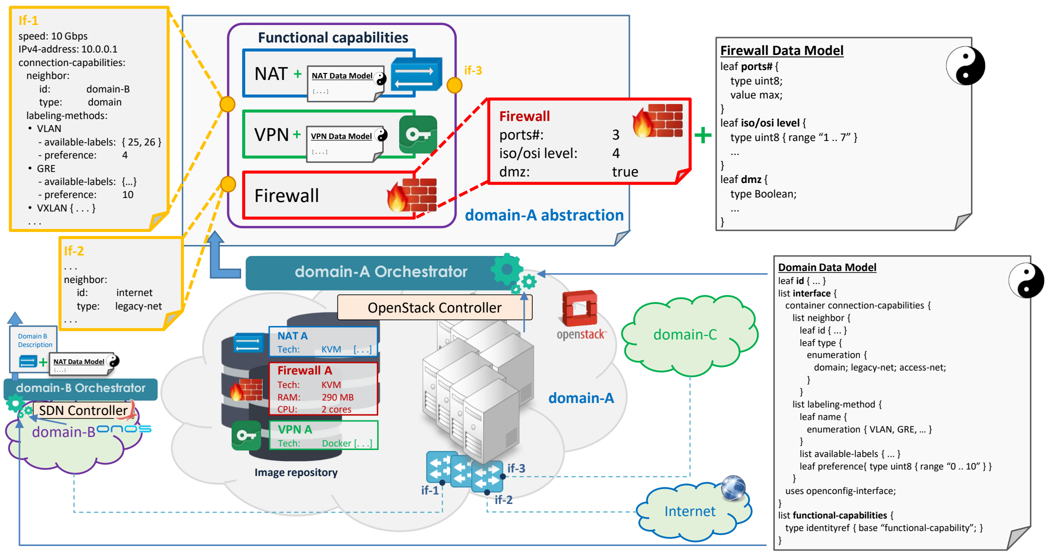
The deployment of service chains in heterogeneous domains is considered by ESCAPE [96] and FROG [60], two multi-layer orchestration architectures proposed in the context of the FP7 UNIFY project [66]. Similarly, Cloud4NFV [97] is an orchestration framework to deploy network services on different OpenStack and OpenDaylight based environments interconnected through a WAN, while the recently started 5GEx project [98] proposes an architecture to deploy services across multiple administrative domains. However, these proposals do not consider which information should be used by the upper layer orchestrator to execute its own tasks, as well as they do not detail how such an orchestrator manipulates the service chain and sets up the inter-domain traffic steering. Finally, they do not aim to exploit hardware modules and SDN controllers to execute NFs.

The NFs/links placement in multi-domain networks is studied e.g., in [99], which proposes an abstraction of the physical domains that, similarly to works like [100], is based on: *(i)* available amount of resources (e.g., CPU, memory and storage); *(ii)* inter and intra-domain link capacity. However, the paper does not consider the information needed to set up the inter-domain traffic steering, and the deployment of NFs in SDN networks and CPEs equipped with software/hardware modules that can be used to realize the requested service.

Proposals like StEERING [101] and FlowFall [102] can be instead considered orthogonal to our work, since they define traffic steering architectures that could be exploited within specific domains. Also the work carried on by the Service Function Chaining Working Group (SFC) [2] in IETF, at the best of our knowledge, mainly focuses on the data plane components. In addition, SFC defines a Network Service Header that identifies the sequence of NFs that have to process a packet, provided that the data plane components understand it.

6.3 Capability-based domain abstraction

This section presents the common data model we designed to represent heterogeneous technological domains, thus enabling the overarching orchestrator to: *(i)* execute NFs by exploiting all the processing resources available in the operator infrastructure



```
If-1
speed: 10 Gbps
IPv4-address: 10.0.0.1
connection-capabilities:
  neighbor:
     id:         domain-B
     type:       domain
  labeling-methods:
  • VLAN
     - available-labels:  { 25, 26 }
     - preference:        4
  • GRE
     - available-labels:  {…}
     - preference:        10
  • VXLAN { . . . }
  . . .
```

```
If-2
. . .
neighbor:
   id:        internet
   type:      legacy-net
. . .
```

```
Firewall Data Model
leaf ports# {
   type uint8;
   value max;
}
leaf iso/osi level {
   type uint8 { range "1 .. 7" }
   ...
}
leaf dmz {
   type Boolean;
   ...
}
```

```
Domain Data Model
leaf id { ... }
list interface {
  container connection-capabilities {
    list neighbor {
      leaf id { ... }
      leaf type {
        enumeration {
          domain; legacy-net; access-net;
        }
      }
    list labeling-method {
      leaf name {
        enumeration { VLAN, GRE, ... }
      }
      list available-labels { ... }
      leaf preference{ type uint8 { range "0 .. 10" } }
    }
  uses openconfig-interface;
}
list functional-capabilities {
  type identityref { base "functional-capability"; }
}
```

Fig. 6.2 Domain abstraction based on *functional* and *connection* capabilities.

(e.g., data centers, CPEs, SDN controllers); *(ii)* split the requested service chain in subgraphs and realize the inter-domain traffic steering.

The defined data model derives from the YANG templates defined by OpenConfig [59], which are used for traditional network services and that we extended to describe a summary of the computing and networking characteristics of the domain. Particularly, as shown in Figure 6.2, the data model includes what we call *functional capabilities*, namely the list of NFs offered by the domain and that can be exploited to create a service chain. In addition, the domain is described as a *big-switch* with a set of interfaces connecting the domain itself with the rest of the network, where each interface is associated with some *connection capabilities*.

6.3.1 Modeling functional capabilities

A *functional capability* represents the ability of the domain to execute a given NF, no matter how it is actually implemented, and it does not include any information about the resources needed for its execution. For instance, it could be a VM image in a data center, a software bundle in an SDN controller, an hardware module in a CPE, and more. Examples of functional capabilities include firewall and NAT, possibly with some specific attributes such as the number of ports, support for IPv4 or IPv6, and more. A service can choose the NFVI where to deploy the NF without knowing how it is implemented. Usually they are generic functions that implements the standard behaviour for that function additional specific capabilities have to be published on the function data model.

As shown in Figure 6.2, each functional capability is in turn associated with its own YANG-based data model, which may also indicate if a parameter represents a maximum, a minimum or a precise value (e.g., the firewall data model in figure indicates that the NF cannot have more than three ports); this information can be exploited by the overarching orchestrator to check whether the domain can be used to implement or not a given NF.

6.3.2 Modeling connection capabilities

Each interface attached to the big-switch represents a connection point between the domain and the external world (e.g., another domain, the Internet). As shown in Figure 6.2, an interface is associated with a set of *connection capabilities* such as the following.

"Neighbor" indicates what can be directly reached through that specific interface, namely: *(i)* another domain that can be exploited for traffic steering and/or to execute NFs (`if-1` in Figure 6.2); *(ii)* a *legacy network* where packets are delivered according to, e.g., the traditional IP routing, and hence paths cannot be set up by an external controller (`if-2` in the figure); *(iii)* an *access network*, which represents an entry point for the traffic into the operator network.

"Labeling method" indicates the ability of the domain to: *(i)* classify incoming traffic (i.e., packets entering in the domain through that interface) based on specific patterns (e.g., VLAN ID, GRE tunnel key); *(ii)* modify traffic that exits from the interface so that it satisfies a specific pattern (e.g., is encapsulated into a specific GRE/VXLAN tunnel, is tagged with a specific VLAN ID). Notably, other labeling methods can be supported as well in addition to those just mentioned (e.g., MPLS label, Q-in-Q, wavelength), according to the specific interface technology.

Each labeling method is associated with the list of "*labels*" (e.g., VLAN ID, GRE key) that are still available and can be exploited to tag/encapsulate new types of traffic, and with a *"preference"*. This information, an integer ranging from 0 to 10, can be used to give priority to a labeling method with respect to another and, implicitly, to express the priority of an interface with respect to the others. Our model does not specify how the preference value must be selected; for instance, it may be derived from a combination of the link capacity and of the overhead introduced by a specific technology, but other policies may be considered as well. Other parameters associated with interfaces are inherited by the OpenConfig model, such as the Ethernet and, potentially, the IPv4/IPv6 configuration. As will be shown later, all the above information can be exploited by the orchestration software to define how the traffic exiting from a subgraph in a first domain can be delivered to its next portion, running in a second domain.

Finally, additional domain information may be exported as well in addition to functional and connection capabilities, such as the available bandwidth between two domain interfaces, which may be useful to select the best domain(s) for NFs, in case multiple placements options exist, as proposed in [99].

6.4 Capability-based orchestration

This section describes how the *capability-based* domain abstraction presented in Section 6.3 is exploited by the overarching orchestrator to deploy service chains on its multi-domain infrastructure.

6.4.1 Service chain

As detailed in [60], service chains consist of a number of *Service Access Points (SAPs)*, NFs (e.g., firewall with 2 ports) and their interconnections. A SAP represents an entry/exit point for the traffic in/from the service chain; hence, as shown at the top of Figure 6.3, it may be characterized with the traffic that has to enter in the service chain through that access point, which can include traffic specifiers (e.g, IP addresses, VLANs, etc.) and physical specifiers (e.g., the entry point of such a traffic in the operator network). According to the picture, links are instead potentially associated with constraints on the traffic that has to transit on that specific connection. SAPs are very flexible identifiers and can be updated over time; for instance, in case of per-user services, a SAP should receive all the traffic coming from the user's device, e.g., all the packets matching the MAC address of the device he is currently using, and should correspond to the connection point of such a device to the network (e.g., `sap-0` in Figure 6.3). Since the user can access the Internet through different devices and

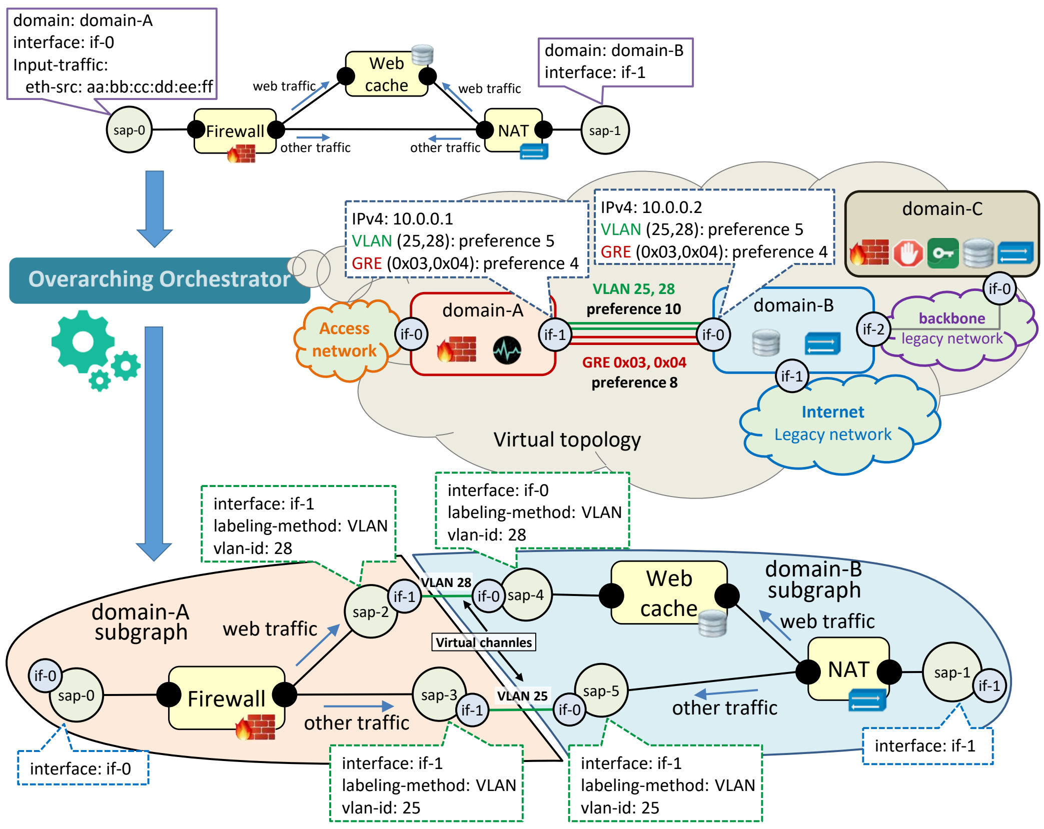


Fig. 6.3 Service chain deployment involving two domains directly connected.

from different locations, these parameters must be dynamically derived each time the service chain is going be instantiated (or updated, in case such a graph already exist). To this purpose we can exploit a *user location* service graph processing all the traffic coming from new devices, which is similar to the service detailed in [60].

6.4.2 Virtual topology

As shown in Figure 6.3, the global orchestrator models the entire network infrastructure with a set of domains characterized by a set of functional capabilities and associated with *big-switches* possibly connected. The virtual topology is created based on the connection capabilities associated with domain interfaces, as described in the following. First, the "neighbor" parameter indicates whether a connection between two interfaces (of different domains) *may* exist or not. Then, a virtual channel is actually established between two interfaces per each pair <labeling-

method,available-label> they have in common[1]; as shown in Figure 6.3, each virtual channel is then associated with such an information and with a "preference" that derives from the preference value of the labeling method in the two interfaces. For instance, a virtual channel may represent all the traffic exchanged between two interfaces and that is encapsulated into a particular GRE tunnel, or that belongs to a specific VLAN, and more. Referring to the picture, VLAN IDs 25, 28 and GRE keys 0x03, 0x04 are available in both interfaces `domain-A/if-1` and `domain-B/if-0`, then four virtual channels are established between them. As described in the remainder of this section, virtual channels play a primary role in the set up of the inter-domain traffic steering.

Figure 6.3 also shows that domains may be connected through a *legacy network*; as this network does not have any orchestrator and implements the legacy IP forwarding, virtual channels based on tunneling protocols can be established directly between interfaces connected to it. As a final remark, other information may be available as well in the virtual topology (e.g., inter-domain bandwidth, intra-domain bandwidth, path latency), in case it is advertised by domain orchestrators in addition to functional and connection capabilities.

6.4.3 Service chain placement and subgraphs generation

To deploy a service chain in the virtual topology, the best domain(s) that will actually implement the required NFs, links and SAPs must be identified. To this purpose, different algorithms may be defined/exploited.

Inspired by the hierarchical/multi-domain routing, the algorithm proposed is a greedy approach that minimizes the distance between two NFs/SAPs directly connected in the service chain, by taking into account the number of domains to be traversed to realize the connection and the following constraints. First, some SAPs are forced to be mapped to specific domain interfaces e.g., because they represent the entry point of the user traffic into the network, as mentioned in Section 6.4.1. Second, a NF must be executed in a domain that advertises the corresponding functional capability; notably, to check whether a domain is candidate or not to execute a certain NF, the description of the functional capability and the associated data model must be

[1]In case two interfaces do not have any common <labeling-method,available-label>, no link is established in the virtual topology, although they are physically connected as indicated by the "neighbor" parameter.

considered. Third, links between NFs/SAPs deployed on different domains require the exploitation of virtual channels to move packets from one domain to another; each virtual channel can be used to set up a single link of the service chain. Fourth, available virtual channels with higher preference value are used first.

However, more sophisticated algorithms may be exploited as well, according to the information exported by each domain in addition to the capabilities defined in Section 6.3, and then available in the virtual topology.

A possible placement of the service chain at the top of Figure 6.3 on the virtual topology shown in the cloud is depicted at the bottom of the picture, which has been possible because `domain-A` and `domain-B`: *(i)* offered the requested functional capabilities, and *(ii)* at least two virtual channels were available between them, one needed to set up the connection between the firewall and the NAT, the other to implement the link between the firewall and the web cache.

As shown in the picture, the output of the placement algorithm is one subgraph per each domain involved in the service chain deployment, which includes the NFs assigned to that domain and, possibly, new SAPs. These SAPs are originated by links of the service chain that have been split because connecting NFs/SAPs mapped to different domains; then, the two SAPs originated by the same link are connected through a virtual channel that terminates in two interfaces of the involved domains, in order to recreate the original link. An example is given by the link between the firewall and the NAT in Figure 6.3, which has been split causing the generation of `sap-3` in `domain-A` and `sap-5` in `domain-B`, connected through the virtual channel corresponding to traffic tagged with the VLAN ID 25. Third, links in the service chain between NFs/SAPs deployed on different domains require the exploitation of virtual links to move packets from one domain to another; each virtual link can be used to set up a single connection described in the service chain. Fourth, available virtual links with the higher preference value should be prioritized.

When applying this heuristic to deploy the service chain at the top of Figure 6.3 on the virtual topology shown in the cloud, we get the mapping depicted at the bottom of the picture, which has been possible because `domain-A` and `domain-B`: *(i)* offered the require services, and *(ii)* at least two virtual links were available between them (one is then selected to set up the connection between the firewall and the NAT, the other to implement the link between the firewall and the web cache).

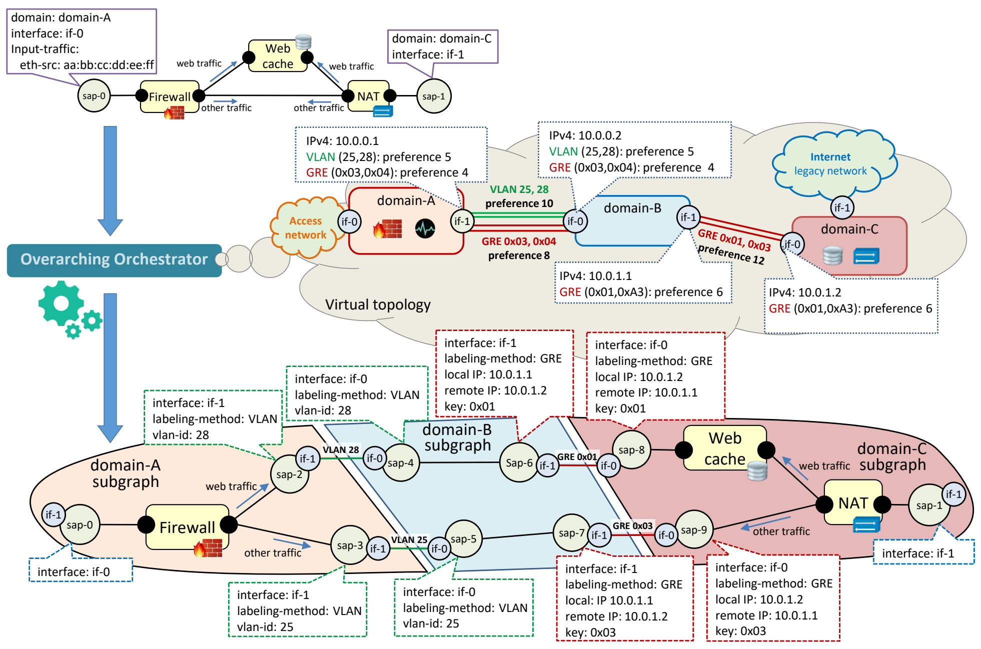


Fig. 6.4 Service chain deployment involving three domains: `domain-A` and `domain-C` execute NFs, while `domain-B` just implements network connections.

As shown in the picture, the placement algorithm originates one subgraph per each domain involved in the service chain deployment, which includes the NFs assigned to that domain and a number of new SAPs. These SAPs are originated by links of the service chain that have been split because connecting NFs/SAPs mapped to different domains; then, the two SAPs originated by the same link are connected through a virtual link that terminates in two interfaces of the involved domains (particularly, in the two interfaces directly connected), in order to recreate the original link of the service chain. An example is given by the link between the firewall and the NAT, which has been split causing the generation of one SAP in `domain-A` and one SAP in `domain-B`, connected through the virtual link corresponding to traffic tagged with the VLAN ID 25.

Figure 6.4 shows instead the case in which NFs are assigned to two domains connected by means of the intermediate `domain-B`, which is only used to forward traffic between its boundary interfaces. As shown, a subgraph is generated for this domain as well, which just includes connections between SAPs. As in the example above, SAPs of this subgraph are then connected with SAPs of other domains through virtual channels, in order to create the connection described in the service chain.

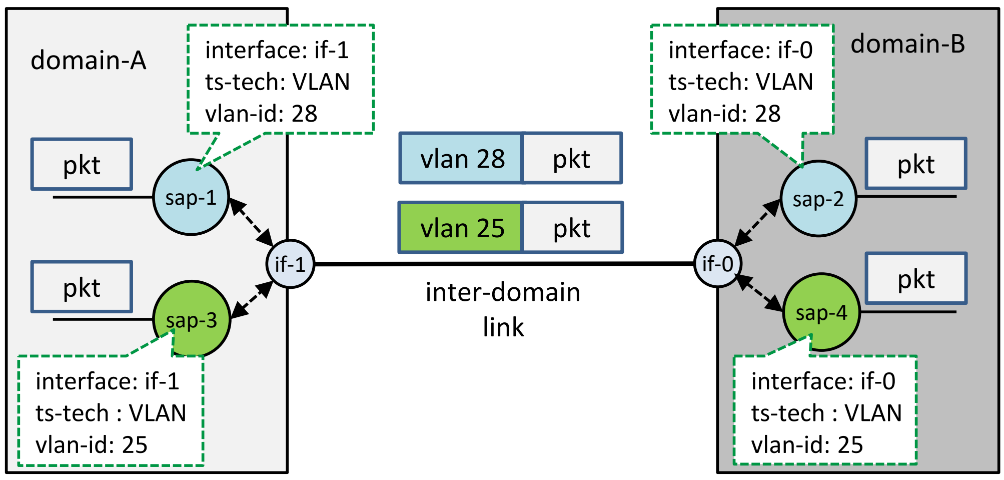


Fig. 6.5 Inter-domain traffic steering based on information associated with SAPs.

6.4.4 Inter-domain traffic steering

Both Figure 6.3 and Figure 6.4 show that, before pushing subgraphs to the proper domain orchestrators, each SAP is enriched with information associated with the virtual channel connected to the SAP itself, thus enabling those orchestrators to properly set up the inter-domain traffic steering.

As highlighted in Figure 6.5, this information allows each domain orchestrator to configure the domain so that packets sent through a specific SAP are properly tagged/encapsulated before being delivered to the next domain through a specific interface. Similarly, since a domain typically receives through the same interface traffic to be delivered to different SAPs, the information associated with each SAP allows the domain orchestrator to known that receiving traffic with a certain tag/encapsulation from a specific interface means receiving it through a certain SAP.

The figure also shows how information associated with SAPs should only be used to implement the inter-domain traffic steering; packets should be tagged/encapsulated just before being sent out of the domain, while the tag/encapsulation should be removed just after the packet classification in the next domain. The definition of the intra-domain traffic steering (i.e., how to implement links between SAPs/NFs deployed in the same domain) is instead a task of the specific domain orchestrator, and depends on the specific domain infrastructure. Moreover, in case the inter-domain traffic steering is done through a tunneling protocol (such as between `sap6` and sap8 in Figure 6.4), the specific tunnel must be set up by the involved domain orchestrators.

This requires that traffic leaving a domain is properly encapsulated/tagged so that the next domain can understand how to process it. For example, domain-B in Figure 6.3 must understand whether packets arriving through if-0 have to be provided to the web cache or the NAT. To this purpose, we attach to the same virtual link the two SAPs (in contiguous domains) that have to be directly connected in order to realize (part of) a connection of the service chain (different pairs of SAPs are instead assigned to different virtual links). In other words, as shown in the figures, we enrich the SAP description with information of the selected virtual link (e.g., GRE tunnel towards a specific remote IP and with a specific key). So that the domain orchestrator understands how specific traffic (i.e., packets sent through that SAP) must be tagged/encapsulated before being actually sent to the next domain. Similarly, the domain orchestrator knows that receiving traffic with a certain tag/encapsulation means receiving it through a certain SAP, and then it knows how to process it (i.e., to which NF it must be provided). For instance, sap-2 and sap-4 in Figure 6.3, which must be connected in order to realize the link between the firewall and the web cache, are assigned to the VLAN ID 28, while sap-3 and sap-5 are assigned to the VLAN ID 25. This way, considering a traffic flow from the left to the right of the picture, domain-A knows that: *(i)* all web traffic arriving from the firewall must be sent on sap-2, which means to tag such a traffic with the VLAN ID 28 before transmitting it on the interface if-1; *(ii)* the other traffic coming from the firewall has to be sent on sap-2, namely tagged with the VLAN ID 25 and then sent on the interface if-1. On the other side of the links, domain-B knows that traffic arriving from if-0 and tagged with the VLAN ID 28 correspond to sap-5 and then must be provided to the web cache, while traffic arriving from the same interface but belonging to the VLAN 25 have to be forwarded to the NAT. This approach can be easily applied also to the more complex example of Figure 6.4, where the two domains hosting NFs are interconnected through a third domain. This picture also highlights how the technology used to identify a specific type of traffic may change between two different domains (e.g., web traffic is associated to a VLAN in the first link, and to a GRE tunnel in the second).

It is worth noting that, in case of direct connection between two domains, the technology associated with the SAPs has just the purpose of allowing the next domain to understand how to process a given packet.

To conclude, it is worth mentioning that domain orchestrators have to take care of setting up the GRE tunnels required by the subgraph they have to implement. Moreover, the encapsulation/tag is just needed to implement the inter-domain traffic steering, while it could not be used inside a specific domain, where the task to properly set up connections as required by the subgraph is again let to the specific domain orchestrator. Then, it is up to this module to decide whether to use the same traffic steering mechanism within the domain, or to exploit some other technique; in this second case, the domain orchestrator has to properly instruct the infrastructure to remove the tag/encapsulation as soon as the packet enters in the domain. Similarly, it is the domain orchestrator that has to configure its own domain to properly tag/encapsulate the traffic, as required by the received subgraph.

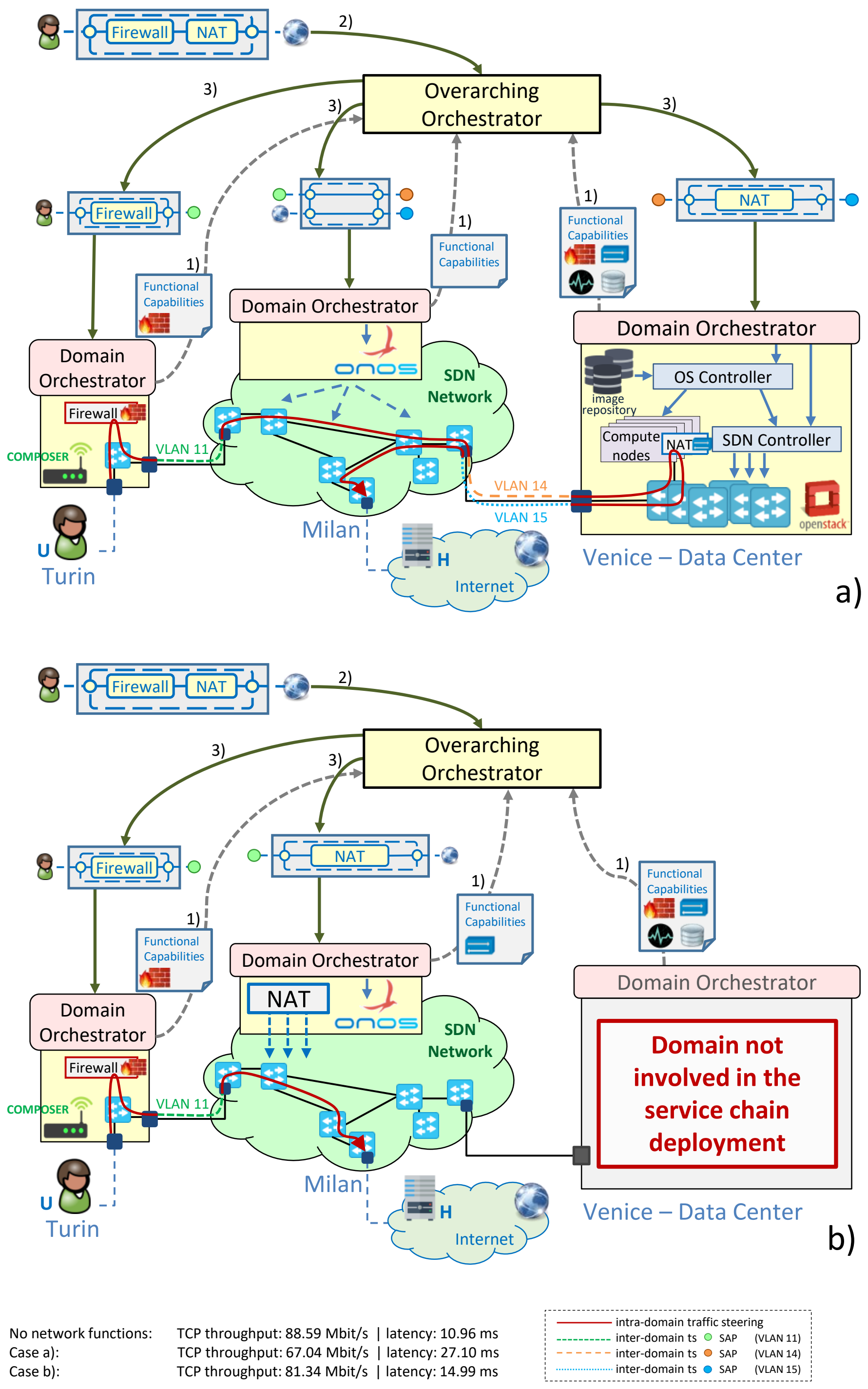
6.5 Validation

To validate both our approach to implement inter-domain traffic steering and the advantages brought by the idea of exposing, per each domain, its functional capabilities, we executed some tests over a geographical testbed consisting of an SDN domain that connects an access network encompassing a set of COMPOSERs to emulate residential gateways, an OpenStack domain and the Internet, as shown in Figure 6.6.

As shown at the top of the picture, the deployed service chain operates between the green user U and the host H on the Internet[2]. Particularly the firewall is deployed in COMPOSER, since this domain represents the entry point of the user's traffic into the network and has the firewall functional capability.

Then, in the first test (Figure 6.6(a)), no functional capability is exported by the SDN domain, which is then used just to implement network paths, while the NAT is deployed inside the data center in Venice, which is the only domain advertising such a functional capability. Notably, VLANs are used for the set up of the inter-domain traffic steering.

[2]NFs implementations: firewall: `iptables` executed in the host; NAT in the SDN domain: ONOS bundle; NAT in data center: `iptables` executed in a KVM-based VM.



No network functions:	TCP throughput: 88.59 Mbit/s	latency: 10.96 ms
Case a):	TCP throughput: 67.04 Mbit/s	latency: 27.10 ms
Case b):	TCP throughput: 81.34 Mbit/s	latency: 14.99 ms

Fig. 6.6 Validation scenario: (a) service chain split across three domains; (b) service chain deployed in two domains.

In the second test (Figure 6.6(b)), the SDN domain orchestrator exports the NAT functional capability, then the overarching orchestrator selects this domain for the execution of this NF, thus reducing the distance between chained NFs.

In both cases, we measured the end-to-end latency introduced by the service (using the `ping` command) and the throughput (using `iperf` generating TCP traffic) between U and H. As shown in Figure 6.6, performance results better when the NAT is executed in the SDN domain (case *(b)*), with respect to the case in which such a NF is deployed in the data center (case *(a)*), and it is very close to the case in which no network functions are present (*baseline*), which is limited by the speed of the geographical connections (100Mbps). This is because *(i)* traffic does not need to be forwarded to the far away data center before being actually delivered to H, and *(ii)* the NAT is actually implemented as a set of Openflow rules installed by the NAT bundle into the switches (only the first packet of a flow is processed in the software bundle, while the following packets are directly processed in hardware by the switches). Finally, the figure also reports performance measured when no NF is instantiated between U and H, in order to give an insight of how the network performs when only used to forward network packets.

6.6 Conclusion

The chapter presents an orchestration framework that can deploy service chains across the heterogeneous resources available in a multi-domain network. Particularly, it proposes a common *capability-based* representation of heterogeneous domains, where each domain is exposed to the overarching orchestrator as: *(i)* a set of *functional capabilities* indicating which NFs it is able to implement; *(ii)* a big switch whose interfaces (representing boundary interfaces of the domain) are associated with *connection capabilities* (e.g., next domain, support to GRE tunnels). The chapter shows how this information can be used by the overarching orchestrator to select domains involved in the deployment of the service chain (some domains can be used to execute NFs, while other may just be exploited to realize network connections) and to create the subgraphs to be deployed in those domains. Notably, each subgraph contains also information needed to set up the inter-domain traffic steering, namely to create links connecting NFs/service access points deployed on different domains. Moreover, functional capabilities enable the overarching orchestrator to also exploit, for the execution of NFs, software bundles available in SDN controllers as well as hardware accelerators/software components available, e.g., in CPEs. Notably, the proposed orchestration model is suitable also for the deployment of service chains across different administrative domains, since it does not mandate to export information that can be considered confidential in such a scenario (e.g., amount of available resources, links capacity).

Chapter 7

Conclusions

This thesis compiles all the scientific work carried out by the PhD candidate in his doctoral studies, which aimed at improving the performance of virtualized network services.

Network Functions Virtualization was an hot and promising technology when this thesis started, in 2014; four year later we can see that this technology has become mainstream, with several operators that already use some early products in their production environment, although the standardization has not been fully completed.

This work mainly focused on investigating the problem of the *performance* of virtualized network services, and from the perspective of a *network operator*. This angle was chosen due to the several collaborations that the candidate established with different operators during his studies and by participating to the FP7 European projects SECURED and, in some parts, UNIFY.

The thesis introduces various improvements in the context of Network Function Virtualization, both in terms of technological improvements and orchestration design. The thesis started from the analysis of the existing technologies used on the packet forwarding on a single server, and proposed an efficient algorithm to improve the packets forwarding between the switch and the network functions.

Thanks to the suggestions from telecom operators, which currently operate the intermediate network infrastructure (fronthaul/backhaul, core) but are interested to play a more active role at the edge of the network as well, even at the customer's premises, the thesis started to investigate the problem of flexible packet inspection on resource-constrained devices, such as residential or SOHO CPEs.

Driven by the observation that even resource-constrained network devices can successfully host network services, this thesis presented the architecture of a node that supports multiple virtualization technologies. In particular, this node supports the novel concept of Native Network Functions, i.e., software components launched directly on the bare metal, leveraging the capability of the home gateway to execute some already existing software. In addition, this idea allows to exploit also the hardware capabilities of the device, when available. As expected, this software architecture is suitable to be deployed also on resource constrained devices, which results in the possibility to instantiate virtual network functions close to the end user.

Finally, to overcome the limitations in terms of scalability and resource usage when operating exclusively on CPEs, the last chapter presented a more complex solutions that proposes a distributed NFV architecture, which enables to transparently deploy network functions on multiple heterogeneous technological domain, such as edge nodes, SDN networks, datacenters. This result has been achieved by defining a standard interface that enable different technological (and, possibly, administrative) domains to talk to an overarching orchestrator. The latter component is in charge of performing the deployment based on the given requirements; furthermore, when the service requirements cannot be fulfilled by a single domain, it can split the graph representing the requested service among different domains, creating the proper points of interconnection between them.

References

[1] ETSI NFV. http://www.etsi.org/technologies-clusters/technologies/nfv.

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

[3] OpenFlow. https://www.opennetworking.org/software-defined-standards/specifications/.

[4] ETSI. Network Functions Virtualisation (NFV); Infrastructure; Methodology to describe Interfaces and Abstractions, 2014.

[5] 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.

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

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

[8] Openvswitch. http://openvswitch.org/.

[9] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance characteristics of virtual switching. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 120–125, Oct 2014.

[10] Y. Zhao, L. Iannone, and M. Riguidel. Software switch performance factors in network virtualization environment. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 468–470, Oct 2014.

[11] M. Casoni, C.A. Grazia, and N. Patriciello. On the performance of linux container with netmap/vale for networks virtualization. In *Networks (ICON), 2013 19th IEEE International Conference on*, pages 1–6, Dec 2013.

[12] LXC. https://linuxcontainers.org/.

[13] L. Rizzo and G. 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.

[14] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li. Evaluating and optimizing i/o virtualization in kernel-based virtual machine (kvm). In C. Ding, Z. Shao, and R. Zheng, editors, *Network and Parallel Computing*, volume 6289 of *Lecture Notes in Computer Science*, pages 220–231. Springer Berlin Heidelberg, 2010.

[15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. volume 28, page 32.

[16] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, and C.A.F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240, Feb 2013.

[17] Xen. http://www.xen.org/.

[18] OpenVZ. https://openvz.org/.

[19] Linux-VServer. http://linux-vserver.org.

[20] DPDK. http://dpdk.org/.

[21] R. Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[22] Ntoppfring. http://www.ntop.org/products/pf_ring/.

[23] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2. USENIX Association, 1993.

[24] I. Cerrato, G. Marchetto, F. Risso, R. Sisto, M. Virgilio, and R. Bonafiglia. An efficient data exchange mechanism for chained network functions. *Journal of Parallel and Distributed Computing*, 114:1 – 15, 2018.

[25] M. Michael and M. 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.

[26] A. Gidenstam, H. Sundell, and P. 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.

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

[28] M. Hoffman, O. Shalev, and N. 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.

[29] P. Tsigas and Y. 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.

[30] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. Enabling fast, dynamic network processing with clickos. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 67–72, New York, NY, USA, 2013. ACM.

[31] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, 2014. USENIX Association.

[32] J. Hwang, K. K. Ramakrishnan, and T. 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.

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

[34] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *USENIX Annual Technical Conference*, pages 99–112, 1996.

[35] C. Dovrolis, B. Thayer, and P. Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *ACM Operating Systems Reviews*, 35:50–60, 2001.

[36] F. Fusco and L. 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.

[37] R. Bonafiglia, A. Sapio, M. Baldi, F. Risso, and P. C. Pomi. Enforcement of dynamic http policies on resource-constrained residential gateways. *Computer Networks*, 123(Supplement C):169 – 183, 2017.

[38] A. Moore and K. Papagiannaki. Toward the accurate identification of network applications. In *Passive and Active Network Measurement*, pages 41–54. Springer, 2005.

[39] W. R. Stevens. *UNIX Network Programming: Networking APIs*, volume 1. Prentice-Hall, Inc., 1997.

[40] OpenWrt: Linux distribution for embedded devices. https://openwrt.org.

[41] ApacheBench. http://httpd.apache.org/docs/2.2/programs/ab.html.

[42] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018 - TCP Selective Acknowledgment Options, 1996.

[43] Mark Allman, Vern Paxson, and Ethan Blanton. TCP Congestion Control, 2009.

[44] M. Mellia, R. Cigno, and F. Neri. Measuring ip and tcp behavior on edge nodes with tstat. *Computer Networks*, 47(1):1–21, 2005.

[45] Tinyproxy. http://tinyproxy.github.io.

[46] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

[47] K9 Web Protection. http://www1.k9webprotection.com.

[48] 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.

[49] Cisco Umbrella. https://umbrella.cisco.com/use-cases/web-filtering.

[50] Blue Coat WebFilter. https://www.bluecoat.com/products/webfilter.

[51] A. Sapio, Y. Liao, M. Baldi, G. Ranjan, F. Risso, A. Tongaonkar, R. Torres, and A. Nucci. Per-user policy enforcement on mobile apps through network functions virtualization. In *Proceedings of the 9th ACM Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '14, pages 37–42, New York, NY, USA, 2014. ACM.

[52] L. Chou, Z. He, D. C. Li, H. Chen, J. Su, C. Chen, H. Wei, and C. Li. Design and implementation of content-based filter system on embedded linux home gateway. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, pages 1046–1051. IEEE, 2012.

[53] N. Herbaut, D. Negru, G. Xilouris, and Y. Chen. Migrating to a nfv-based home gateway: introducing a surrogate vnf approach. In *Network of the Future (NOF), 2015 6th International Conference on the*, pages 1–7. IEEE, 2015.

[54] R. Cziva, S. Jouet, and D. Pezaros. Roaming edge vnfs using glasgow network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 601–602. ACM, 2016.

[55] F. Sánchez and D. Brazewell. Tethered linux cpe for ip service delivery. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–9. IEEE, 2015.

[56] L. Shuai, G. Xie, and J. Yang. Characterization of http behavior on access networks in web 2.0. In *Telecommunications, 2008. ICT 2008. International Conference on*, pages 1–6. IEEE, 2008.

[57] R. Bonafiglia, S. Miano, S. Nuccio, F. Risso, and A. Sapio. Enabling nfv services on resource-constrained cpes. In *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, pages 83–88, Oct 2016.

[58] Ivano Cerrato, Fulvio Risso, Roberto Bonafiglia, Kostas Pentikousis, Gergely Pongrácz, and Hagen Woesner. Composer: A compact open-source service platform. *Computer Networks*, 139:151 – 174, 2018.

[59] OpenConfig. http://www.openconfig.net.

[60] I. Cerrato, A. Palesandro, F. Risso, M. Suñé, V. Vercellone, and H. Woesner. Toward dynamic virtualized network services in telecom operator networks. *Computer Networks*, 92:380–395, 2015.

[61] eBPF. http://man7.org/linux/man-pages/man2/bpf.2.html.

[62] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[63] UNIFY consortium. Deliverable 4.3: Updated concept and evaluation results for SP-DevOps. http://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables/UNIFY-WP4-D4.3_final_v1.0-unify-web.pdf, 2016.

[64] Google. Google cAdvisor. https://github.com/google/cadvisor.

[65] P. Kreuger and R. Steinert. Scalable in-network rate monitoring. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 866–869, May 2015.

[66] UNIFY consortium. Unify: unifying cloud and carrier network. http://www.fp7-unify.eu/, 2013.

[67] SECURED consortium. SECURity at the network EDge. http://www.secured-fp7.eu/, 2013.

[68] A. Manzalini, F. Risso, and M. Ullio. Exploiting infrastructure capabilities to dynamically orchestrate nfv services across multiple domains. In *ETSI Workshop - From Research To Standardization*, May 2016.

[69] Openstack. https://www.openstack.org/.

[70] OpenStack community. OpenStack Tacker. https://wiki.openstack.org/wiki/Tacker.

[71] Tosca. TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0. http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html.

[72] TU Berlin Fraunhofer FOKUS. Open Baton. http://openbaton.github.io/.

[73] Telefonica. OpenMANO. https://github.com/nfvlabs/openmano.

[74] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento. Cloud4nfv: A platform for virtual network functions. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 288–293, Oct 2014.

[75] J. Soares, C. Gonçalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento. Toward a telco cloud environment for service functions. *IEEE Communications Magazine*, 53(2):98–106, Feb 2015.

[76] W. Shen, M. Yoshida, T. Kawabata, K. Minato, and W. Imajuku. vconductor: An nfv management solution for realizing end-to-end virtual network services. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–6, Sept 2014.

[77] S. Dräxler, M. Peuster, H. Karl, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris. Sonata: Service programming and orchestration for virtualized software networks. *arXiv preprint arXiv:1605.05850*, 2016.

[78] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso. Multi-domain service orchestration over networks and clouds: a unified approach. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 377–378. ACM, 2015.

[79] F. Lucrezia, G. Marchetto, F. Risso, and V. Vercellone. Introducing network-aware scheduling capabilities in openstack. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5, April 2015.

[80] V. Cunha, I. Cardoso, J. Barraca, and R. Aguiar. Policy-driven vcpe through dynamic network service function chaining. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 156–160. IEEE, 2016.

[81] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K.K. Ramakrishnan, and T. Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 26–31, New York, NY, USA, 2016. ACM.

[82] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. K. Ramakrishnan, and T. Wood. Opennetvm: Flexible, high performance nfv (demo). In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 359–360, June 2016.

[83] T. Wood, K. K. Ramakrishnan, J. Hwang, G. Liu, and W. Zhang. Toward a software-based network: integrating software defined networking and network function virtualization. *IEEE Network*, 29(3):36–41, May 2015.

[84] R. Morris, E. Kohler, J. Jannotti, and M. F. 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.

[85] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. nf.io: A file system abstraction for nfv orchestration. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 135–141, Nov 2015.

[86] R. Cziva, S. Jouet, and D. Pezaros. Gnfc: Towards network function cloudification. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 142–148. IEEE, 2015.

[87] R. Cziva, S. Jouet, K. White, and D. Pezaros. Container-based network function virtualization for software-defined networks. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 415–420. IEEE, 2015.

[88] R. Cziva, S. Jouet, and D. Pezaros. Roaming edge vnfs using glasgow network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 601–602. ACM, 2016.

[89] F. Sánchez and D. Brazewell. Tethered linux cpe for ip service delivery. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–9, April 2015.

[90] Cisco. MANTL. http://mantl.io.

[91] Docker. Docker for the enterprise. https://www.docker.com/enterprise.

[92] Docker. Docker Universal Control Plane. https://www.docker.com/sites/default/files/DCR_UniversalControlPlane_TechBrief_070716_v1.pdf.

[93] R. Bonafiglia, G. Castellano, I. Cerrato, and F. Risso. End-to-end service orchestration across sdn and cloud computing domains. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, July 2017.

[94] ONLAB. Open Network Operating System. http://onosproject.org/.

[95] Open Daylight. https://www.opendaylight.org/.

[96] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso. Multi-domain service orchestration over networks and clouds: a unified approach. *ACM SIGCOMM Computer Communication Review*, 45(4):377–378, 2015.

[97] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento. Cloud4nfv: A platform for virtual network functions. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 288–293, Oct 2014.

[98] 5g exchange, 2015. Accessed on: 2017-02-06.

[99] I. Vaishnavi, R. Guerzoni, and R. Trivisonno. Recursive, hierarchical embedding of virtual infrastructure in multi-domain substrates. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–9. IEEE, 2015.

[100] T. Soenen, S. Sahhaf, W. Tavernier, P. Sköldström, D. Colle, and M. Pickavet. A model to select the right infrastructure abstraction for service function chaining. In *IEEE NFV-SDN2016, the IEEE Conference on Network Function Virtualization and Software Defined Networks*, pages 1–7, 2016.

[101] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, et al. Steering: A software-defined networking for inline service chaining. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2013.

[102] R. Nakamura, K. Okada, S. Saito, H. Tanahashi, and Y. Sekiya. Flowfall: A service chaining architecture with commodity technologies. In *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, pages 425–431. IEEE, 2015.

Appendix A

Publications list

This appendix lists all the paper that have been published during the PhD studies.

A.1 Journals

- Bonafiglia R., Sapio A., Baldi M., Risso, F. and Pomi P. C. (2017). *Enforcement of dynamic HTTP policies on resource-constrained residential gateways*. Computer Networks, 123, 169-183.
- Cerrato I., Marchetto G., Risso F., Sisto R., Virgilio M. and Bonafigli, R. (2017). *An efficient data exchange mechanism for chained network functions*. Journal of Parallel and Distributed Computing, 114, 1-15.
- Cerrato I., Risso F., Bonafiglia R., Pentikousis K., Pongracz G., Woesner H., (2018). *COMPOSER: A Compact Open-source Service Platform*. Computer Networks, 139, 151–174.

A.2 Conferences

- Bonafiglia R., Ciaccia F., Lioy A., Nemirovsky M., Risso F. and Su T., *Offloading personal security applications to a secure and trusted network node*, Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft), London, 2015, pp. 1-2.

- Bonafiglia R., Cerrato I., Ciaccia F., Nemirovsky M. and F. Risso, *Assessing the Performance of Virtualization Technologies for NFV: A Preliminary Benchmarking*, 2015 Fourth European Workshop on Software Defined Networks, Bilbao, 2015, pp. 67-72.

- Bonafiglia R., Miano S., Nuccio S., Risso F. and Sapio A., *Enabling NFV Services on Resource-Constrained CPEs*, 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), Pisa, 2016, pp. 83-88.

- Baldi M., Bonafiglia R., Risso F. and Sapio, A., *Modeling Native Software Components as Virtual Network Functions*, In Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference (pp. 605-606). ACM.

- Bonafiglia R., Castellano G., Cerrato I. and Risso F., *End-to-end service orchestration across SDN and cloud computing domains*, 2017 IEEE Conference on Network Softwarization (NetSoft), Bologna, 2017, pp. 1-6.