COMPOSER: A Compact Open-source Service Platform

Ivano Cerrato[a,*], Fulvio Risso[a], Roberto Bonafiglia[a], Kostas Pentikousis[b], Gergely Pongrácz[c], Hagen Woesner[d]

[a] *Department of Control and Computer Engineering, Politecnico di Torino, Italy*
[b] *Travelping GmbH, Berlin, Germany*
[c] *TrafficLab, Ericsson Research, Budapest, Hungary*
[d] *Berlin Institute for Software Defined Networks GmbH, Berlin, Germany*

Abstract

Compute and network virtualization enable to deliver network services with unprecedented agility and flexibility based on a) the programmatic placement of service functions across the available infrastructure and b) the real-time setup of the corresponding network paths. This paper presents and validates COMPOSER, a compact, flexible and high-performance service platform for the deployment of network services. COMPOSER supports multiple virtualization engines (e.g., virtual machines, containers, native network functions) and it can use seamlessly the above different execution environments to instantiate network services belonging to different chains, hence facilitating domain-oriented orchestration and enabling the joint optimization of compute and network resources. We demonstrate that COMPOSER can run on resource-constrained hardware, such as residential gateways, as well as on high-performance servers. Finally, COMPOSER integrates optimized data plane components that enable our platform to reach top-class results with respect to data plane performance as well.

Keywords: Service orchestration, Service virtualization, Compute node, High performance, Resource-constrained device, NFV, SDN, Service chain

1. Introduction

Compute and network virtualization enable the instantiation of Service Functions (SF) across the (possibly heterogeneous) 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.

In order to enable efficient service deployment and delivery on such resources, we designed COMPOSER (COMPact Open-source SERvice platform), that offers a high-level abstraction for composing service functions in arbitrary service graphs used to deliver virtualized services. We design and implement 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 IT virtualization on resource-constrained hardware such as home/SOHO CPEs, also known as residential gateways, thus enabling

[*] Corresponding author. Email address: ivano.cerrato@polito.it.

telecom operators to use the same service deployment model across the entire (heterogeneous) infrastructure available from the end-user sites through the fronthaul/backhaul/core network till to the remote datacenters.

COMPOSER, whose source code is available at [1], can be executed on different hardware platforms, ranging from high-volume servers equipped with Intel x86 CPUs and several GBs of memory and disk, to embedded devices equipped with ARM or MIPS processors, limited hardware resources, and possibly a custom version of the Linux operating system. COMPOSER exploits locally available information to optimize service deployment. For instance, COMPOSER evaluates local resources/constraints to select the best implementation of a required service and binds SF(s) to the most appropriate CPU core. Finally, COMPOSER is able to execute SF(s) running in different execution environments, according to the different operating contexts. For instance, a resource-constrained CPE can execute SF(s) on bare metal, while full-fledged virtual machines are more appropriate for "fat" servers.

Figure 1 is a high-level view of the overall operation. First, COMPOSER receives a service graph from an overarching (hierarchy of) orchestrator(s) and subsequently takes care of all operations required to make the service fully functional on edge and mid-path resources, e.g., residential gateways and server(s) at customer premises or at the Points of Presence (POPs) of the operator, while data center servers are typically managed by cloud computing software platforms such as OpenStack.

Note that this paper focuses on COMPOSER and does not delve into the details of the overarching orchestrator, which is orthogonal and complementary to this work. Interested readers can refer to earlier literature on FROG [2] and ESCAPE [3], which have been demonstrated to work seamlessly with COMPOSER at different venues, including IETF 96.

The remainder of the paper is structured as follows. Section 2 briefly recaps the terminology used in this paper, while Section 3 analyzes related work and compares it with COMPOSER. Section 4 highlights the design objectives of our platform, Section 5 presents the overall software architecture of COMPOSER, while Section 6 focuses on the data plane. Section 7 discusses how COMPOSER can be used to implement pure SDN/OpenFlow services, as well as its relationship with the ETSI NFV architecture, showing how COMPOSER can also provide Network Functions Virtualization (NFV) services according to the ETSI model. An extensive evaluation of the proposed platform is then provided in Section 8, while Section 9 concludes the paper.

2. Terminology

This section recaps the concepts and terms used in the remainder of the paper.

A **service function (SF)**, or simply **function**, is a functional block or application that either operates on (e.g., reads, manipulates) traffic in transit (in this case, it corresponds to a Virtual Network Function (VNF) defined by ETSI [4]), or acts as the origin/final destination for some types of traffic. Without loss of generality, and essentially for illustration purposes, we will often refer to "firewall" and "NAT" as examples of the first class of functions. The second class includes, for example, applications related to the "traditional" cloud computing world, such as a private storage service.

A **service graph** is a set of functions suitably interconnected to implement a specific service, such as, for example a comprehensive security solution. As illustrated in Figure 2, 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*

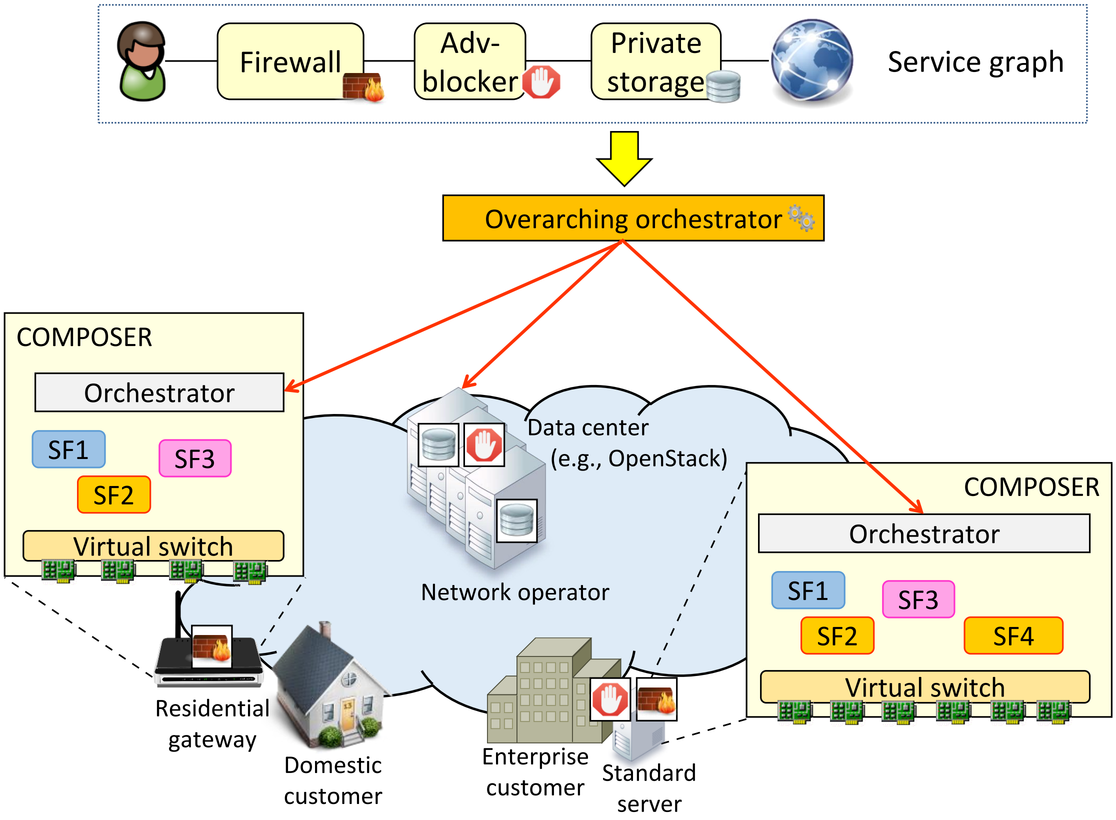


Figure 1: COMPOSER network operation.

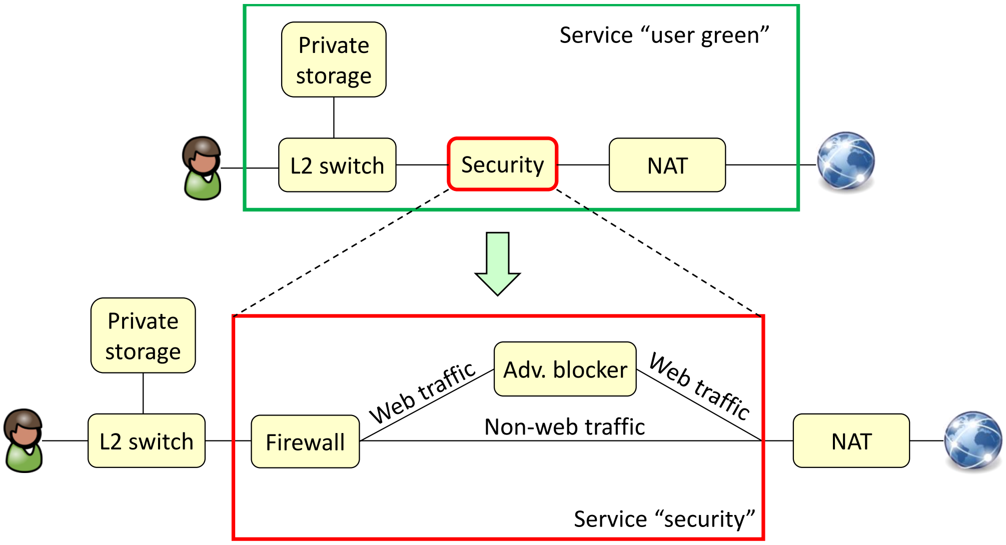


Figure 2: Service graph, network functions, and recursive functional decomposition.

blocks concept advanced by ETSI [5], each function may, in turn, be defined as a service graph. As an example, SF `security` in Figure 2 is in fact a service graph comprising a `firewall` and an advertisement blocker (`adv-blocker`).

An **infrastructure node** (or simply *node*) is a physical machine that can execute SFs in one or more execution environments and can implement the network paths among them as described in the service graph. As stated in the introduction, we target a deployment environment that is heterogeneous in terms of node capabilities, ranging from high-volume standard servers to resource-constrained devices.

A **capability** is an information that indicates what an entity “*can do*” and how it can be exploited. For example, an overarching orchestrator can use capabilities to select the best node on which (part of) the service graph has to be instantiated. We distinguish between two types of capabilities: *functional* and *infrastructure*. The former indicates the SFs that can be executed in a given infrastructure node, which must be able to launch them by simply receiving the name of the function that has to be started. Examples of functional capabilities include `firewall` and `NAT`, possibly with some specific attributes such as *3 ports, Gigabit Ethernet NICs, support for IPv4 or IPv6*, and so on. A *functional capability* does not specify *how* a function is implemented. That means that COMPOSER can implement a service graph using any arbitrary combination of virtual machines (VMs), containers, or processes running natively on the node and taking advantage of any available hardware acceleration facilities (e.g., for traffic encryption/decryption). The selection of the most suitable implementation for the task at hand is left to the infrastructure node. An *infrastructure capability* is instead a low-level characteristic of the infrastructure node, such as its CPU architecture, the possibility to execute KVM-based VMs or Docker containers, and so on.

A **resource** is an information that relates to what an infrastructure node “*can offer*”. For example, resource represents the available amount of a hardware component of the infrastructure node, such as memory (e.g., *4 GB*), CPU (e.g., *CPU load: 66%*), possible hardware accelerators, and more. This information can be exploited by the overarching orchestrator while selecting the best infrastructure node for SF deployment.

3. Related work

The scientific literature includes several software-based architectures of network nodes that, similarly to COMPOSER, can execute service functions exploiting the advantages of compute and network virtualization.

NetVM [6] is a platform designed to efficiently transfer packets between SFs 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 [6].

OpenNetVM [7, 8] and SDNFV [9] are platforms built on top of NetVM, which execute ad-hoc DPDK-based SFs within Docker containers and provide a high-level abstraction to compose SFs in service chains, control packet flows, and manage SF resources. Particularly, the SFs that have to process a packet can be selected both by an SDN controller (not part of the OpenNetVM framework) and by SFs, which can program the vSwitch forwarding table without the necessity to interact with said controller. Notably, SDNFV [9] also focuses on creating paths among SFs deployed on multiple hosts.

ClickOS [10] proposes a platform for high-performance NFV services. Particularly, it uses the VALE vSwitch [11] to provide packets to ClickOS virtual machines, i.e., Xen-based [12] VMs executing a Click [13] program running on top of a minimal operating system. Unlike COMPOSER, which supports many execution environments and focuses on control and orchestration aspects (although it integrates many technologies oriented to optimize data path as well), 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 [14] 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 VNFs 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 [15] and GLANF [16, 17] are frameworks to deploy VNFs. In addition to a manager that allocates VNFs to servers, and a network controller that configures traffic steering rules both in the server itself and among different servers, these frameworks define a per-server agent 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.

OpenStack [18] is a widespread cloud platform used for orchestrating all the resources available in the data center, namely compute, network and storage. Then, unlike COMPOSER, which manages the resources of a single node, OpenStack is able to deploy SFs across multiple compute nodes.

Being oriented to large-scale infrastructures, OpenStack presents several limitations when focusing on a single node, even more when resource-constrained devices are involved. Furthermore, it does not take properly into account some peculiar aspects of some service functions, such as the I/O bound ones. For instance, it does not have the concept of network service, which means that each SF 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 SFs in the service to be deployed, which may result in poor performance for the whole service, particularly when I/O bound SFs are involved. An attempt to introduce the concept of service in OpenStack is described in [19], which also highlights how the so-called *network-aware scheduling* is hard to be implemented in such an environment. Moreover, as shown later in Section 8.1, OpenStack is not suitable for resource-constrained environments such as CPEs. Instead, OpenStack can be used to implement a virtual CPE such as in [20], in which the traditional CPE functions are implemented as VNFs running in an OpenStack-based data center.

As detailed later in Section 7.2, COMPOSER can provide NFV services according to the architecture proposed by the European Telecommunications Standards Institute (ETSI) [21], as it includes orchestration and Virtual Infrastructure Manager (VIM) functionalities defined by such a proposal. Based on the ETSI architecture, both industry and academia introduced several prototypes and proofs of concept (PoCs) to deploy network services and functions. However, most of said earlier works define the architecture of the orchestrator (NFVO in the ETSI terminology [4]) that sits on top of many infrastructure nodes and deploys network services through OpenStack (which then acts as a VIM). An example of such proposals is OpenStack Tacker [22], which implements the NFVO and a generic Virtual Network Functions Manager (VNFM), and uses the *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [23] as a formalism to describe the various aspects of the service to be deployed, which is an implementation of the ETSI descriptors. Open Baton [24] defines a NFVO and a generic VNFM and can be installed on top of existing cloud infrastructures like OpenStack. Open Source MANO (OSM) [25], whose fourth version has been released in October 2017, provides an open source Management and Orchestration (MANO) framework aligned with ETSI NFV Information Models. Among the other features, it is compatible with several VIMs (i.e., OpenStack, OpenVIM, Vmware and Amazon Web Services) and SDN controllers (OpenDaylight, ONOS, Floodlight), and supports the integration of new ones thanks to its plugin model. All 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 COMPOSER instead of the other supported environments.

Similar considerations are valid for Cloud4NFV [26, 27], a platform for managing network services in a cloud environment; it exploits both a data center controller (OpenStack) to deploy VNFs and a WAN controller (OpenDaylight) to interconnect parts of the service deployed in different data centers. OpenStack and OpenDaylight are used also by vConductor [28], while SONATA [29] can support different infrastructures by means of adapters; moreover, SONATA supports recursion at the orchestrator layer. Recursion was considered also in the UNIFY project [30], where two orchestrators [31], [2] that can sit on top of different infrastructures have been defined, and in the 5GEx project [32], which considers orchestration of network services across multiple administrative domains.

Table 1 summarizes the above-mentioned literature and compares it against the COMPOSER design objectives (detailed later in Section 4). Note that platforms based on OpenStack are not included in the table, as they inherit the characteristics summarized in the corresponding OpenStack row of the table. Moreover, it is worth to point out that, according to the table, the uniqueness of

Table 1: COMPOSER vs. service platforms and frameworks.

	Domain-oriented Orchestration	**Multiple vSwitch Support**	**Execution Engine Support**	**Joint optimization net/compute**	**Performance**	**CPE Support**
COMPOSER	YES	OvS (+DPDK) xDPd ERFS	KVM, Docker, NSFs, DPDK proc.	Possible	YES: DPDK, SF direct connection, ERFS	YES: small footprint, NSFs
OpenStack	NO	Possible	YES	Hard to implement [19]	Possible	NO: each machine is a Nova node
NetVM	Info not available	NO: ad-hoc DPDK-based switch	NO: ad-hoc DPDK process in VM	Info not available	YES: DPDK-based zero-copy	NO: VMs require significant resources
OpenNetVM	NO: info not exported	NO: based on NetVM	NO: ad-hoc DPDK process in Docker)	Info not available	YES: DPDK-based	Info not available
ClickOS	Orchestration out of scope	NO: based on VALE	NO: Xen-based VMs running Click)	NO: no concept of network service	YES: efficient data transfer mechanism between Xen and the guest	Info not available
nf.io	Info not available	OvS, iptables	VM, Docker, LXC, processes	Info not available	Info not available	Info not available
GNFC **GLANF**	Info not available	Possible: currently OvS with veth ports	Docker, Linux containers	NO	Info not available	YES: small footprint, container-based

COMPOSER is not in the support of each single feature, but in the fact that, at the best of our knowledge, it is the only platform that supports all those features at the same time.

In addition to services including VNFs and cloud oriented applications, COMPOSER supports SDN services. For instance, it can be used to provide pure OpenFlow services as detailed later in Section 7.1, as well as it can be used as a platform to run the home gateway proposed in [33], which controls the wireless devices available in an home network.

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 [34], which has the limitation of being able to only run SFs 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.

Docker Datacenter [35] 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) [36], 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. Similar considerations are valid for Kubernetes (K8s) [37] as well, a framework for automating the deployment and management of containerized applications, which orchestrates computing, networking, and storage infrastructure.

Finally, the architecture presented in this paper seamlessly integrates together some technologies that were already presented in our previous works (the NSF concept [38], the SF-to-SF optimization [39], the ERFS [40] and the Elastic Router service [41]), and makes the prototype usable in the real world.

4. Design objectives

This section summarizes the COMPOSER design objectives.

4.1. 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 SF is actually implemented. This enables the upper orchestration layers to request a SF 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 SF and use them transparently from the upper orchestration layer; and *(ii)* flexibly select the best implementation for a given request and specific SF, according to the current state of COMPOSER per se, i.e., number of CPU cores currently available or availability of a SF 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.

4.2. 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.

4.3. Compute Abstraction

The COMPOSER control plane is able to interact with different execution engines and therefore to execute SFs 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 SFs (e.g., VMs use more memory than containers), and hence are well suited for particular COMPOSER deployments.

4.4. Joint Network and Compute Service Graph Optimization

The service to be implemented by COMPOSER is specified according to the Service Functions-Forwarding Graph (SF-FG) formalism detailed in Section 5.1.1, 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 SFs and CPU cores by considering the way SFs are interconnected with each other in the service graph.

4.5. High-performance Data Plane

When traffic steering is implemented through Open vSwitch (OvS) and SFs are DPDK [42] processes executed in VMs, COMPOSER is able to optimize packet transfer between SFs by bypassing the vSwitch when the service specifies point-to-point connections among them. In addition, COMPOSER can control the Ericsson Research Flow Switch [40], which implements a high-speed OpenFlow pipeline compliant with OpenFlow 1.3.

4.6. Small Footprint

This enables to deploy services (or part of them) also on resource-limited hardware already available at the edge of the network, e.g., on residential gateways, that cannot be controlled by existing orchestration platforms such as OpenStack.

4.7. Support for Native Service Functions

Existing virtualization engines are quite demanding in terms of resources required to run SFs (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 SFs 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 SFs as well.

Native Service Functions (NSF) represent a way to exploit these native (software and hardware) modules, delivering efficient SFs implementations *and* reduced overhead. As a consequence, COMPOSER, due to its small footprint and seamless support for NSFs, enables the operator to deploy the service graph on existing residential gateways, which are then integrated in the overall virtualization telco infrastructure. This is a design feature that in a real-world deployment enables an overarching orchestrator to optimize SF placement and scheduling. As an illustrative example, SFs 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.

5. COMPOSER architecture and control plane components

This Section presents the main components of the COMPOSER architecture, whose high-level view is depicted in Figure 3.

The COMPOSER orchestrator (*c-orch* in the remainder of the paper) is undoubtedly 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

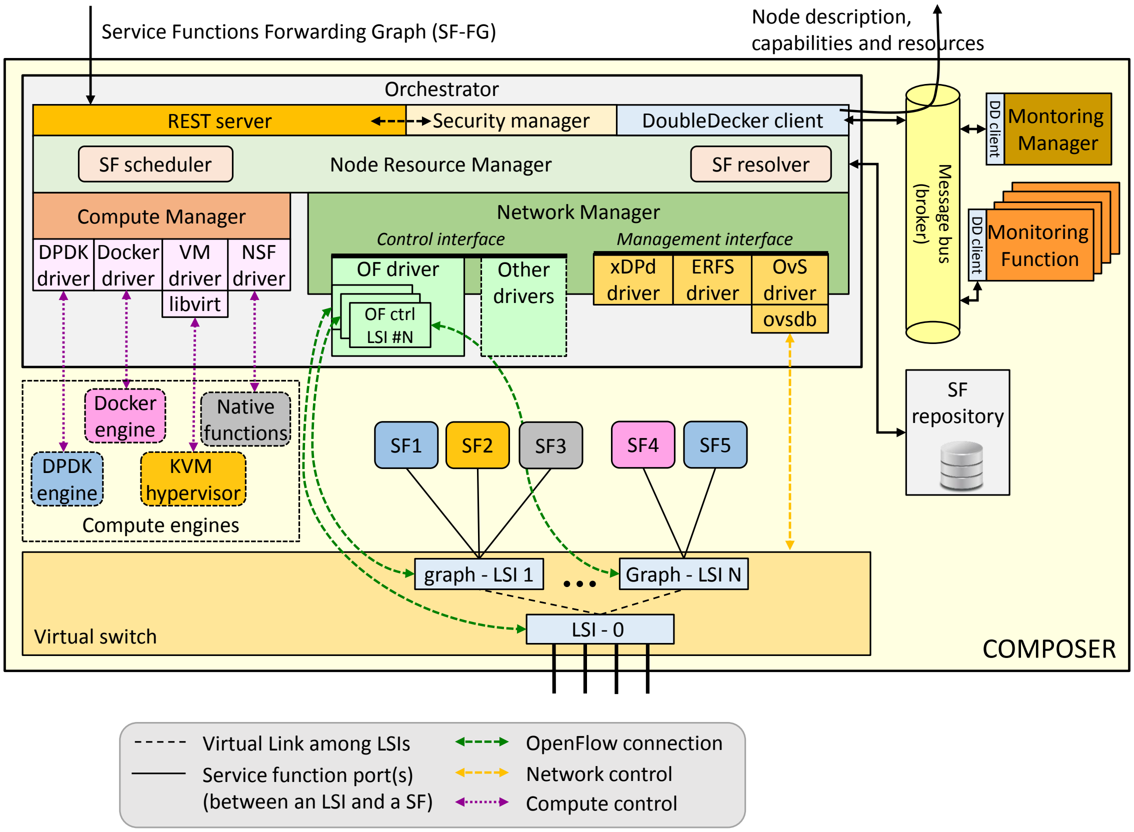


Figure 3: COMPOSER architecture.

abstraction, hence it orchestrates the above resources within COMPOSER by (*i*) handling the complete lifecycle of the virtual execution environment(s) and (*ii*) configuring the proper network 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 *SF repository* to select the best SF implementation available that matches the service request. The SF 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 SFs and a number of *compute engines* that can execute SFs implemented with different technologies (e.g., virtual machines, containers, or natively).

The remainder of this section details the modules of c-orch from both an architectural and a functional point of view, 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 SFs. Data plane features of COMPOSER will instead be considered in Section 6.

5.1. Northbound Interface

c-orch interacts with the overarching orchestrator(s) through a bidirectional northbound interface. Specifically, c-orch receives a service graph described according to the SF-FG formalism [2], and exports information using an OpenConfig-derived [43] YANG model describing the COMPOSER domain.

As shown in Figure 3, Create, Read, Update and Destroy (CRUD) commands related to the SF-FG are received through a REST API, while the COMPOSER description is exported through DoubleDecker (DD) [44], a hierarchical messaging bus based on ØMQ [45] and that, among other things, supports a publish/subscribe model. The rationale behind this choice is the following. 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, thanks to the message bus, 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 through DD using a specific *topic*, enabling all entities interested in such information to subscribe to it. 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 3, 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 SF-FGs, while end users may have limited access to the current portion of the service graph that handles their own Internet connection, and in read-only mode.

5.1.1. Service Functions - Forwarding Graph

The *Service Functions - Forwarding Graph (SF-FG)* formalism [2] describes the service to be instantiated with respect to compute (i.e., functions composing the service) and network (i.e., traffic steering rules) primitives. Furthermore, it may contain some annotations expressed using the MEASURE language (described in [46]) that specify which parameters of the SF-FG should be monitored.

As shown in Figure 4, each SF-FG consists of three main parts. First, the `SFs` section lists the functions that compose the service. Particularly, the SF-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 SF repository. However, the SF-FG can also ask for a particular 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. In general, we have considered generic L1, L2 and L3 SAPs, although the current prototype implements only four specific SAPs, namely *interface*, *vlan*, *GRE* and *host stack*, as typical representatives. In this case, while the *interface* SAPs (Figure 4) 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 5a), and that all traffic sent on such a SAP is tagged (by COMPOSER) with the proper VLAN ID.

```
{
 "sffg":
 {
  "id": "0x1",
  "name": "example graph",
  "SFs": [
  {
   "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": "service-access-point:0x1",
    ....
   },
   "actions":
   {
    "output_to_port": "sf:0x1:0xa",
    ....
   }
  },
  ....
  ]
 }
}
```

Figure 4: Excerpt of Service Functions - Forwarding Graph (SF-FG).

```
{
 ”id”: ”0x2”,
 ”type”: ”vlan”,
 ”vlan”:
 {
  ”vlan−id”: ”25”,
  ”if−name”: ”eth1”
 }
}
```

(a)

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

(b)

Figure 5: Examples of SAPs in the SF-FG: *(a) vlan* SAP; *(b) GRE* SAP.

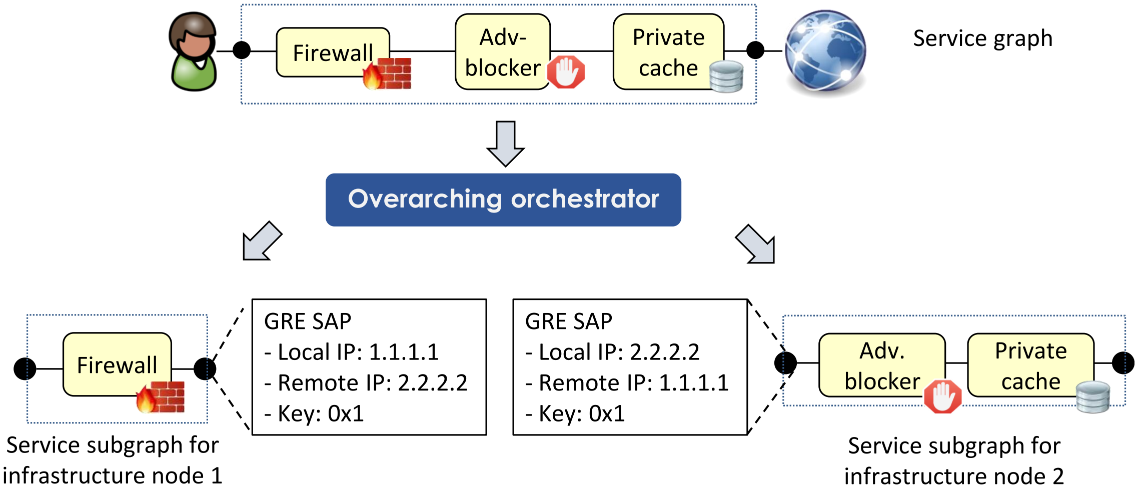


Figure 6: Using *GRE* SAPs to set up traffic steering between subgraphs.

The *GRE* SAP (an example is shown in Figure 5b) 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* and *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, or even in different domains. Figure 6 provides an example in which 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 SFs to operate irrespective of the nature of the network connection, because the actual delivery of the traffic to the next SF in the chain is done transparently by the infrastructure node, e.g., by encapsulating packets in a given VLAN or in a tunnel toward a remote destination.

The *host stack* SAP represents the connection point between the service graph and the TCP/IP stack of the device where COMPOSER is executed. This allows any TCP/IP service executed in the host (e.g., the COMPOSER REST server) to connect to the external world through a service graph, resulting in the possibility to control their connections to the network (e.g., the COMPOSER REST server may be "protected" through a firewall instantiated in a service graph).

Finally, the `flowrules` section in Figure 4 describes the interconnections between SF 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/SF port and potentially on protocol fields (e.g., IP source); *(iii)* an action that forwards packets through a specific SAP/SF port and that potentially modifies the packet content (e.g., decrease the IPv4 TTL).

5.1.2. Domain Description

The COMPOSER domain description published by c-orch via DD 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.

5.2. Traffic Steering Model

As shown in Figure 3, c-orch implements the network paths between SF 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 SFs 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 SF-FG has to be deployed) implement the traffic steering paths among the SFs, *host stack* and GRE SAPs that belong to that graph. In fact, while physical interfaces are connected to `LSI-0`, SF ports, *host stack* and GRE SAPs are connected to the associated graph-LSI.

Notably, as each graph-LSI is connected to SFs, *host stack* and GRE SAPs of a different SF-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, SFs 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 SF-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 SF-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 SFs networking, hence it can implement network connections as defined in the SF-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 usually attach VMs/containers always to a L2 bridge.

5.3. 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 7, when c-orch receives a command to create a new SF-FG, the node resource manager: *(i)* interacts with the SF repository in order to select the most appropriate SF image for each function that is part of the service and that is not explicitly associated with an image in the SF-FG; *(ii)* configures the vSwitch to create a new LSI and the ports required to connect it to the SFs to be deployed; *(iii)* deploys and starts the selected SFs; 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 3 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 SFs. The *SF resolver* interacts with the SF 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 SF (e.g., 3 ports). Finally, the *SF scheduler* can optimize the SF/CPU core(s) binding(s) by taking into consideration information such as how a SF interacts with the rest of the SF-FG.

5.4. 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 SF-FG; such an architecture supports the parallel instantiation of multiple service graphs according to the traffic steering model presented in Section 5.2.

Setting up the paths described in the SF-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 attached to the SFs[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 2, 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 SF, *(iii)* create/destroy virtual links between two LSIs, and *(iv)* create/destroy SAPs.

The support for multiple switching technologies enables the selection of the best vSwitch for a given scenario. Particularly, our prototype supports (one at a time) the widely-used Open vSwitch (OvS) [47], the extensible Data-Path daemon (xDPd) [48] and the Ericsson Research Flow Switch (ERFS) [40]. In fact, while xDPd supports offloading OpenFlow rules to the underlying hardware and then is well suited in case COMPOSER is deployed on a box with hardware switching capabilities, ERFS is very fast but requires to be executed on high-end servers, due to its high demand it terms of CPU cores. Further vSwitches can be supported by writing the corresponding driver implementing the primitives of Table 2.

Similarly, the control interface of the network manager can potentially enable the configuration of the forwarding table(s) of the LSIs by means of different technologies (e.g, OpenFlow [49], eBPF [50], P4 [51]), 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 3 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 3, a different technology-specific controller is created for each LSI, which

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

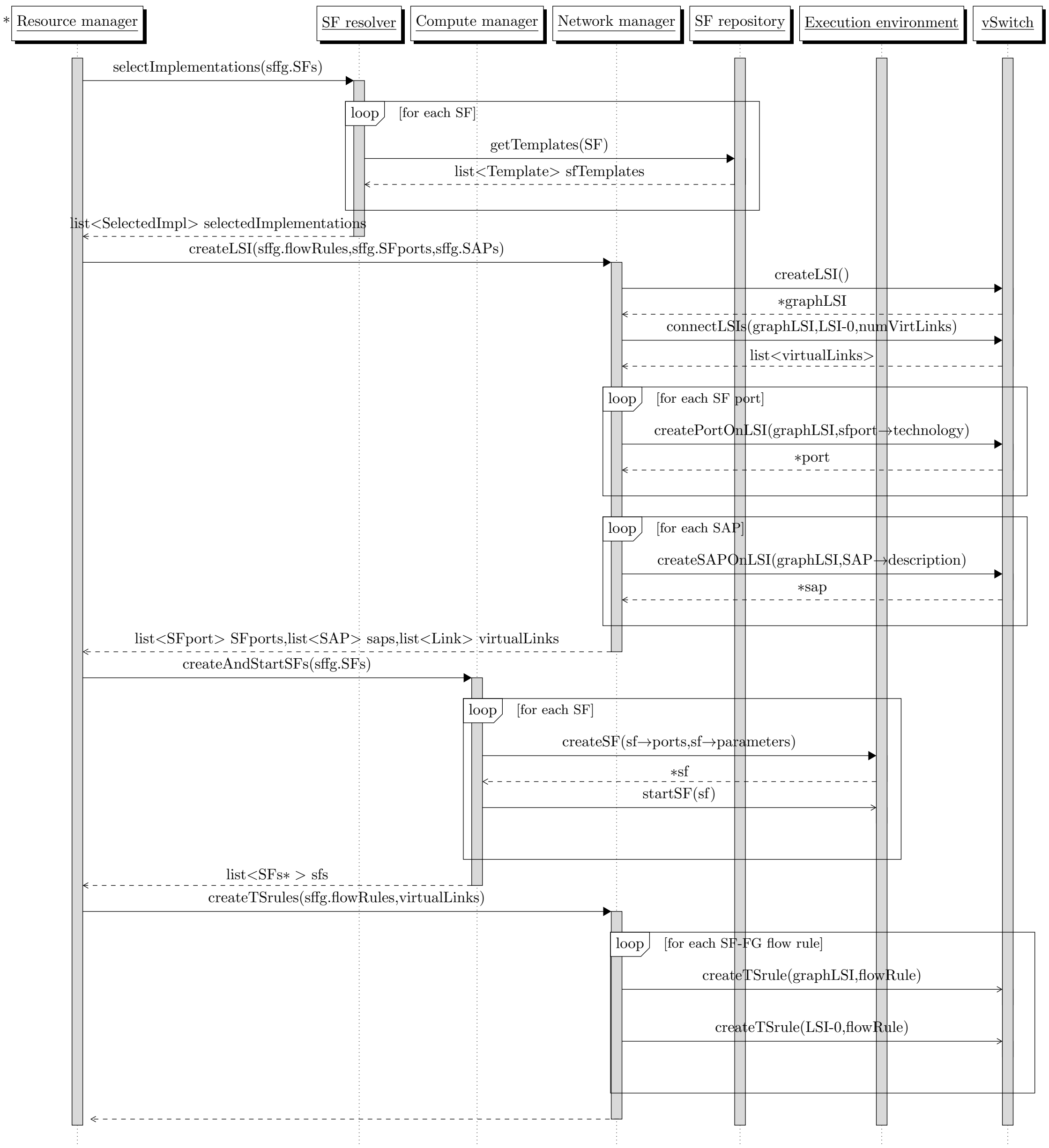

* Resource manager
SF resolver
Compute manager
Network manager
SF repository
Execution environment
vSwitch
selectImplementations(sffg.SFs)
loop
[for each SF]
getTemplates(SF)
list<Template> sfTemplates
list<SelectedImpl> selectedImplementations
createLSI(sffg.flowRules,sffg.SFports,sffg.SAPs)
createLSI()
∗graphLSI
connectLSIs(graphLSI,LSI-0,numVirtLinks)
list<virtualLinks>
loop
[for each SF port]
createPortOnLSI(graphLSI,sfport→technology)
∗port
loop
[for each SAP]
createSAPOnLSI(graphLSI,SAP→description)
∗sap
list<SFport> SFports,list<SAP> saps,list<Link> virtualLinks
createAndStartSFs(sffg.SFs)
loop
[for each SF]
createSF(sf→ports,sf→parameters)
∗sf
startSF(sf)
list<SFs∗ > sfs
createTSrules(sffg.flowRules,virtualLinks)
loop
[for each SF-FG flow rule]
createTSrule(graphLSI,flowRule)
createTSrule(LSI-0,flowRule)

Figure 7: c-orch new SF-FG deployment message sequence diagram.

Table 2: 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 (SF) port with a specific technology on an LSI
void destroyPort(port)	Destroy a specific (SF) port
sap ∗createSAPOnLSI(lsi, description)	Create a SAP on an LSI, according to a specific description
void destroySAP(sap)	Delete a specific SAP

Table 3: 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

controls the forwarding table of the LSI itself[2]. In the current prototype we opted to support OpenFlow for traffic steering; thus, each technology-dependent controller is actually a minimal OpenFlow controller exploiting OpenFlow `flowmod` messages to set up the traffic steering rules. Similarly to the management interface, further technologies to program the forwarding table(s) of the vSwitch can be supported by writing the corresponding driver implementing the primitives of Table 3.

5.4.1. Translating SF-FG Flow Rules for Traffic Steering

In order to implement the network paths described in the SF-FG, the network manager has to map the `flowrules` requested by the SF-FG on the traffic steering model defined in Section 5.2. Particularly, this model requires that, for each SF-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 SF-FG, originates two sets of traffic steering rules to be installed respectively in `LSI-0` and in the new graph-LSI.

According to Table 4, 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.

Table 4: LSIs involved in implementing flow rules with specific match/action.

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

[2] Also in this case, only one technology at a time is supported, i.e., all the LSIs are controlled through the same technology.

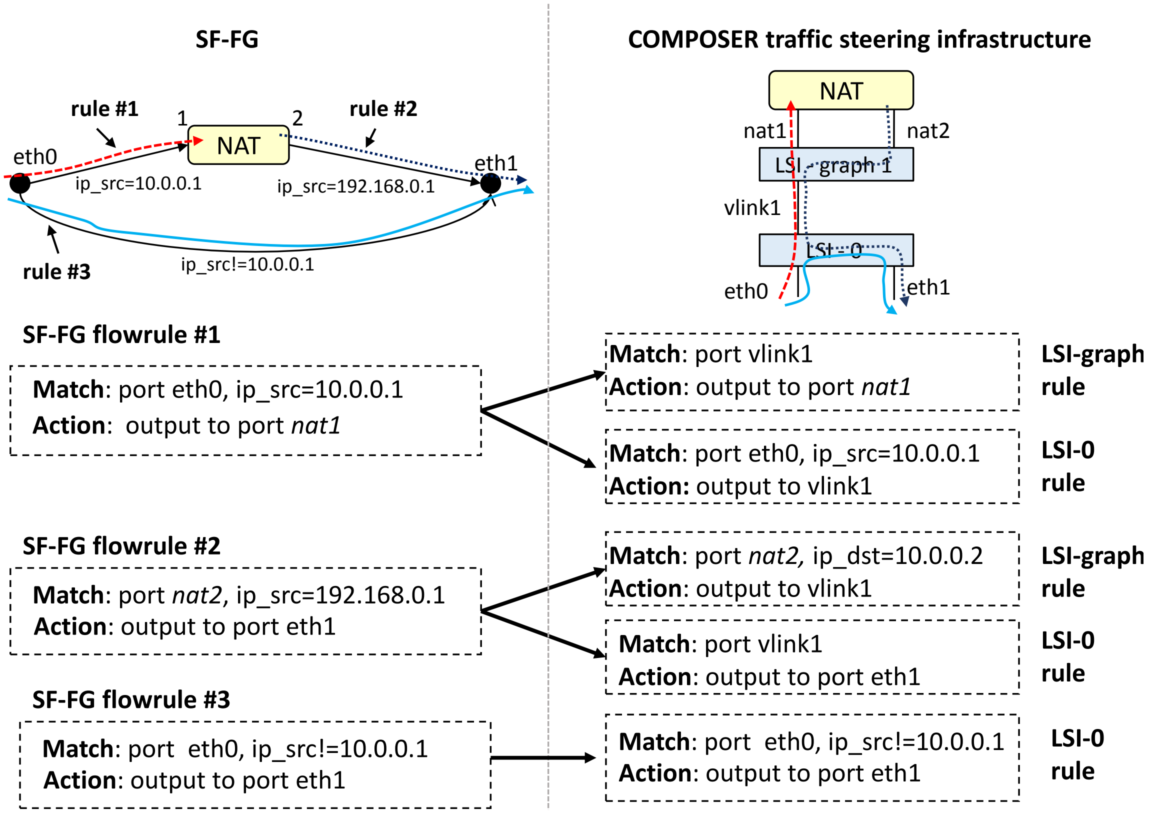
Particularly, as shown in Algorithm 1, we chose to create a different virtual link for each *different* SAP/SF port that appears in the action of rules involving both LSIs (Table 4), which is then used to move all traffic that must be sent on that specific SAP/SF port from one LSI to another. According 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 SF port/GRE SAP/host stack SAP, and to send towards `LSI-0` all traffic for a specific interface or vlan SAP.

Algorithm 1 Virtual links creation.

```
procedure createVlinks(first_id,sffg,lsi0,graphLsi)
vlink_to_lsi0 $\leftarrow$ vlink_to_graphlsi $\leftarrow$ first_id
vlinks $\leftarrow \emptyset$
for all r $\in$ sffg.flowrules do
  if vlink_needed[r.match.port][r.action.out] and vlinks[r.action.out] $\in \emptyset$ then
    if r.action.out $\in$ sf_port or r.action.out $\in$ gre_sap or r.action.out $\in$ hoststack_sap then
      vlinks[r.action.out] $\leftarrow$ vlink_to_graphlsi
      vlink_to_graphlsi $\leftarrow$ vlink_to_graphlsi+1
    else if r.action.out $\in$ interface_sap or r.action.out $\in$ vlan_sap then
      vlinks[r.action.out] $\leftarrow$ vlink_to_lsi0
      vlink_to_lsi0 $\leftarrow$ vlink_to_lsi0+1
    end if
  end if
end for
N $\leftarrow$ max(vlink_to_lsi0,vlink_to_graphlsi)$-$first_id
return connectLSIs(lsi0,graphLsi,N)
```

Transforming a SF-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 example is provided by the SF-FG `flowrule #3` of Figure 8). Algorithm 2 shows how the network manager derives the traffic steering rules corresponding to SF-FG `flowrules` involving both LSIs, after that the SF ports/SAPs have been associated with one of the virtual links just created. Particularly, according to lines `#4-#12` of Algorithm 2, 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 SF-FG `flowrule #1` of Figure 8, where `port 1` of the `NAT` is associated with the virtual link `vlink1`.

Lines `#13-#21` of Algorithm 2 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` show that the rule created for `LSI-0` just matches the proper virtual link. An example of this procedure is shown for the SF-FG `flowrule #2` in Figure 8, where the virtual link used to transfer traffic to `eth1` is the same one used to bring traffic to the port of the NAT.



SF-FG flowrule #1

Match: port eth0, ip_src=10.0.0.1
Action: output to port *nat1*

SF-FG flowrule #2

Match: port *nat2*, ip_src=192.168.0.1
Action: output to port eth1

SF-FG flowrule #3

Match: port eth0, ip_src!=10.0.0.1
Action: output to port eth1

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

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

Match: port eth0, ip_src!=10.0.0.1
Action: output to port eth1
LSI-0 rule

Figure 8: Example of transformation of SF-FG `flowrules` in traffic steering rules.

Algorithm 2 Traffic steering rules creation.

```
procedure splitRules(vlinks,sffg)
for all r ∈ sffg.bigswitch do
    if vlink_needed[r.match.port][r.action.out] then
        if r.action.out ∈ sf_port or r.action.out ∈ gre_sap or r.action.out ∈ hoststack_sap then
            {The rule brings traffic from LSI-0 to the graph-LSI}
            {Create the rule for LSI-0}
            rule-LSI0.match ← r.match
            rule-LSI0.action.out ← vlinks[r.action.out]
            rule-LSI0.action.other ← r.action.other
            {Create the rule for the graph-LSI}
            rule-graphLSI.match.port ← vlink[r.action.out]
            rule-graphLSI.action.out ← r.action.out
        else if r.action.out ∈ interface_sap or r.action.out ∈ vlan_sap then
            {The rule brings traffic from the graph-LSI to LSI-0}
            {Create the rule for LSI-0}
            rule-LSI0.match.port ← vlink[r.action.out]
            rule.LSI0.action.out ← r.action.out
            {Create the rule for the graph-LSI}
            rule-graphLSI.match ← r.match
            rule-graphLSI.action.out ← vlink[r.action.out]
            rule-graphLSI.action.other ← r.action.other
        end if
    end if
end for
```

5.5. Compute Manager

The *compute manager* interacts with the available execution environments to manage the SF lifecycle, including operations needed to attach SF ports already created on the vSwitch (by the network manager) to the SF itself. The compute manager module can interact with different execution engines, and can thus manage SFs based on different technologies, through the *compute interface* defined in Table 5.

As shown in Figure 3, this abstraction is implemented by a set of drivers, each one in charge of a specific execution environment technology. Currently, COMPOSER supports the QEMU/KVM hypervisor, Docker containers, processes based on the DPDK framework [42], and native network functions. Particularly, the KVM driver interacts with the QEMU/KVM hypervisor through the `Libvirt` API, while the Docker driver uses the CLI defined by Docker to manage SFs executed in

Table 5: Compute interface.

Function	**Description**
sf *createSF(ports,other parameters)	Allocate the resources needed by a SF; create a local copy of/download the SF image
void destroySF(sf)	Release the resources allocated to the SF
void startSF(sf)	Start a SF previously created
void stopSF(sf)	Stop a SF, without deallocating resources
void updateSF(sf,. . .)	Update a running SF (e.g., remove/add network interfaces)
void pause(sf)	Suspend the execution of the SF (e.g., for a possible migration)

containers. Multiple technologies are supported at the same time. For instance, c-orch can deploy a service including a first SF executed in a Docker container and a second SF running inside a VM.

5.6. SF resolver

The *SF resolver* is the part of c-orch that interacts with the SF repository in order to select the SF images to be instantiated. The SF resolver is used when the SF-FG requirements (e.g., *firewall with 3 ports*) can be satisfied by multiple SF images. In this case, it selects the best image through the following steps. First, it asks to the SF repository the templates of all SF implementing the function (e.g., firewall). Subsequently, it selects the best SF that, according to the template, satisfies the constraints/attributes associated with the function in the SF-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 SF 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 SF as another SF-FG, according to the principle of hierarchical decomposition. In this case, the SF resolver recursively repeats the operations described above for each function that is part of the new *sub*-SF-FG, until all required SF images have been selected.

5.6.1. SF Repository

The SF repository contains the templates and images of the available SFs. The *SF template* describes a specific SF image in terms of functionality implemented (e.g., firewall, storage server), 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 *SF image* varies according to the technology implementing the SF; as an example, it is the VM disk in case of virtual machines.

5.7. Internal Message Bus

As shown in Figure 3, COMPOSER includes an internal message bus implemented by DoubleDecker (DD) [44]. Although in the picture only c-orch, the monitoring manager and the monitoring functions (Section 5.8) are connected to such a bus for the sake of clarity, SFs may be connected to the DD bus as well, for instance in order to receive monitoring alarms or configuration parameters.

The connection between SFs and the internal message bus can be defined through the *host stack* SAP, since the DD broker (namely the module actually implementing the bus) is a process waiting for messages on a specific TCP port of the device where COMPOSER is executed.

5.8. 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 SFs), 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 SF-FG and written according to the MEASURE monitoring language [46].

After the deployment of the SF-FG, the monitoring manager instantiates and configures the proper *monitoring functions (MFs)* (e.g., *Google cAdvisor* [52] and *Ramon* [53]) that monitor the required metrics and generate alarms on the COMPOSER internal bus.

The monitoring manager, in addition to starting and configuring the proper MFs, receives alarms through DD, aggregates the received information as required by the MEASURE instructions, and propagates them again on the DD bus. This way, the aggregated information can reach the interested SFs. For instance, a SF may exploit the monitoring results to require an update of the SF-FG, so that it can properly react to the received event.

6. COMPOSER data plane features

According to Section 5, the c-orch is the main component of the COMPOSER control plane, as it takes care of satisfying all the requests received through its northbound API (e.g., deploy a new service graph). However, COMPOSER also brings together in a unique platform several innovations related to the data plane, making them seamlessly usable in real deployments. Main data plane features, described in the remainder of this section, are the following: *(i)* the possibility to exploit native (software and hardware) modules to execute SFs with reduced overhead; *(ii)* the possibility to dynamically create a direct communication channel between two VMs, which bypasses the virtual switch, in case the service to be deployed satisfies some constraints; and *(iii)* the support for the Ericsson Research Flow Switch, which targets high-speed packet networks.

Before detailing these aspects of the COMPOSER data plane, it is worth mentioning that such features, although implemented and validated in COMPOSER, are orthogonal to our platform, whose main novelty consists in the control plane. In addition, the last two are almost all compartmentalized and implemented in the vSwitch and require little support for an orchestrator, although, so far, COMPOSER is the only platform that integrates them in a usable prototype.

6.1. Native Service Functions

Unlike VMs and Docker containers that are well-known virtual execution environments, the COMPOSER concept of Native Service Functions (NSFs) [38] enables the deployment of service graphs on current, resource-constrained CPEs, which are then fully integrated in the programmable carrier infrastructure, as opposed to requiring their own silo for control and management. In fact, as shown later in Section 8.6, NSFs 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 NSF as a data plane processing component that exploits capabilities natively present on the platform, which are instantiated as software or hardware modules. Each NSF is executed directly on the host. In practice, a NSF can be implemented as a `tgz` archive containing a set of bash scripts that are called by the *NSF driver* (see Figure 3) to implement the functions defined in Table 5, required in the SF lifecycle management.

Obviously, to be able to execute a particular NSF, all required modules, or *dependencies*, must be available on the node. Then, besides all information required for the execution of a generic SF (e.g., number of ports), the template associated with a NSF 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 NSF to operate. The SF 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 SFs, NSFs rely on scripts executed in the host operating system. As such, the NSF driver needs to explicitly implement a layer that provides some form of isolation of the NSF against the rest of the system. Particularly, the NSF driver creates a *network namespace* before starting the NSF, adds to that namespace the virtual ports required to connect the NSF to the LSI (we use virtual Ethernet (`veth`) interfaces), and then starts the NSF within the namespace.

Launching a NSF, 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. However, it is worth noting that, in any case, no protection exists that prevents a SF, which is expected to provide a given service (e.g., firewall, captive portal), 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 SF is malicious is emphasized in case of NSF 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 SF. In the context of the UNIFY project, such operational aspects have been considered in [54, 46].

6.2. Direct SF-to-SF communication

Figure 9 illustrates a service graph and one of its possible implementations in COMPOSER: all SFs are executed inside VMs interconnected through the vSwitch. While point-to-multipoint connections actually require the vSwitch in order to classify and send traffic to the proper next SF, point-to-point (`p-2-p` in the following) links could be implemented with a direct communication path, hence taking the vSwitch out of that portion of the data plane. This may result in higher throughput, lower latency, and lower resource consumption thanks to the CPU cycles saved by avoiding a further pass through the vSwitch.

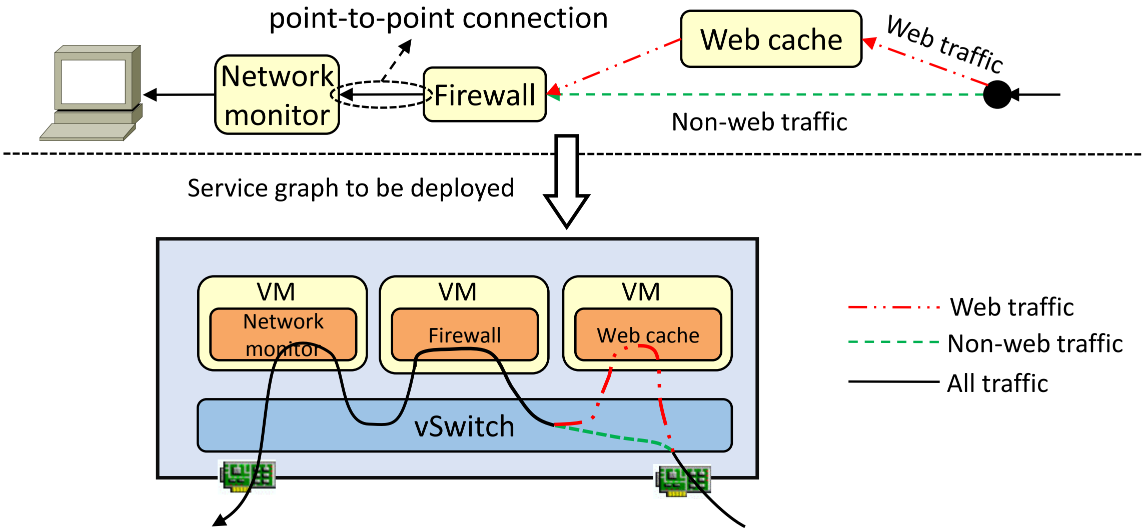


Figure 9: Traffic crossing SFs: the service graph (up); its implementation on COMPOSER (down).

COMPOSER supports an extended version of OvS-DPDK that can accelerate *transparently* and *dynamically* packet exchanges between two VMs, by creating a direct connection between them in case of `p-2-p` links [39][55]. *Transparency* in this context refers to the possibility for a SF to exploit direct communication without any knowledge of this optimization, and for an OpenFlow controller to attach to OvS as usual. *Dynamicity* refers to the capability to either create a direct VM-to-VM channel or return to a traditional VM-to-vSwitch-to-VM path on the fly, based on the runtime analysis of the service graph that is being instantiated or modified.

The framework to create direct connections between SFs has been designed to operate in case of SFs implemented as DPDK-based applications executed inside VMs and connected to the OvS forwarding engine through `dpdkr` ports. The OvS forwarding engine processes packets according to the content of its forwarding table, which can be configured with OpenFlow `flowmods` messages. `dpdkr` ports are implemented using shared memory and are exposed to the VM through `ivshmem` devices; moreover, SFs access `dpdkr` ports using a poll mode driver (PMD).

As shown in Figure 10, `dpdkr` ports have been modified to include a *standard* channel connected to the OvS forwarding engine, while the optional *bypass* channel is directly connected to another VM. The PMD has been modified too, so that the same instance can manage both channels and expose them as a single `dpdkr` port to SFs, which are not aware of the actual implementation of that port. Moreover, a new *p-2-p link detector* module has been added to OvS, which analyses each `flowmod` received by the vSwitch and dynamically detects the creation/destruction of a `p-2-p` link between two `dpdkr` ports.

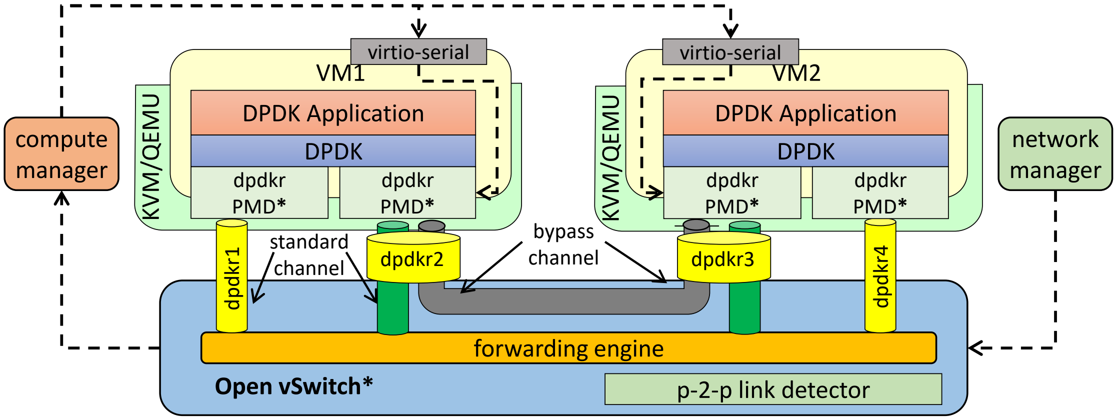


Figure 10: Implementing direct channels between VMs in OvS. Components marked with '*' have been modified to support transparent inter-SF communication.

When the VM is created (e.g., by c-orch), it is connected to `dpdkr` ports that have only the *standard* channel. Then, when the vSwitch detects the creation of a `p-2-p` link, it creates a new pair of *bypass* channels mapped on the same piece of memory, shared by both the communicating VMs. This way, the two VMs are directly connected and able to exchange packets without the intervention of the OvS forwarding engine.

The new *bypass* channels must be plugged into the proper VMs, and assigned to the proper PMD instance. To this purpose, the vSwitch has to rely on c-orch, which receives requests from OvS to: *(i)* plug the *bypass* channel (as an `ivshmem` device) into the VM by interacting with QEMU; *(ii)* configure the PMD instance to use the *bypass* channel by means of a control mechanism based on a `virtio-serial` device.

Notably, in case of `dpdkr` port involved in a `p-2-p` link, the PMD can still receive packets from the *standard* channel. In fact, it may happen that the OpenFlow controller sends to OvS an OpenFlow `packet-out` message containing a packet that must be sent through such a port; in this case, OvS uses the *standard* channel to provide the packet to the VM.

To keep compatibility with external components (e.g., the OpenFlow controller), OvS exposes the *standard* and *bypass* channels as a single (standard) `dpdkr` port, so that the above entities can continue to issue commands involving those ports as they usually do (e.g., get statistics, turn them on/off). For the same reason, the PMD has been extended so that, when a packet is sent through the *bypass* channel, it increases the counters associated to that OpenFlow rule and port, which are stored in a shared memory area. When the vSwitch needs to export statistics, it reads the proper values from that memory.

6.3. Ericsson Research Flow Switch

The Ericsson Research Flow Switch (ERFS) [40] is an optimized OpenFlow pipeline built directly for supporting the OpenFlow 1.3 specification without any legacy support for earlier versions. ERFS uses a novel switch architecture that capitalizes on the observation that OpenFlow pipelines are sufficiently structured to admit efficient machine code representations, constructed out of simple packet processing and classification templates.

High performance is the main focus; to achieve this goal, the two main optimizations that ERFS uses are the *real-time* and *per-table optimization* for classification/lookup and the *just-in-time* building of the packet processing code, as detailed in the following.

Lookup/classification. In ERFS the lookup algorithm is selected optimally on a per table basis. This means that based on the incoming rules to a given table a real-time decision takes place that tries to find the best possible classification algorithm for that table.

JIT - ERFS uses just-in-time linked code for executing parsing, (some) lookup algorithms and the actions that are to be executed in the given table or group. This means that the packet processing code is generated dynamically from code templates for the most widely used OpenFlow actions - regardless whether they are mandatory, optional or experimental.

The resulting specialized data paths is shown to give major gains over flow-caching-based alternatives, with several times higher raw packet rates, much smaller latency, and, perhaps most importantly, robust and predictable performance even with widely varying, or straight-out borderline malicious, workloads. The ERFS switch architecture easily scales to hundreds of flow tables and hundreds of thousands of traffic flows, while supporting updating the fast path at similar, or higher intensity.

7. Domain-specific services

Thanks to its flexibility and generality, COMPOSER supports different service models, which range from pure OpenFlow-based services to NFV services defined according to the ETSI MANO architecture. In order to exemplify how COMPOSER can be used to implement such services, the remainder of this section will first discuss how our platform can provide SDN services, then it will compare COMPOSER with the ETSI model and show how it can be integrated into an ETSI MANO architecture in order to deliver NFV services.

7.1. Support for Pure OpenFlow Deployments

In a Software Defined Network (SDN), applications running on top of OpenFlow controllers such as OpenDaylight [56] and ONOS [57] “program” the forwarding tables of different (physical

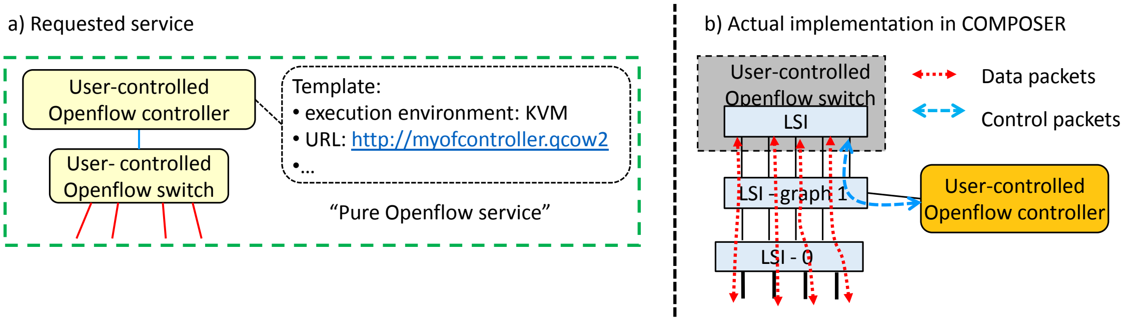

a) Requested service
User-controlled Openflow controller
User- controlled Openflow switch
Template:
• execution environment: KVM
• URL: http://myofcontroller.qcow2
•…
"Pure Openflow service"
b) Actual implementation in COMPOSER
User-controlled Openflow switch
LSI
LSI - graph 1
LSI - 0
Data packets
Control packets
User-controlled Openflow controller

Figure 11: Exploiting COMPOSER to deploy an OpenFlow service.

and/or virtual) OpenFlow switches in order to implement traffic steering and some more advanced functionalities (e.g., NAT, firewall). While controllers can be deployed in some virtualized environments (e.g., VM or Docker), the switch is usually deployed on the bare metal, as it should provide high performance [58].

In order to implement a "pure" OpenFlow deployment with COMPOSER, both the controller and the vSwitch(es) must be considered as SFs that are part of a service graph. Specifically, this requires the definition of a service graph that includes the OpenFlow controller (specified through the proper template that maps the NF with a specific image/implementation) connected to a generic OpenFlow switch (Figure 11).

This allows c-orch to select the most efficient implementation for the vSwitch. Particularly, if COMPOSER uses an OpenFlow-based switch to implement connectivity among SFs (Section 5.4), c-orch can deploy the OpenFlow switch as a NSF by creating an additional LSI in the vSwitch, but executed in a separate network namespace. Obviously, unlike `LSI-0` and the `graph-LSIs`, the new LSI is completely controlled by the OpenFlow controller SF and hence implements the behavior required by the deployed service graph.

Using the infrastructure vSwitch to implement a SF may lead to performance improvements because of the lighter virtualization layer and the reduced resources required by NSFs with respect to, e.g., VMs. In addition, the vSwitch may even implement some algorithm to optimize the interconnections among LSIs, in case some conditions on the traffic steering rules are identified.

7.2. COMPOSER and ETSI NFV

The European Telecommunications Standards Institute (ETSI) has defined a reference model for the NFV architecture [59], which is reproduced in Figure 12 [21]. The main functional blocks include an orchestrator (NFVO) that handles the lifecycle of the services, a (set of) VNF manager (VNFM) in charge of managing one or more VNFs (i.e., start/stop, configure, scale in/out), a Virtual Infrastructure Manager (VIM) that actually implements the commands triggered by the NFVO and/or the VNFMs on the physical infrastructure. In addition, the NFV infrastructure (NFVI) includes both physical and virtual resources (hypervisor, vSwitch, etc.) and actually executes the service. Finally, the ETSI reference model defines reference points between functional blocks, and descriptors (e.g., service descriptors, VNF descriptors, etc.).

Although the COMPOSER architecture is not explicitly designed for NFV (in fact, in the paper we always referred to generic service functions, which may include cloud computing services, OpenFlow services as described in Section 7.1, and VNFs), at first look COMPOSER resembles

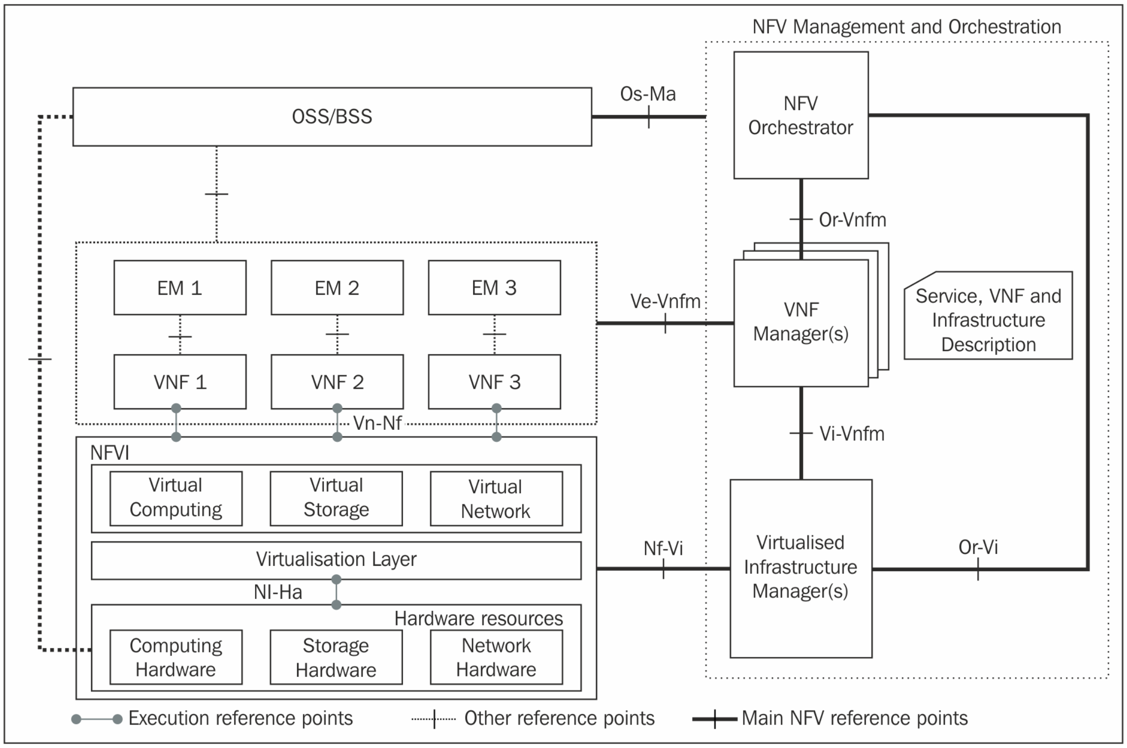


Figure 12: ETSI NFV architecture.

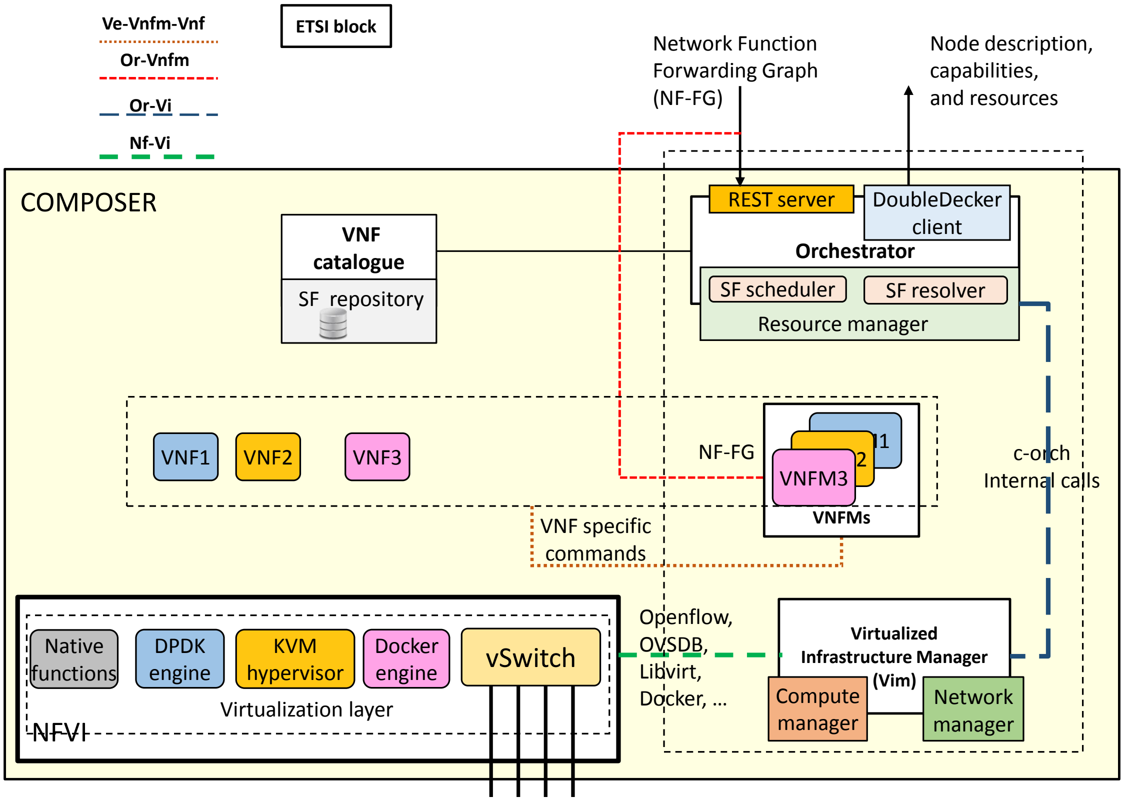


Figure 13: Mapping COMPOSER components to the ETSI NFV architecture.

an ETSI infrastructure node as it includes both a vSwitch and a set of execution engines with its VIM, which can be driven by an external NFVO sitting on top of many COMPOSER nodes. However, viewed in this way, much of our proposal potential is left untapped by such an external orchestrator. In fact, as shown in Figure 13, which maps the functional blocks and reference points of the ETSI NFV architecture to the COMPOSER components, our node includes also an NFVO and can support VNFMs.

Notably, the NFVO inside COMPOSER does not preclude the existence of one (or more) orchestrators on top of many COMPOSER nodes. Indeed, it provides the additional benefit of further optimizing service deployment, e.g., by selecting the best available implementation for a VNF, and at the same time provides more scalability, since an external orchestrator does not need to know the intricate details (e.g., the VNF images actually implementing a NF) which are only known to c-orch.

While COMPOSER implements the NFVO, VIM and NFVI, VNFMs are not actually part of the COMPOSER architecture. Such VNFMs are in fact VNFs themselves and therefore part of the SF-FG, as shown in the example of Figure 14 (this example is discussed below); this is the reason why the dashed box in the picture is extended to include both the VNFs and the VNFMs. It is worth mentioning that from the c-orch point of view, there is no difference between *data plane* VNFs and VNFs implementing the VNFM functionality, as both types are executed in some execution

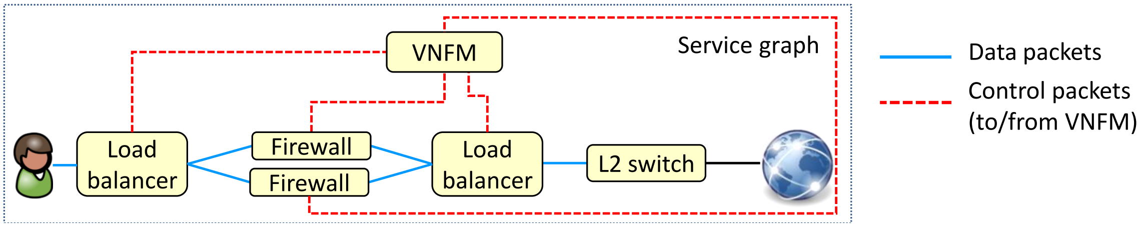


Figure 14: Deploying a service with the VNFM (Virtual Network Functions Manager) in COMPOSER.

environment and are connected to the vSwitch with a number of vNICs.

Finally, the SF-FG represents an implementation of the ETSI network service descriptor (NSD), as it describes the service in its whole, i.e., both the compute (VNFs) and network (links) parts.

7.2.1. Deploy a VNFM in COMPOSER

Figure 14 shows an example of service graph that includes a VNFM that manages two firewalls and two load balancers operating on the traffic in transit between the end user and the Internet. In fact, if one (or more) VNFM is needed for the service to operate, COMPOSER requires that such a component is defined as any other VNF in the service description, as well as such a description must indicate how the VNFM is connected with the other modules of the service itself. In the example of Figure 14 the VNFM is connected with the two firewalls, with the two load balancers, and with the Internet; this last connection allows in fact the VNFM to contact the c-orch in order to require modifications to the service. For instance, it may ask the deployment of a third firewall between the two load balancers, together with the requirement of adding a new vNIC to such VNFs needed to establish the new connection. After that the c-orch added the new vNICs to the load balancer VNFs, the VNFM interacts again with such functions in order to configure them to use the new network interface.

Finally, the L2 switch VNF is required to dispatch the traffic coming from the Internet either to the VNFM in case it is a packet directed to such a component (e.g., an answer from the c-orch), or to the load balancer if it is traffic to be processed by the service data plane.

8. Experimental evaluation

COMPOSER, whose source code is available at [1], was used as a validation platform for the FP7 projects UNIFY [30], SECURED [60], and the H2020 project 5GEx [32]; in addition, we carried out an extensive set of tests that are reported in the reminder of this section. Particularly, at the best of the author's knowledge COMPOSER has been successfully installed and executed on the hardware platforms shown in Table 6, which feature very different characteristics (also in terms of operating system and software build mechanisms) and represent the huge variety of hardware running in carrier network deployments[3]. Although portability of C/C++ software does not seem a big challenge nowadays, in fact the portability of a complex software, relying on many external components/libraries is still a major issue as far as embedded platforms are concerned, due to many

[3] The exact hardware used to get the reported results will be specified during the description of each specific test.

Table 6: Hardware platforms known to be able to execute COMPOSER.

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

limitations such as having a custom version of the operating system, proprietary drivers, dedicated hardware components, stripped-down software (e.g., the operating system does not contain all the features, tools and libraries of the full-fledged version), and more.

In the remainder of this section we will first show the resource consumption (memory and CPU) of COMPOSER, including also the vSwitch and the execution engine(s); such results are compared with OpenStack, as it represents one of the leading platform used to execute virtualized services, being them either VNFs (e.g., firewall, NAT) and more traditional cloud-oriented services (e.g., storage server, web server). We will then consider the scalability of COMPOSER, as well as we demonstrate the advantages, in terms of performance, of the creation of direct connections between SFs, and of the usage of the ERFS vSwitch in comparison with vanilla OvS-DPDK. Next, we demonstrate how COMPOSER can deploy network services on a residential CPE thanks to the NSFs, which are well suited for resource constrained environments. We will then present, from a functional point of view, the Elastic Router service, namely a service that adapts itself based on information gathered by monitoring functions dynamically started and configured by the COMPOSER framework. Finally, we terminate our validation campaign by discussing the portability of our prototype, and by highlighting some limitations that we have experienced when building and running COMPOSER on specific hardware platforms.

8.1. 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 SFs as of this writing.

Starting from a clean installation of Ubuntu server 14.04 LTS with default settings on the high-end server (Table 6), we set up, one at a time, the configurations shown in Table 7, 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.

As reported in Table 7, COMPOSER compiled only with support for NSFs represents the lightest configuration, requiring less than 20% of RAM and less than 15% of disk space than an

Table 7: Resources consumed by COMPOSER and OpenStack.

Configuration	**RAM (MB)**	**Disk (MB)**
COMPOSER- only NSF 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

OpenStack compute node, as NSFs are executed natively by a compute manager which launches shell scripts. As expected, KVM and Docker are more resource demanding than NSFs 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 OpenStack compute node. As expected, the most resource consuming configuration for COMPOSER 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.

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

To conclude, it is worth mentioning that the *Revised OpenFlow Library (ROFL)* [61], which is currently employed to handle the OpenFlow messages between c-orch and the vSwitch, requires 49MB of disk space. We expect that in the future further optimization in this direction, e.g. by using a lighter OpenFlow library, is possible.

8.2. Service Deployment Time

This section compares the time required by COMPOSER and OpenStack to deploy the two service graphs shown in Figure 15(a), which include SFs that will be instantiated with dedicated VMs. 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 #2 (Table 6) 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 SF repository 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 it from the image server[4].

Tests are repeated 100 times and averaged; results are reported in Figures 15(b) and 15(c). In case of COMPOSER (Figure 15(b)), the picture reports the total time (average value, plus the minimum and the maximum values of the the margin of error calculated at 95% confidence level) 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 SF ports, and to set up the traffic steering rules into the

[4] 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.

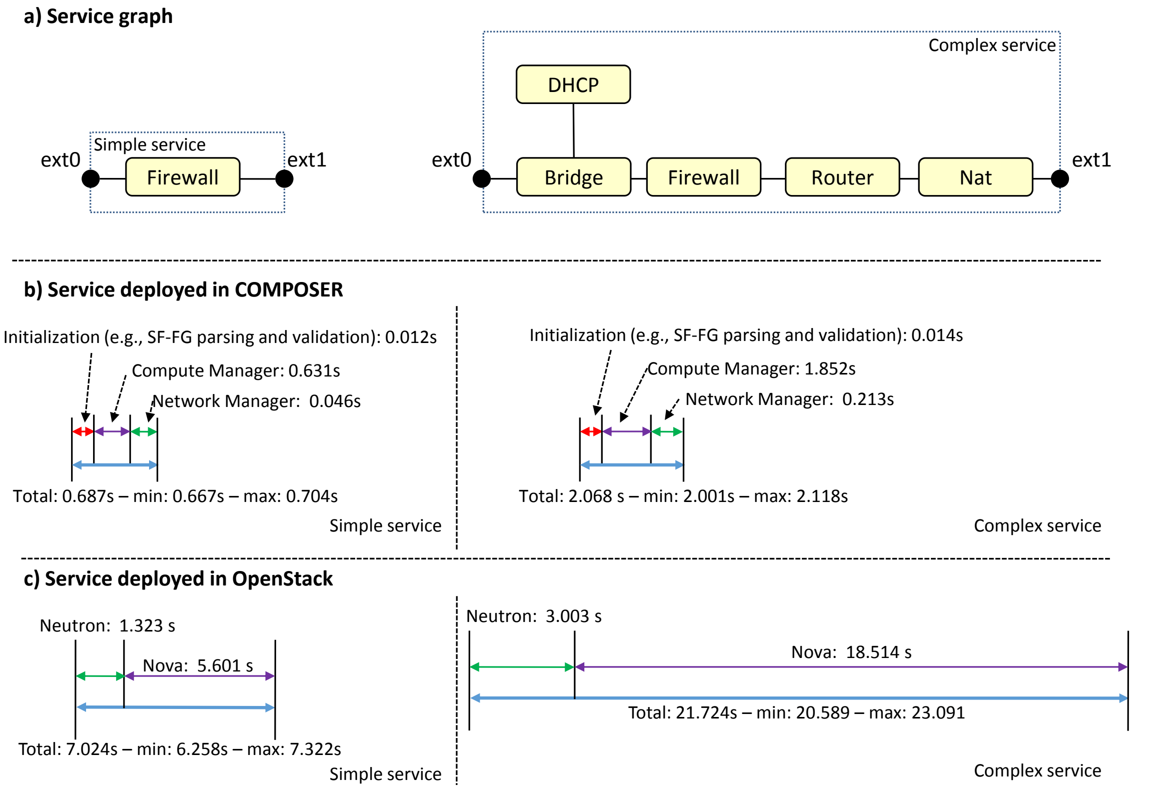

a) Service graph
Simple service
ext0
Firewall
ext1
Complex service
DHCP
ext0
Bridge
Firewall
Router
Nat
ext1
b) Service deployed in COMPOSER
Initialization (e.g., SF-FG parsing and validation): 0.012s
Compute Manager: 0.631s
Network Manager: 0.046s
Total: 0.687s – min: 0.667s – max: 0.704s
Simple service
Initialization (e.g., SF-FG parsing and validation): 0.014s
Compute Manager: 1.852s
Network Manager: 0.213s
Total: 2.068 s – min: 2.001s – max: 2.118s
Complex service
c) Service deployed in OpenStack
Neutron: 1.323 s
Nova: 5.601 s
Total: 7.024s – min: 6.258s – max: 7.322s
Simple service
Neutron: 3.003 s
Nova: 18.514 s
Total: 21.724s – min: 20.589 – max: 23.091
Complex service

Figure 15: Service deployment time: *(a)* used service graph; *(b)* results with COMPOSER; *(c)* results with OpenStack.

vSwitch; *(ii)* the time required by the compute manager to create and start the SF(s) as a VM(s). In the case of OpenStack, we report the total time (again the average value, plus the minimum and the maximum values of the the margin of error calculated at 95% confidence level) 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(s) (e.g., the firewall) to actually start, which depends on the software executed in the VM itself and not from the orchestration framework and is anyway equivalent in all the tests, given that we used exactly the same VMs for, e.g., the firewall.

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, for both the considered services; 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 is based on either REST or distributed message bus (RabbitMQ) calls, and that is much more complicated due to the necessity to support a datacenter-wide environment.

8.3. COMPOSER scalability

This section evaluates the scalability of COMPOSER, both in terms of time required to deploy a service and in terms of aggregated throughput supported by the service itself. To this purpose, we first instantiate the graph depicted at the top of Figure 16, where no SF is actually deployed on COMPOSER. Then, as shown in the picture, we deploy service graphs with an increasing number of personal firewalls, where each SF operates on the traffic to/from a different device identified through the MAC address of the device itself. In our opinion, this represents a concrete use case for COMPOSER when installed at the edge of the network and used as an Internet gateway, e.g., in a public building/at a public event.

Tests are executed on the mid-range server #2 (Table 6) equipped with a clean Ubuntu server 14.04 LTS image and all the components involved in the service deployment (i.e., the COMPOSER software and the SF repository); OvS is used as a vSwitch, while SFs are deployed as Docker containers. Tests are repeated 100 times and the average value of the samples, together with the margin of error calculated at 95% confidence level, is reported in Figure 17 (deployment time) and Figure 18 (throughput).

According to the former picture, the time needed to deploy a graph grows with the number of SFs, starting from an average value of less than 1s when the service does not include any SF, to almost 6min and half when the graph includes 500 SFs. It is worth to point out that the results do not include the time needed by the firewall application inside the docker container to actually start, which depends by the application itself and not by the orchestration framework. This growing trend may be due to some locking mechanism implemented by the Docker engine, since the c-orch, in order to run the SFs, starts N threads in parallel (where N is the number of SFs to be instantiated), each one interacting with the Docker engine using its command line interface.

Figure 18 shows instead the aggregate throughput provided by COMPOSER, measured using the `iperf` tool installed on two servers used respectively as traffic source/sink, and directly connected to COMPOSER through a 10Gbps link. The `iperf` client generates TCP streams at maximum speed, exploiting the maximum allowed MTU. As reported in the picture, the throughput is almost constant regardless of the number of SFs (and then Openflow rules) deployed, since in our test each packet always traverse a single SF (i.e., firewall) in each direction. Throughput scalability in case more SFs are chained will be discussed in Section 8.4.

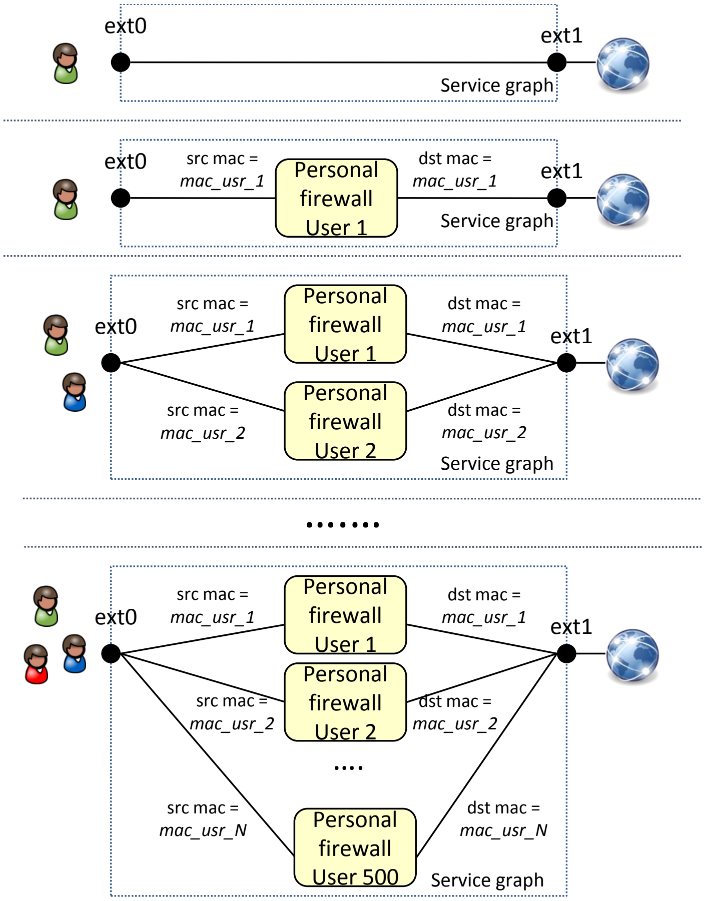


Figure 16: Service graphs with an increased number of SFs (firewalls in this case) deployed in parallel.

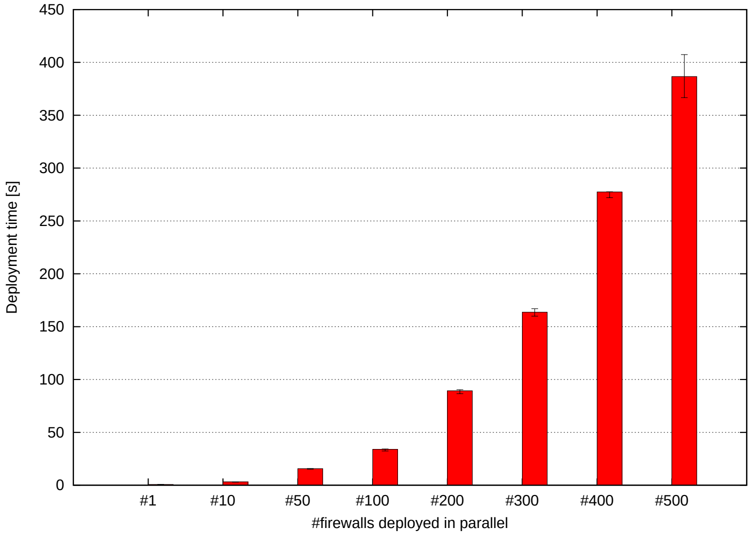


Figure 17: Time required by COMPOSER to deploy service graphs with a growing number of SFs.

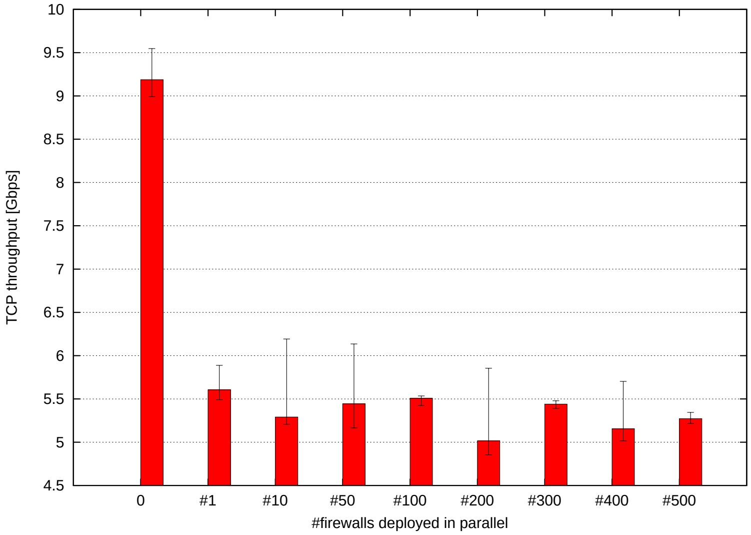


Figure 18: Throughput supperted by COMPOSER with service graphs of increasing complexity.

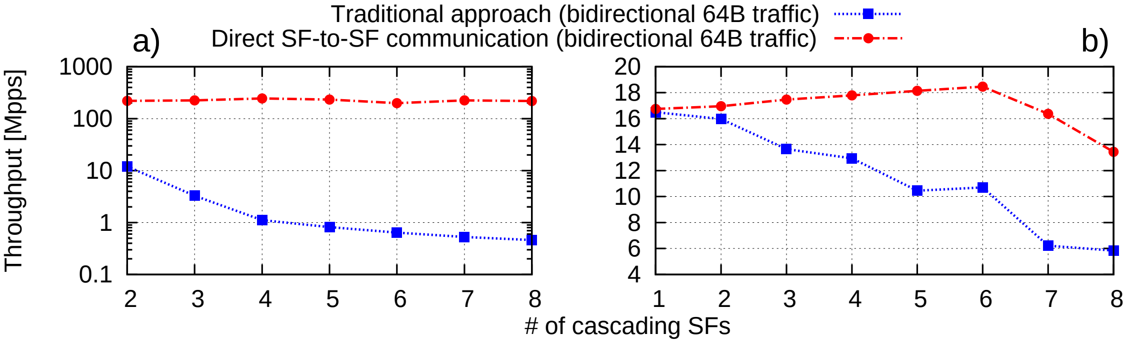


Figure 19: (a) memory-only; (b) NICs involved.

8.4. Direct SF-to-SF communication

We validated our idea of creating direct channels between SFs (Section 6.2) on an Intel Xeon (Table 6), comparing our approach with the vanilla OvS-DPDK. To this purpose, we use chains of VMs connected only through `p-2-p` links, where each VM has two `dpdkr` ports and runs a single core DPDK SF that forwards packets between them. Notably, due to the transparency of the direct SF-to-SF communication, we were able to use the same VM in all tests (with/without the bypass channel).

Figure 19 shows the throughput measured with chains of different length. In detail, Figure 19(a) refers to the case in which the first and the last SF of the chain act as traffic source/sink; this test validates our proposal without the overhead introduced by NICs and the PCI-e bus. Figure 19(b) refers instead to the case in which traffic is provided/drained to/from the chain through a couple of 10Gbps NICs. Both graphs show that a chain of SFs executed in VMs exploiting our technology provides far better throughput than the same chain based on the vanilla OvS-DPDK.

Figure 20 shows the latency measured with traffic coming from/delivered to the 10Gbps NICs, tuning the TX speed in order to avoid packet loss in the chain. Particularly, the picture reports the median value of the samples; the margin of error is of $\pm 0.4\mu$s in the worst case at the 95% confidence level and hence not visible in the graph. According to the graph, latency introduced by both approaches is almost the same until 5 chained VMs; then the values start to diverge, resulting in an improvement of about 80% in the case of 8 chained VMs. This is due to the inferior number of CPUs required to forward traffic with the SF-to-SF approach, which avoids the pass in the vSwitch and hence saves precious CPU cores that are still available when deploying longer chains.

To conclude, the set up of a direct channel between two VMs, from the moment in which the vSwitch identifies a p-2-p link, to the moment in which the PMD starts to use the bypass channel, is about 100 ms.

8.5. ERFS Forwarding Performance

When using the same technology to implement traffic steering among SFs (e.g., OvS), both COMPOSER and an OpenStack compute node provides the same performance from the point of view of the network traffic (e.g., throughput). However, several vSwitches are supported by COMPOSER; particularly, unlike the current OpenStack release, it can implement traffic steering through the high-speed ERFS.

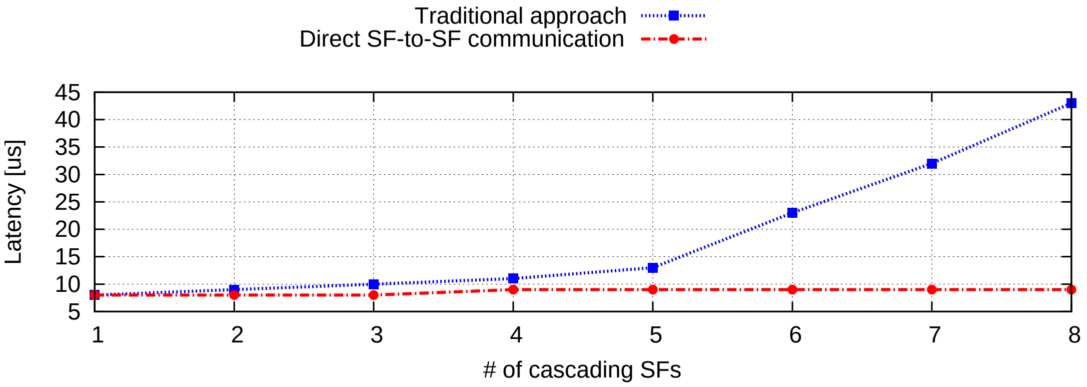


Figure 20: Latency when physical NICs are involved.

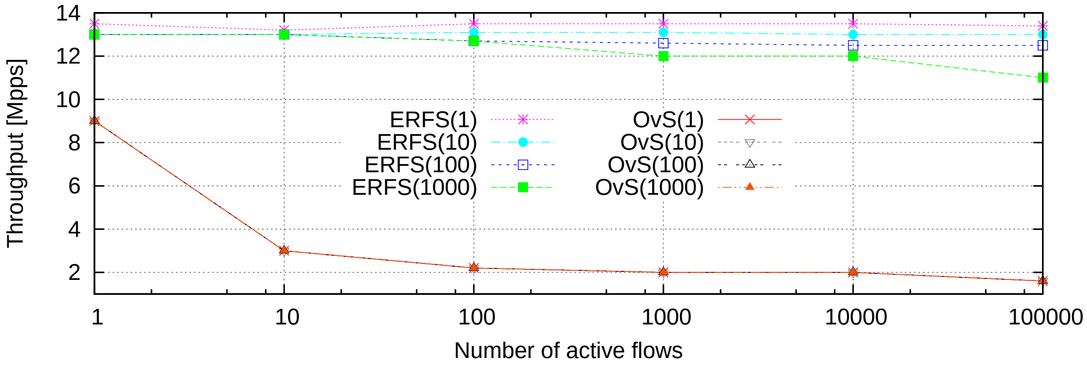


Figure 21: Packet rate for L2 switching over MAC tables of size 1, 10, 100, and 1K entries.

In order to show the advantages of ERFS with respect to (vanilla) OvS-DPDK, we measured the throughput when the vSwitch implements the L2 switching on the high-end server of Table 6. During the test, a traffic source machine generating 64B packets at 10 Gb/s was directly connected to the high-end server, in turn connected through a second 10 Gb/s Ethernet interface to a traffic receiver machine. Tests were repeated with a different number of MAC addresses in the vSwitch forwarding table; for each number, we generated, in round robin, packets with different destination MAC addresses in order to match all the entries of the forwarding table.

Measured throughput is reported in Figure 21, which shows how ERFS outperforms OvS-DPDK regardless of the number of MAC address stored in the forwarding table and matched by generated packets.

8.6. Native Service Functions

This section validates the NSF idea from the point of view of throughput, CPU load, image size and time required to start the SF. 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.,

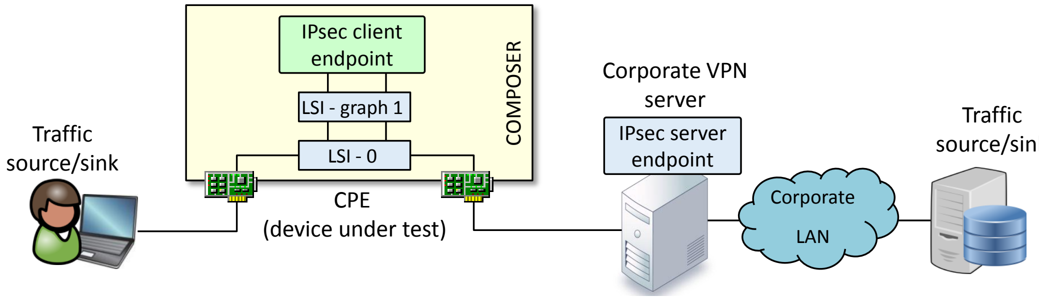


Figure 22: Testbed to validate the COMPOSER when running NSFs.

Table 8: Different implementations of the *IPSec client* SF.

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

laptop, smartphone, etc.), the user deploys the VPN client as a SF on the CPE, which provides secure access to the corporate network independently of the specific user terminal.

Our testbed, shown in Figure 22, includes two devices acting both as traffic source and sink, a CPE executing the IPsec client SF 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 6), 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 SF, whenever possible, as shown in Table 8.

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 8, NSFs 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, NSFs and Docker show the same level of performance, because they are based on the same technology (i.e., kernel-based processing in the host plus namespaces).

The last column of Table 8 reports the SF image size[5], which confirms the advantages of the NSF approach in resource-constrained environments. In fact, the reason for not testing VMs and

[5] 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 SF to work.

Docker on the residential gateway and on the professional CPE is the disk size limitation of these platforms. Moreover, the SF image size also impacts on the time required to download the SF 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 SF types. Averaged results show 3016 ms with VM (which requires to start the entire VM), 350 ms with Docker, and 727 ms with NSF (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 NSF is due to some implementation-dependent timeouts required to attach the network ports to the NSF, which still need to be optimized.

8.7. Dynamic Service Adaptation: the Elastic Router Example

This section provides a functional validation of the support of COMPOSER to services that dynamically adapt themselves based on metrics gathered by MFs deployed along with the service. To this purpose, we consider the Elastic Router [41, 62], a service that routes traffic between SAPs and that can automatically scale in/out according to the amount of traffic to be processed by the service.

Figure 23 depicts data gathered during the execution of the Elastic Router service. As shown, two metrics are used to detect conditions that make the service to scale: the CPU load (gathered by the *cAdvisor MF*) and the overload risk of the data plane components (evaluated by the *Ramon MF*). The labels `A, B, C, D` in the picture correspond to the main life-cycle events of the service, which are described in the following.

In `A`, the SF-FG shown in Figure 24(a) is provided to the c-orch, which starts both the *Ctrl App* and the switch functions as Docker containers (although the switch may be implemented as a NSF (i.e., LSI) using the infrastructure vSwitch, as detailed in Section 7.1); notably, while the former represents the control plane of the router, the latter implements the data path and forwards traffic according to the rules inserted by the *Ctrl App*. According to the picture, the *Ctrl App* is also connected to the internal bus in order to receive alarms generated by the monitoring system, since such a SF is also responsible to determine how to scale in/out data plane components, basing its decision on the information coming from the monitoring functions. This is why the SF-FG describing the Elastic Router service also includes instructions on how to deploy and configure monitoring components, expressed according to the MEASURE monitoring language and used by the monitoring manager of the c-orch. Particularly, as mentioned above, the provided MEASURE string requires the c-orch to configure the *cAdvisor* and the *Ramon* MFs, which respectively gather metrics related to the CPU load and bandwidth utilization.

In `B`, a traffic generator is started and packets are sent to the four SAPs of the service; initially, the single switch is enough to handle all the packets flowing through the service itself.

Then, starting from `C`, the traffic generator gradually increases the packet rate, until a threshold for the aggregated overload risk is reached. At this point, the MFs generate alarms on the message bus, which are received by the monitoring manager of COMPOSER; in turn, this component sends a *scale out* message on the message bus, which is then received by the *Ctrl App*. Such a SF then generates the new SF-FG shown in the right of Figure 24, which includes the number of switches required to handle the increased load, and provides it to the c-orch. It is worth mentioning that, while the scale-out is executed, the monitoring is put on hold, as shown by the gap in the graph of Figure 23 between points `C` and `D`.

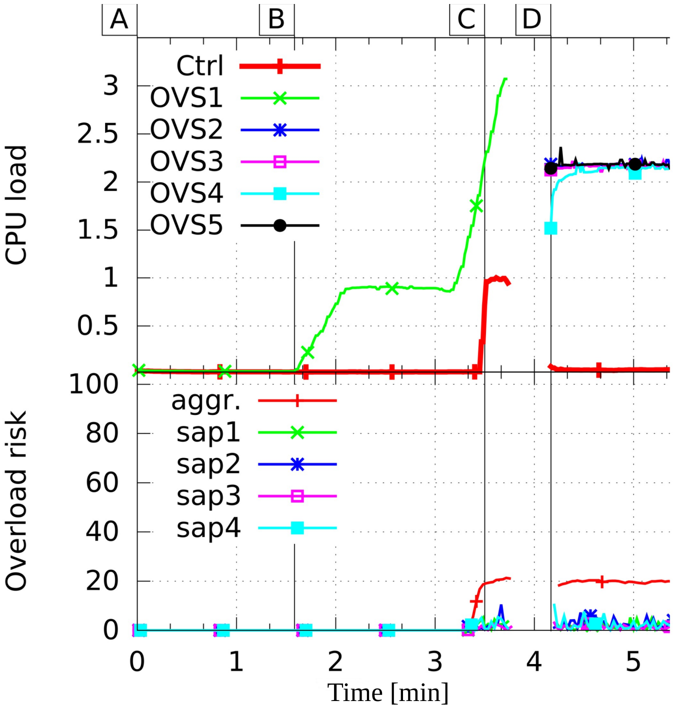


Figure 23: The monitored service data that leads to automated scaling. Main events are indicated with labels from A to D.

In D the scale out operation finally terminates, the MFs start again collecting data, and the switch SFs start forwarding traffic between the four SAPs; as depicted in Figure 23, four stream are now being generated, one for each switch.

8.8. COMPOSER portability

The portability of COMPOSER has been demonstrated by installing and running the software on all the hardware platforms shown in Table 6. It is worth to mention that, while many of them have been used as validations platforms for different aspects of our prototype as reported earlier in this section, others have only been exploited in order to showcase the prototype in the context of the FP7 project UNIFY [30], or have been used either by project members or third parties for their own purposes.

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 NSFs with OvS used as a vSwitch. However, since service access points such as tunnels (e.g., GRE) and VLANs were fully supported, we were able to instantiate services connecting COMPOSER to external domains and to create complex services requiring the stitching of multiple sub-graphs spanning across multiple domains [63]. Another example is the use of COMPOSER on CarOS, an embedded Linux distribution targeting carrier networks, running on a Banana Pi R1.

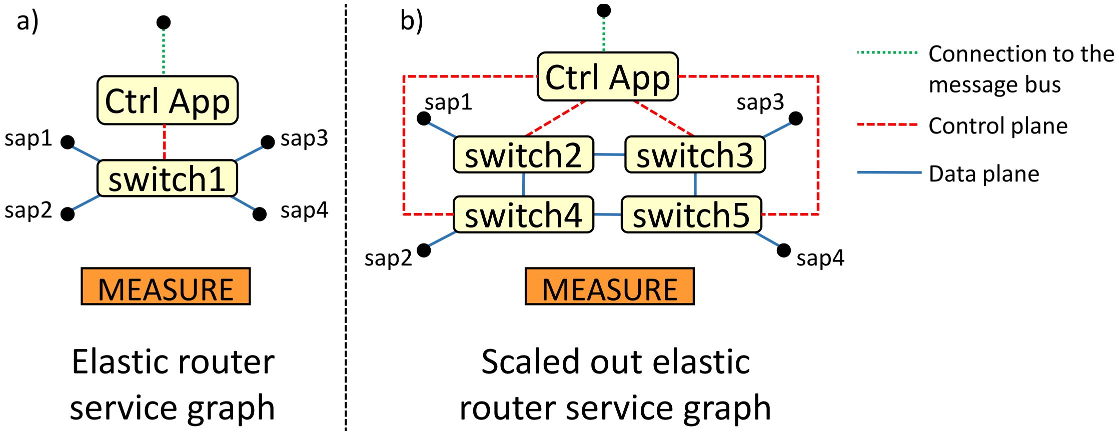


Figure 24: System view of the prototyped Elastic Router.

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 NSFs 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.

According to Table 6, 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.

9. Conclusion

This paper presents COMPOSER, a versatile and high-performance service platform that can execute several types of SFs 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 virtualization solutions in part because of the possibility to seamlessly employ, for instance, a high-performance underlying software switches (such as ERFS) as real-world deployment needs dictate, while at the same time retaining all the benefits of domain-oriented orchestration.

Acknowledgment

This work was conducted within the framework of the FP7 UNIFY project, which is partially funded by the Commission of the European Union. Study sponsors had no role in writing this report. The views expressed do not necessarily represent the views of the authors employers, the UNIFY project, or the Commission of the European Union. The authors wish also to thank David Verbeiren and Mauricio Vásquez Bernal for their precious help and suggestions, and all the colleagues working on the UNIFY project for the inspiring discussions.

References

[1] Universal Node Orchestrator, `https://github.com/netgroup-polito/un-orchestrator`.

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

[3] A. Csoma, B. Sonkoly, L. Csikor, F. Németh, A. Gulyás, W. Tavernier, S. Sahhaf, Escape: Extensible service chain prototyping environment using mininet, click, netconf and pox, ACM SIGCOMM Computer Communication Review 44 (4) (2015) 125–126.

[4] ETSI, Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV (2014).
URL `http://www.etsi.org/deliver/etsi_gs/NFV/001\_099/003/01.02.01_60/gs_nfv003v010201p.pdf`

[5] ETSI, Network Functions Virtualisation (NFV); Infrastructure; Methodology to describe Interfaces and Abstractions (2014).
URL `http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/007/01.01.01_60/gs_NFV-INF007v010101p.pdf`

[6] J. Hwang, K. K. Ramakrishnan, T. Wood, Netvm: High performance and flexible networking using virtualization on commodity platforms, IEEE Transactions on Network and Service Management 12 (1) (2015) 34–47. doi:10.1109/TNSM.2015.2401568.

[7] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, 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, ACM, New York, NY, USA, 2016, pp. 26–31. doi:2940147.2940155.
URL `http://doi.acm.org/2940147.2940155`

[8] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. K. Ramakrishnan, T. Wood, Opennetvm: Flexible, high performance nfv (demo), in: 2016 IEEE NetSoft Conference and Workshops (NetSoft), 2016, pp. 359–360. doi:10.1109/NETSOFT.2016.7502410.

[9] T. Wood, K. K. Ramakrishnan, J. Hwang, G. Liu, W. Zhang, Toward a software-based network: integrating software defined networking and network function virtualization, IEEE Network 29 (3) (2015) 36–41. doi:10.1109/MNET.2015.7113223.

[10] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, F. Huici, Clickos and the art of network function virtualization, in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), USENIX Association, Seattle, WA, 2014, pp. 459–473. URL `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins`

[11] L. Rizzo, G. Lettieri, Vale, a switched ethernet for virtual machines, in: Proceedings of the 8th international conference on Emerging networking experiments and technologies, CoNEXT '12, ACM, New York, NY, USA, 2012, pp. 61–72. doi:10.1145/2413176.2413185.

[12] Xen.
URL `http://www.xen.org/`

[13] R. Morris, E. Kohler, J. Jannotti, M. F. Kaashoek, The click modular router, in: Proceedings of the seventeenth ACM symposium on Operating systems principles, SOSP '99, ACM, New York, NY, USA, 1999, pp. 217–231.

[14] M. F. Bari, S. R. Chowdhury, R. Ahmed, 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, 2015, pp. 135–141. doi:10.1109/NFV-SDN.2015.7387418.

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

[16] R. Cziva, S. Jouet, K. J. White, D. P. Pezaros, Container-based network function virtualization for software-defined networks, in: 2015 IEEE Symposium on Computers and Communication (ISCC), IEEE, 2015, pp. 415–420.

[17] R. Cziva, S. Jouet, D. P. Pezaros, Roaming edge vnfs using glasgow network functions, in: Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, ACM, 2016, pp. 601–602.

[18] OpenStack community, Openstack.
URL `http://www.openstack.org/`

[19] F. Lucrezia, G. Marchetto, F. Risso, V. Vercellone, Introducing network-aware scheduling capabilities in openstack, in: Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft), 2015, pp. 1–5. doi:10.1109/NETSOFT.2015.7116155.

[20] V. A. Cunha, I. D. Cardoso, J. P. Barraca, R. L. Aguiar, Policy-driven vcpe through dynamic network service function chaining, in: 2016 IEEE NetSoft Conference and Workshops (NetSoft), IEEE, 2016, pp. 156–160.

[21] ETSI, Network Functions Virtualisation (NFV); Architectural Framework (2014).
URL http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf

[22] OpenStack community, OpenStack Tacker.
URL https://wiki.openstack.org/wiki/Tacker

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

[24] T. Fraunhofer FOKUS, Open Baton.
URL http://openbaton.github.io/

[25] Open Source MANO.
URL https://osm.etsi.org/

[26] J. Soares, M. Dias, J. Carapinha, B. Parreira, S. Sargento, Cloud4nfv: A platform for virtual network functions, in: Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on, 2014, pp. 288–293. doi:10.1109/CloudNet.2014.6969010.

[27] J. Soares, C. Gonalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, S. Sargento, Toward a telco cloud environment for service functions, IEEE Communications Magazine 53 (2) (2015) 98–106. doi:10.1109/MCOM.2015.7045397.

[28] W. Shen, M. Yoshida, T. Kawabata, K. Minato, 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, 2014, pp. 1–6. doi:10.1109/APNOMS.2014.6996522.

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

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

[31] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, 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, ACM, 2015, pp. 377–378.

[32] 5GEx consortium, 5GEx: 5G Exchange (2016).
URL http://www.5gex.eu/

[33] P. Gallo, K. Kosek-Szott, S. Szott, I. Tinnirello, Sdn@home: A method for controlling future wireless home networks, IEEE Communications Magazine 54 (5) (2016) 123–131. doi:10.1109/MCOM.2016.7470946.

[34] F. Sánchez, D. Brazewell, Tethered linux cpe for ip service delivery, in: Network Softwarization (NetSoft), 2015 1st IEEE Conference on, 2015, pp. 1–9.

[35] Docker, Docker for the enterprise.
URL https://www.docker.com/enterprise

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

[37] , Kubernetes.
URL https://kubernetes.io/

[38] R. Bonafiglia, S. Miano, S. Nuccio, F. Risso, A. Sapio, Enabling nfv services on resource-constrained cpes, in: 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), 2016, pp. 83–88. doi:10.1109/CloudNet.2016.24.

[39] M. Vásquez Bernal, I. Cerrato, F. Risso, D. Verbeiren, Transparent optimization of inter-virtual network function communication in open vswitch, in: 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), 2016, pp. 76–82. doi:10.1109/CloudNet.2016.26.

[40] L. Molnár, G. Pongrácz, G. Enyedi, K. Zoltán, L. Csikor, F. Juhász, A. Kőrösi, G. Rétvári, Dataplane specialization for high-performance openflow software switching, in: Proceedings of the 2016 ACM conference on SIGCOMM, 2016.

[41] S. V. Rossem, X. Cai, I. Cerrato, P. Danielsson, F. Nemeth, B. Pechenot, I. Pelle, F. Risso, S. Sharma, P. Skoldstrom, W. John, Nfv service dynamicity with a devops approach: Demonstrating zero-touch deployment & operations, in: IFIP/IEEE International Symposium on Integrated Network Management, 2017.

[42] Intel, DPDK (2015).
URL http://dpdk.org/

[43] Openconfig.
URL http://www.openconfig.net

[44] W. John, C. Meirosu, B. Pechenot, P. Skoldstrom, P. Kreuger, R. Steinert, Scalable software defined monitoring for service provider devops, in: 2015 Fourth European Workshop on Software Defined Networks, 2015, pp. 61–66. doi:10.1109/EWSDN.2015.62.

[45] P. Hintjens, ZeroMQ: Messaging for Many Applications, " O'Reilly Media, Inc.", 2013.

[46] UNIFY consortium, Deliverable 4.3: Updated concept and evaluation results for SP-DevOps (2016).
URL https://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables

[47] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, Extending networking into the virtualization layer, in: Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII), 2009.

[48] "BISDN", xDPd (2014).
URL http://www.xdpd.org

[49] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74. doi:10.1145/1355734.1355746.
URL `http://doi.acm.org/10.1145/1355734.1355746`

[50] ebpf.
URL `http://man7.org/linux/man-pages/man2/bpf.2.html`

[51] 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) (2014) 87–95.

[52] Google, Google cAdvisor.
URL `https://github.com/google/cadvisor`

[53] P. Kreuger, R. Steinert, Scalable in-network rate monitoring, in: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), 2015, pp. 866–869. doi:10.1109/INM.2015.7140396.

[54] W. John, G. Marchetto, F. Nemeth, P. Skoldstrom, R. Steinert, C. Meirosu, I. Papafili, K. Pentikousis, Service provider devops, IEEE Communications Magazine 55 (1) (2017) 204–211. doi:10.1109/MCOM.2017.1500803CM.

[55] M. Vásquez Bernal, I. Cerrato, F. Risso, D. Verbeiren, A transparent highway for inter-virtual network function communication with open vswitch, in: Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, 2016, pp. 603–604. doi:10.1145/2934872.2959068.

[56] Open Daylight.
URL `https://www.opendaylight.org/`

[57] ONLAB, Open Network Operating System.
URL `http://onosproject.org/`

[58] S. Miano, F. Risso, H. Woesner, Partial offloading of openflow rules on a traditional hardware switch asic, in: (to appear) Network Softwarization (NetSoft 2017), 2017 3rd IEEE Conference on.

[59] ETSI, Network Functions Virtualisation.
URL `http://www.etsi.org/technologies-clusters/technologies/nfv`

[60] SECURED consortium, SECURity at the network EDge (2013).
URL `http://www.secured-fp7.eu/`

[61] BISDN, Revised openflow library (2013).
URL `https://github.com/bisdn/rofl-common`

[62] S. V. Rossem, X. Cai, I. Cerrato, P. Danielsson, F. Nemeth, B. Pechenot, I. Pelle, F. Risso, S. Sharma, P. Skoldstrom, W. John, Nfv service dynamicity with a devops approach: Insights from a use-case realization, in: IFIP/IEEE International Symposium on Integrated Network Management, 2017.

[63] A. Manzalini, F. Risso, M. Ullio, Exploiting infrastructure capabilities to dynamically orchestrate nfv services across multiple domains, in: ETSI Workshop - From Research To Standardization, 2016.