Kathará: A Container-Based Framework for Implementing Network Function Virtualization and Software Defined Networks

Gaetano Bonofiglio*, Veronica Iovinella*, Gabriele Lospoto†, Giuseppe Di Battista†
Roma Tre University
* {gaetano.bonofiglio,vero.iovinella}@gmail.com † {lospoto,gdb}@dia.uniroma3.it

***Abstract*—Network Function Virtualization (NFV) and Software-Defined Networking (SDN) are deeply changing the networking field by introducing software at any level, aiming at decoupling the logic from the hardware. Together, they bring several benefits, mostly in terms of scalability and flexibility. Up to now, SDN has been used to support NFV from the routing and the architectural point of view.**

In this paper we present Kathará, a framework based on containers, that allows network operators to deploy Virtual Network Functions (VNFs) through the adoption of emerging data-plane programmable capabilities, such as P4-compliant switches. It also supports the coexistence of SDN and traditional routing protocols in order to set up arbitrarily complex networks. As a side effect, thanks to Kathará, we demonstrate that implementing NFV by means of specific-purpose equipment is feasible and it provides a gain in performance while preserving the benefits of NFV.

We measure the resource consumption of Kathará and we show that it performs better than frameworks that implement virtual networks using virtual machines by several orders of magnitude.

I. Introduction

Software is pervading several aspects of today's life. Networking is also involved in such a process and Software-Defined Networking (SDN) is the most representative example of how the routing can be managed by writing software possibly running on general-purpose hardware. In the last years, even the functions supporting the network (e.g. firewall or load balancing) have been softwarized by the Network Function Virtualization (NFV) architectural concept, leading to Virtual Network Functions (VNFs).

NFV greatly benefits from the advantages introduced by the SDN architecture since a central controller is able to determine where VNFs are located in the network. Indeed, SDN steers the traffic to cross the Service Function Chain (SFC) independently from the location of the VNFs themselves. Hence, in several cases, NFV follows the SDN architecture (e.g. [1] and [2]) where a logical centralized controller manages the whole set of VNFs. However, due to their software nature, VNFs run on general-purpose hardware or even on virtual machines (VMs), leading to poor performance compared to standard middleboxes where the same NFs natively run.

Besides SDN and NFV, there are two trends that are deeply changing how services and functions are deployed: data-plane programmability and containers. Network operators also have the opportunity to program the packet forwarding planes of the network devices. The current market leader in programming protocol independent packet processing is the P4 language [3]. This language can be compiled and executed on specific network equipment that is optimized for data forwarding [4]. Also, container technologies are rapidly replacing several services that were previously provided by means of VMs. A container allows software to run into a virtual environment that guarantees isolation of processes as in a VM with easier deployment and less overhead.

978-1-5386-3416-5/18/$31.00 © 2018 IEEE

However, even if NFV, SDN, P4, and containers are among the main players of today's networking scenario, as far as we know, there is no tool or framework that simultaneously exploits their benefits.

In this paper we present Kathará, a framework combining the efficiency of containers with the flexibility and portability of programmable control- and data-plane. Kathará can be used to create arbitrarily complex virtual networks that are efficient enough to be used in production scenarios and facilitates the transition of virtual network nodes to physical ones and vice-versa. This is made possible by exploiting the fact that both the P4 language and the OpenFlow protocol [5] are compatible with both software and hardware switches.

Kathará relies on containers, offering good performance since software running inside a container is comparable to software natively running on a system. Our framework has a simple high-level interface that allows the user to define network nodes and links between them. Such an interface consists of commands and/or pieces of configuration.

Furthermore, thanks to Kathará we are able to show that VNFs can be implemented mainly through the data-plane by using the P4 language. By doing so, packets are forwarded as from a standard network device, keeping unchanged the performance of such a device as well as the benefits of NFV.

The rest of the paper is organized as follows: in Sec. II we review the most relevant literature, comparing Kathará to existing systems to set up VNFs on virtual environments and technologies to deploy and connect multiple containers; in Sec. III we present the architecture of the Kathará framework, giving an explanation of the most relevant choices and an overview on its user interface; in Sec. IV we discuss several security aspects behind our framework that may arise from an undesired usage of the Docker platform; in Sec. V we focus on

two use cases showing how Kathará supports the implementation of NFV architecture using P4 and how to move network nodes from virtual environments to physical ones; in Sec. VI we show results in terms of Kathará resource consumption, start-up, and response time; finally, in Sec. VII we draw the conclusions, talk about possible Kathará enhancements, and discuss about open research perspectives.

II. Related Work

In this section, we review the state of the art, pointing out the main differences between Kathará and other frameworks for deploying virtual networks and for implementing architecture concepts such as SDN, NFV, and traditional routing.

To design Kathará, we started from [6]. In particular, we inherit its simple and well-defined user interface and we extend that tool by adding NFV support and programmable data-plane devices (e.g. P4 switches [7]). With respect to [6], our framework relies on containers instead of virtual machines, in such a way to reduce the set up and provisioning time, as well as memory and processing resources. Also, Kathará opens to the possibility of an easy migration of network nodes and functions from virtual to physical environments and vice-versa.

We chose Docker [8] as default container platform since it currently leads the market [9] and it is multi-platform, allowing Kathará to run on different operating systems with no limitations. Nevertheless, our framework can integrate other container platforms with no restrictions. Kathará implements functionalities for creating, configuring and connecting sets of containers, possibly starting from a configuration file.

Natively, Docker offers Docker Compose [10] as a tool to manage multiple containers. Despite its usefulness, it is mainly focused on services, offering a limited interaction with the networking configuration of the containers. In fact, Docker itself is not made to implement fully fledged networks, so we faced and solved several issues to arrange the networking properly. Because of this, Kathará currently uses many Docker primitives unaccessible from Docker Compose, which is also agnostic with respect to specific security issues (Sec. IV).

In [11], the authors highlight the most relevant challenges that should be addressed to manage network functions, as well as the big opportunities NFV opens. For example, it is recognized how difficult it is to manage NFs that are strongly integrated in the hardware. Indeed, updating one of them strictly implies to buy new hardware and the NFV architecture aims at overcoming such a limitation. Kathará can take care of that by enabling the deployment of VNFs independently from the hardware, which can be specific- or general-purpose.

In [12], the authors propose a system, called ClickOS, to provide NFs in lightweight middleboxes. ClickOS is built on top of virtual machines and the authors only focus on NFs. Kathará differs from such a system as it also provides advanced routing functionalities (e.g. control- and data-plane programmability) and it is based on containers.

In [13], the authors propose GLANF, a system to deploy VNFs in SDN networks. Kathará differs from that system because it is agnostic with respect to the underlying network architecture. It allows the interaction between standard routing and SDN protocols, without forcing a centralized positioning of VNFs in the controller premise.

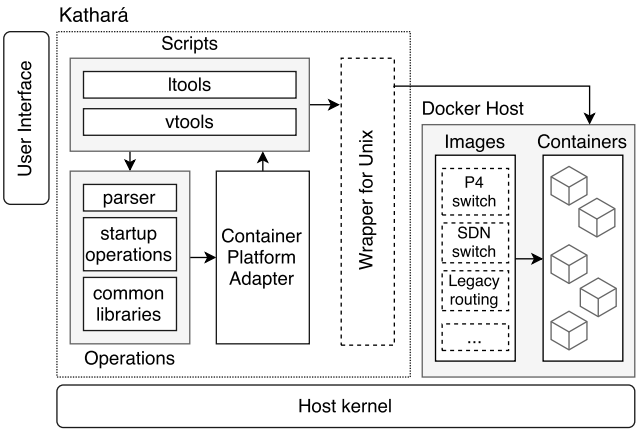


Fig. 1: Kathará Architecture

III. The Kathará Framework

In this section we present the Kathará framework. We discuss about the most relevant design choices we did in order to define its architecture, making it able to implement NFV and SDN capabilities in a container-based environment.

Architecture Overview. The Kathará architecture is depicted in Fig. 1 and consists of three main modules: *Scripts*, *Operations*, and *Container Platform Adapter*. Our framework fully relies on containers to set up the network. Indeed, we model each network node (e.g. IP-speaking router) as a container. As a consequence of such a choice, each network node acts depending on the software that runs on it. For example, to set up an IP-speaking router with a dynamic routing capability, Kathará creates a container and executes on it an instance of the Quagga routing protocol suite [14].

Containers represent a lightweight and multi-platform alternative to standard virtualization solutions, since they implement an isolated environment in which processes run independently, avoiding the overhead introduced by full virtualization that needs to maintain multiple kernels (one per virtual machine). Indeed, Kathará is built on top of the host kernel.

Kathará comes with a set of pre-built images that can be used to run several services, as shown in Fig. 1. We highlight that the images we provide are not ready-to-use VNFs. Rather, those images enable the network nodes to have different behaviours, such as:

1) standard routing protocols provided by the Quagga routing suite [14];
2) control-plane programmable network nodes based on the SDN protocol whose implementation is provided by OpenVSwitch (OVS) [15];
3) data-plane programmable switches provided by the Behavioral Model (BMv2) [7] implementation;
4) any application level service (e.g. software for DNS application, web server, DBMS, etc.).

Those services are executed on each container according to the source *image* which the container is derived from. An image is a package containing everything needed to run one or more specific services. Thus, *a container is a runtime instance of a specific image*. The images are usually retreived in two alternative ways. In the first one, the user builds its own images directly on the host from a configuration file; in the second one, the user pulls pre-built images from a remote repository.

Kathará allows users to specify the way in which containers are connected to each other by means of a piece of configuration. In that specification, the user states:

1) the list of all network nodes;
2) how those nodes are connected between them, resulting in a network topology;
3) the image for each network node (if it differs from the default one we provide);
4) a set of files that can be copied inside the network node. Those files contain specific pieces of configuration (e.g. for routing protocols);
5) a set of (optional) configuration parameters (e.g. the amount of memory to assign to each node, a list of commands to be executed inside the container at start-up time, the image to use for creating the node, etc.).

The users interact with Kathará through a set of scripts, that are implemented by the Scripts module. Those scripts implement basic operations that are used to start, stop, and pause the network nodes defined in the Kathará configuration. We divided the set of scripts in two categories:

1) `ltools`, including scripts acting on a collection of network nodes referring to the Kathará configuration;
2) `vtools`, including scripts acting on a single network node whose configuration is given as a list of parameters.

The Operations module includes a set of utility functions that are used by the Scripts module in order to allow Kathará to interpret the configuration provided by the users. Such a module is composed by three main logical components: parser, start-up operations, and common libraries.

The parser component translates the configuration, which can be provided through a command as list of parameters or through a configuration file, into a network topology. Such a topology consists in a list of network nodes, a list of links between those nodes, and a list of options obtained from the Kathará configuration (e.g. the image and the amount of memory assigned to each container). This module is also able to decide the start-up order of the nodes and the links according to specific dependency rules declared in the Kathará configuration. The start-up operations component is in charge of ensuring the cross-system compatibility, properly acting on the files that are copied inside the nodes. The common libraries component suitably produces a set of platform independent commands representing the actions to be executed. Such commands are generated according to the three lists (network nodes, links, and options) that the parser component handles.

The commands issued by the Operations module are provided to the Container Platform Adapter as input. The goal of this module is to make Kathará independent from any specific container platform. This results in the production of Docker commands that allow Kathará to execute operations, such as run the containers, create the network, and connect them. By substituting this module, Kathará can interact with different container platforms provided that they offer the same set of Docker primitives (in terms of container and network management). Still referring to Fig. 1, the output of the Container Platform Adapter comes back to the Scripts module, allowing Kathará to properly interact with a specific container implementation. For security reasons, if Kathará is executed on Unix systems, such commands have to be approved by a wrapper component, whose design is discussed in Sec. IV.

Once the commands are sent to the Docker daemon, network nodes and the corresponding links are created according to the Kathará configuration. Finally, terminal windows are (optionally) opened to allow the user to interact with the containers.

We argue that the Kathará architecture we showed is compliant with respect to the SFC Architecture [16] since it allows the user to implement chains of functions according to all the principles reported in RFC 7665. In particular, Kathará ensures topological independency and separation of components. This is due to its nature since our framework is agnostic with respect to the underlying topology and offers tools allowing the user to separately program control- and data-plane.

Why Containers. By design, Kathará relies on containers. We selected such a technology for several reasons. Containers are made possible by a set of facilities from the kernel, allowing lightweight partitioning of the host operating system into isolated spaces where processes can safely run. Their primary benefits are efficiency and agility. In those terms, containers are much more faster to provision (as shown in [17], [18]) and much simpler to build and define compared to native software [19] or traditional virtual machines. Since they do not require to run an hypervisor and multiple kernels, containers can be added, removed, paused, stored and redistributed in seconds, and even replicated on demand if needed, allowing smooth scaling and high availability.

Most service providers allow the quick deploy of a container by just uploading a text file (commonly known as *Dockerfile*) that can be built on the remote server in minutes, generating a container image that can be later used on a single server or in cloud environments. Such a file contains a set of instructions (e.g. to install specific software, to execute commands, etc.) that are used by the container engine to build an image.

Once the virtual machines, as well as the containers, finished the start-up process (ready-state), their performance is comparable in terms of response time and Input/Output procedures. However, containers are not memory hungry, as well as they do not require a full disk image, resulting in less overhead and space occupation. Hence, full replication of files is also avoided [18]. Container platforms such as Docker are also multi-platform, working on all commonly used operating systems by using the same interface.

User Interface. We now discuss the user interface of Kathará, focusing on the main commands and the basic pieces of

configuration used to deploy a set of network nodes (even a single one). As we reported in Sec. II, Kathará inherits its user interface from SDNetkit [6]. Such a choice may lead to a terminology overlap. For the sake of clarity, we now briefly present the user interface of our framework.

As reported at the beginning of the section, we group the commands into two suites: `vtools` and `ltools`. The `vtools` suite includes command acting on a single network node and all links which that node is connected to. The suite is mainly composed by commands to run, stop, remove, and inspect a single network node. The semantics of each command is described in the following.

By executing the `vstart` command, Kathará creates a single container representing a network node. By declaring several parameters, our framework is able to:

1) specify the image of the container (if it is different from the default one) and possibly limit the amount of memory the network node can use;
2) set specific configurations (e.g. assign a network interface of a network node to a link).
3) run general instructions at start-up time (e.g. assign an IP address to a network interface).

Once the network node and the corresponding links are created and connected, a terminal window shows up to allow the user to interact with the node.

We show two examples of how to start a network node with Kathará by using the command `vstart`.

```
test@kathara:~$ vstart --eth 0:A PC1
```

```
test@kathara:~$ vstart --eth 0:A \
                --eth 1:B --image=OVS SW1
```

By executing the first command, Kathará creates a single network node called *PC1* with one interface connected to the link named *A*. If not specified, Kathará uses the default image we provide. The second command results in the creation of a network node called *SW1* with two interfaces. One of them is connected on the link *A*, while the other one is linked to *B*. Note that, the link *A* already exists since it has been created by the first command. Thus, *PC1* and *SW1* are on the same collision domain. While creating *SW1*, Kathará looks for an image called *OVS* on the system to deploy the container, instead of using the default one. Kathará also allows the post-creation of new links by executing the command `vconfig`.

To stop a network node, Kathará offers two possibilities. By executing the `vhalt` command, that network node is gracefully stopped, while by using the `vcrash` it is forcefully stopped. By gracefully we mean that Kathará waits for all the processes inside the network node to be stopped, while by acting in a forceful way, Kathará instantly kills the network node and, accordingly, everything running inside it. Note that the instance of that node is still present in Kathará and it can be restored by using `vstart`.

By executing the `vclean` command, our framework stops the network node and removes it (also removing the instance from Kathará) and every link which that node is connected to, if no other network nodes are sharing the same link.

Finally, the `vlist` command allows the user to inspect useful statistics about network nodes created by using the command `vstart`, such as memory and CPU usage.

The `ltools` suite consists of a set of commands that allow a more automated management of multiple network nodes and links at once. To do that, Kathará relies on a configuration file. Such a configuration is mainly a list of *key-value* pairs, in which each pair has the form `name[option]=value`. From the following example configuration, `lstart` is able to create three network nodes called *service*, *host*, and *sw1* and two links *A* and *B*. The network node *service* is connected to *A*, while *sw1* is connected to *A* and *B*. Finally, *host* is connected to *B*. The network node *sw1* will also be a P4 switch instead of a legacy one (the default option).

An example of Kathará configuration file (lab.conf).

```
service[0]=A
sw1[0]=A
sw1[1]=B
sw1[image]=P4
host[0]=B
```

Basically, each command of the `ltools` suite has the same semantic described for the corresponding `vtools` suite, but due to the configuration, they are able to act on multiple network nodes and links at once. The Kathará configuration may also include start-up commands for each network node and folders that can be copied inside the container.

The interface we just described derives from our previous test environment, Netkit [6], [20], honed after years of practice and applications. As a side note, every other tool [21] and lab [22] previously designed for Netkit is completely portable to Kathará (the opposite is not true because of the additional capabilities of the new tool).

Extensibility. Kathará has been designed to be easily extended. Indeed, by relying on the Container Adapter Platform module, Docker can be replaced with any other equivalent container platform. To preserve the semantic of the Kathará user interface, only the implementation of a proper adapter towards the new container platform is required.

We provide several pre-built images allowing the user to run software inside the network nodes. At the moment of writing this paper, Kathará is able to manage network nodes running:

1) standard routing protocols (e.g. OSPF, BGP, etc.) and services (e.g. Apache, BIND, etc.);
2) SDN protocols enabling control-plane programmability (e.g. the OpenFlow protocol [5] and several SDN controllers, like the Ryu framework [23]);
3) data-plane programmable network switches compliant with the P4 language [3].

However, the architecture allows anyone to easily extend Kathará by providing additional images in a typical container fashion, namely by writing and building a specific Dockerfile.

IV. Kathará Security

In this section we discuss several security aspects related to Kathará. We also present the choices that we made in order to prevent that a malicious user exploits Kathará to enforce an undesired usage of the Docker platform.

As mentioned on the official Docker security page [24] under *Docker daemon attack surface*, only trusted users should be allowed to control the Docker daemon. This is a direct consequence of some powerful Docker features. Specifically, Docker gives root privileges inside containers and allows users to share or copy files and folders from the container to the host. This kind of attack can allow for privilege escalation (by copying or mounting a system folder or file, like `/etc/shadow`), or worse, a direct damage to the host.

From our perspective however, Kathará is not only a tool for system administrators, but also for practitioners who want to experiment with computer networks, often on shared machines. So one of our goals was to allow non-root users to run Kathará, without posing a threat to the host. Hence, we wrap Docker in order to solve this issue. The wrapper is modeled as a Finite-State Machine (FSM) that discards every command line that doesn't follow a specific allowed structure, accepting only what is safe to run for the host system. A simplified diagram for the FSM can be found here [25].

The FSM parses the commands and changes its internal state depending on each space-separated parameter. At the end of the line, if the machine is in an accepting state, the command is issued to Docker on a different shell with its own environment. The wrapper allows files to be copied only from the host machine to a container and it doesn't allow any folder to be mounted inside a container. It checks all the possible ways that those functions could be achieved since Docker allows folder mount and file copy with several commands.

The Kathará installer compiles the wrapper and than attributes to it the proper owner. It also sets the SUID bit appropriately and manages the permissions for every file of the framework. This allows the wrapper to execute the allowed Docker commands even if the user has no root privileges.

The security issue is so moved from the Docker daemon to our wrapper and the installer. Our installer has permissions so that it can only be executed by a system administrator (using rights for the minimum amount of time required to perform the installation) and we made sure that the wrapper is safe by checking every input and line of code for known attacks such as buffer overflow. The wrapper is also pretty small, amounting to just 170 lines of code that are very easy to check.

It is worth mentioning that this particular issue, and so the wrapper, exists only on Unix systems. In fact, Docker currently runs on Windows over a virtual machine on the Hyper-V hypervisor [26], and the system administrator can select which drives can be shared by the Docker daemon.

V. Use Cases

In this section we show several practical use cases for Kathará. We focus on how to use Kathará for setting up a fully virtualized NFV architecture and for implementing transitions of network nodes from virtual environments to physical ones.

Exploring Network Function Virtualization and P4. The NFV architecture concept is gaining the attention of both the scientific community and the industry because it allows practitioners to produce software-based versions of widely used NFs (e.g. firewall, packet counter, etc.). NFV brings several benefits, like the decoupling of NFs from the hardware, the flexible deployment and the dynamic scaling [27].

When the NF is decoupled from the hardware, those components are maintained independently, resulting in the possibility of updating and/or deploying the VNF without the need of changing the underlying hardware and vice-versa, also leading to more flexibility. Since software is instantiable, adopting the NFV architecture allows network operators to scale according to real-time requirements [27].

Initially, VNFs were mainly located inside data centers, allowing providers to easily manage and distribute those functions inside their existing network infrastructure. In contrast, the distributed NFV architecture aims at placing some of the NFs as close as possible to the final user (see [27], [28]).

Both centralized and distributed NFV architectures rely on general-purpose hardware, and frequently exploit the SDN architecture. This allows to achieve great flexibility at the expense of relatively poor performance [29] compared to switches that are able to process packets at wire speed [30]. Data-plane programmable switches have been introduced to provide data-plane flexibility, focusing on economical and commercial advantages, but also adding some interesting twists to the NFV scope. Data-plane programmable switches combine the programmability thanks to languages like P4 and wire speed packet process (see [4], [30]). For example, it is very easy to implement a firewall as a set of rules with the P4 language. We argue that data-plane programmable switches become a relevant solution for several distributed or even centralized approaches. Relying on this observation, it is very convenient to move NFs, such as firewall, Network Address Translation (NAT), Demilitarized Zone (DMZ), load balancer, web switch, and packet counter that are commonly known to be data-plane intensive, on a high performance programmable switch. Those NFs are suitable to be implemented as a set of rules acting on the packet headers. Note that, due to the presence of data-plane programmable switches, the architecture proposed in [31] is still supported. Indeed, using the terminology of [31], the control layer is assimilated in the forwarding plane, by implementing a proper classifier through the P4 language and the on-board controller.

By relying on this approach, we keep all the advantages of the NFV architecture (decoupling from the hardware, flexibility of deployment, and dynamic scaling) and increase the performance. In addition, the distributed VNFs represent a possible implementation of the SFC architecture, as discussed in Sec. III. Kathará can combine NFs through the provided P4 image, making possible the creation of arbitrary chains of service functions, according to [16].

Based on those observations, we now present two archi-

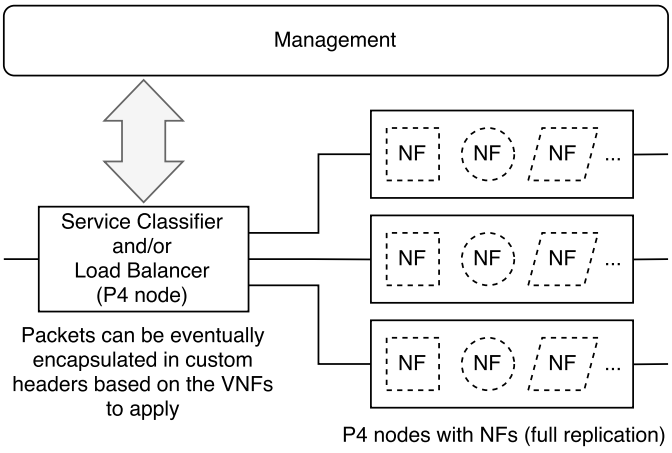
tectures (implementable with Kathará), depicted in Fig. 2, where all the network nodes (except the management) are P4-compliant switches. In the architecture of Fig. 2(a) the VNFs are replicated on every P4 switch, but for the one used as load balancer. Indeed, that switch is used to classify and/or direct each packet flow to one of the network nodes implementing NFs. Those nodes have the same set of VNFs that are applied to each packet according to the P4 program running on it.

In the second architecture, depicted in Fig. 2(b), we show an even more dynamic approach. The VNFs can be deployed as needed, according to the current traffic or demand. The drawback is that the first node on the left cannot be a simple load balancer, but it needs to keep complete flow tables to manage every packet and keep track of the NF to apply. However this is made easy by P4 because of the possibility to add data inside custom headers [32] that can later be parsed by other P4-compliant switches with the correct parser.

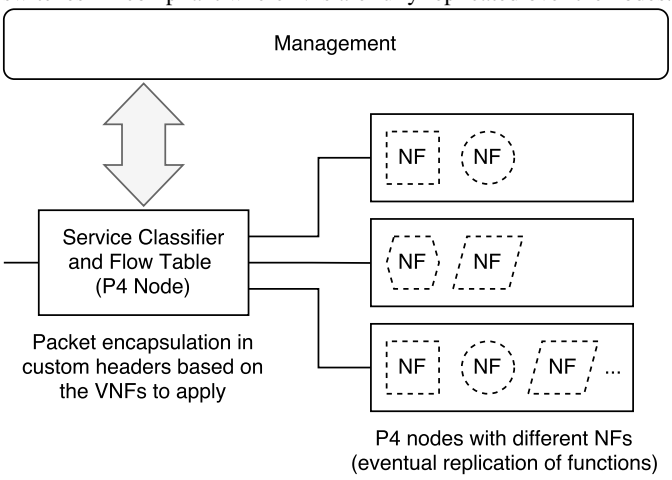
We remark that the architectures we just described are fully compliant with [16], since every architectural component reported in that RFC is present in our architecture. Note that, the SFC encapsulation is made easy by P4-compliant switches (see [3], [32]). Also, depending on our proposed architectures, the Service Function Path is implemented in a single network node (Fig. 2(a)) or in multiple ones (Fig. 2(b)). We point out that by using Kathará it is possible to implement common NFV architectures as reported in [33], but in this use case we focus on an architecture in which VNFs can be placed together in the same programmable switch (often referred as *whitebox*). We argue that, depending on the service to be issued, grouping VNFs may bring several benefits, such as faster and easier packet forwarding and less network equipment.

By using Kathará we implemented P4-compliant switches and tested a management system that allows for remote compiling of P4 code and continuous delivery of service, updating each network node after the redirection of every packet flow towards the available ones. It is worth mentioning that some physical P4-compliant switches may implement a system that allows updates without packet loss by using buffers.

Transferring Network Nodes from Virtual to Physical Equipment and Vice-Versa. According to Sec. III, Kathará allows the user to create virtual networks and to run software inside each network node in order to set up arbitrarily complex networks. On one hand, Quagga is a well known and stable routing protocol suite implementing many standard routing protocols (e.g. OSPF). On the other hand, OVS is a widely adopted solution to realize SDN-based services, as shown in [34]. Even if BMv2 is just a P4 programmable software switch which is not used in production networks, every P4 program compiled for BMv2 can be also compiled on every physical P4-compliant switch. As a consequence, Kathará *opens the possibility to manage transitions of virtual network nodes towards physical ones and vice-versa*, allowing network operators to move services from specific- to general-purpose hardware (and vice-versa). Thus, it leads to easily implement distributed NFV architectures also involving advanced routing functionalities, such as both programmable control-plane (e.g. SDN) and data-plane (e.g. P4).



(a) NFV architecture implemented by using data-plane programmable switches P4-compliant where NFs are fully replicated over the nodes.



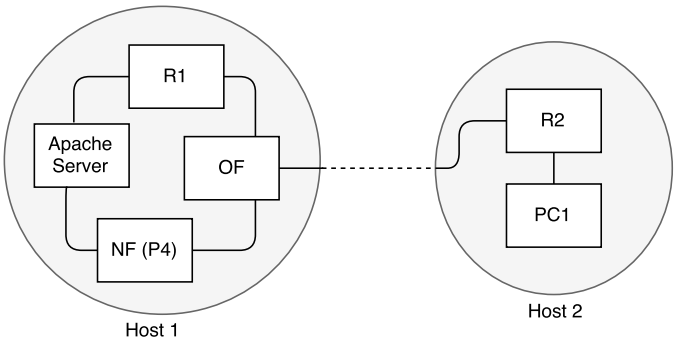
(b) NFV architecture implemented by using data-plane programmable switches P4-compliant where NFs can or cannot be replicated over the nodes.

Fig. 2: NFV architectures implemented through the adoption of programmable data-plane switches.

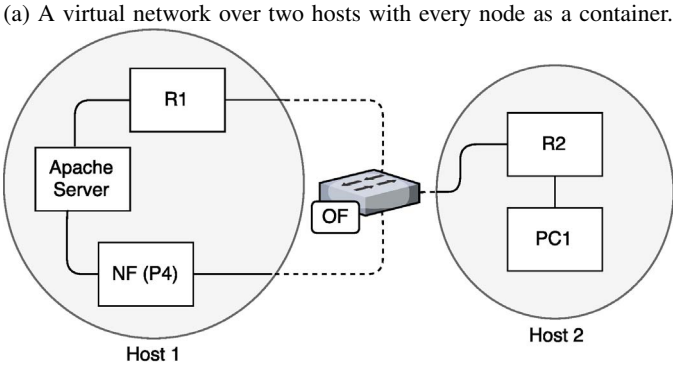
Note that, as already discussed in Sec. III, Kathará can be easily extended with additional services by adding suitable user created images. This allows the transition of any service that can run on a container.

An example of this use case is depicted in Fig. 3. We have that Kathará is running on two physical machines (Host 1 and Host 2 in the figure) and each of them, in turn, runs network nodes implementing different network functions and/or services. R1 and R2 are IP-speaking routers running Quagga, while OF is an SDN-enabled switch running OVS. NF is a node representing a P4 implementation of a generic network function (e.g. a packet classifier, a packet counter, etc.). PC1 represents a standard host, while Apache Server is a node running an instance of the Apache web-server.

Fig. 3(a) shows a scenario in which Kathará manages all



(a) A virtual network over two hosts with every node as a container.



(b) The same network with one node moved from the virtual environment to a physical network device.

Fig. 3: Transition of a network element from a virtual environment to a physical one.

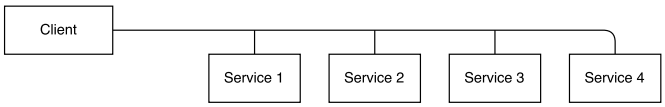
the network nodes. Since the virtual network components have the same operational features of those running on a specific-purpose hardware, it is easy to move nodes from a container running inside a machine to bare metal. Fig. 3(b) shows the outcome of the transition in which an SDN node has been moved from a container to a dedicated network switch.

Consider that, since containers do not introduce much overhead compared to natively running software, sometimes it is possible to move services from physical devices to containers running on general-purpose hardware. Even in this case Kathará can be used to implement a smooth transition between the two scenarios.

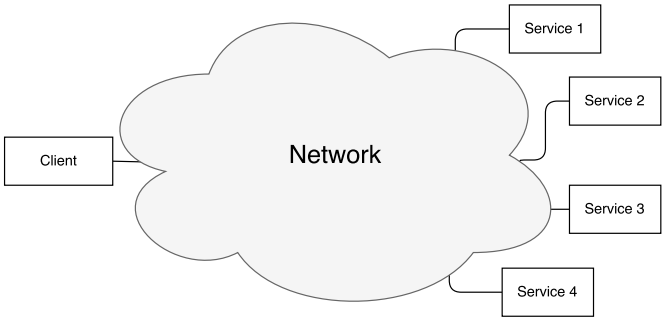
By means of the above use cases, we showed how Kathará can be used to create arbitrarily complex NFV architectures, also opening the possibility to use data-plane programmable P4-compliant switches to implement VNFs. Since the P4 language is made to run on bare metal, this makes very easy to move virtual network nodes from inside Kathará to physical switches just by compiling the same P4 code. Our use cases show that it is possible to support the same transition when SDN and traditional switches are used (as we said in Sec. III).

VI. Evaluation

To assess the effectiveness of Kathará, we focus on measuring its performance with respect to the amount of time to deploy a single network node, memory, and CPU compared



(a) Scenario in which client and services are directly connected.



(b) Scenario in which client and services are connected to an arbitrarily complex network.

Fig. 4: The testbed for Kathará evaluation.

with a solution based on virtual machines. Also, we measure the response time of services in a network where they are directly connected to a client opposed to the same scenario in which client and services are separated by an arbitrarily complex network consisting of several nodes. By measuring the amount of network nodes that can run on a single machine and the resource consumption of single nodes, we can show how our system allows for better scalability and deploy compared to previous solutions based on VMs. We release the Kathará source code and the testbed we used at [35].

Testbed and Scalability. We run Kathará on a virtual machine equipped with Ubuntu. We assigned 3 GBytes of RAM and 4 cores at 2.21 GHz to that virtual machine. We set up two different scenarios, depicted in Fig. 4. In particular, Fig. 4(a) shows a network in which the client is directly connected to a set of services, that is a typical way in which container-based services are exposed. Fig. 4(b) shows a scenario that is equivalent in terms of service, but where such services are located at the edge of a network. The scenario is representative of a typical topology where NFs (e.g. firewall, load balancer, packet counter, etc.) are decoupled from the services.

As a scalability evaluation, we run a growing number of virtual machines and containers. We observed that the maximum number of running VMs based on User Mode Linux was 40, whereas we successfully run 300 network nodes using containers. Such results prove that Kathará scales better by one order of magnitude using the default Docker settings.

Performance Tests. In terms of performance, we focus on start-up time and resource consumption. Our results are depicted in Fig. 5. As expected, according to those metrics, we observe that container-based virtualization outperforms User Mode Linux (UML) virtualization by at least one order of magnitude. We recall that UML is a kernel implementation running in user-space, thus it does not require root privileges.

We measure the start-up time by creating a single network node with the same network capabilities. In that case, Kathará

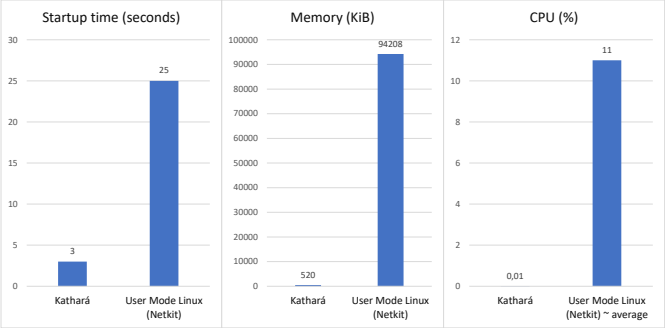


Fig. 5: Comparison between Kathará and virtual machine-based solution in terms of deploy time and resource consumption.

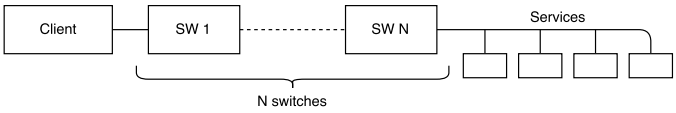


Fig. 6: Actual implementation of the testbed.

is shown to be almost 10 times faster, taking less than 3 seconds as opposed to 25 seconds average spent by a virtual machine based solution. We ascribe this behaviour to the fact that containers do not need to load any kernel.

In terms of resource consumption, we measure the amount of CPU and memory used by a single network node, implemented with a container and a UML virtual machine running the same software. Still referring to Fig. 5, we observe that the amount of memory is around 520 KiB in case of container virtualization, while in case of UML a single machine occupies roughly 90 MiB of memory, three orders of magnitude more.

Finally, we measure the amount of CPU consumed by the two systems on a single core during a ping. As shown in the picture, a container-based network node reaches at most 0.01% of the CPU, while a UML solution ranges from 3% up to 65%. For readability, in the picture we just report the average value of 11% measured across the network nodes of the testbed that were running on the same machine. However memory was our primary bottleneck for the number of network nodes that could be executed at the same time with both systems.

Response Time. During this experiment, we measure the time spent by a service to issue a response to a client. We simulate the service as a small web application querying a MySQL database. Such an application runs inside an Apache web-server and it is implemented using the PHP language. We argue that such a service setting is realistic and widely adopted to support many web applications, as shown in [36].

We implemented the two scenarios depicted in Fig. 4 by creating a lab in Kathará following the topology shown in Fig. 6. We modelled the network as a path with a growing number of switches (labelled as *SW 1*, *SW 2*, etc., *SW N*) that simulates an arbitrarily complex network connecting the client to the services. We vary the length of the path in a range $[0, 60]$ and we perform requests by querying the service from

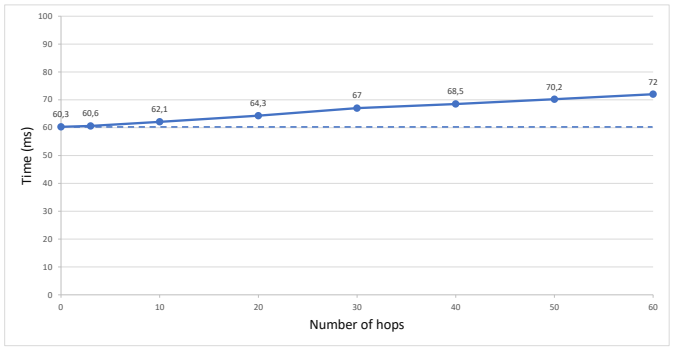


Fig. 7: Response time of a complex service in presence of an variable number of hops.

different positions in that path.

The results of the response time are depicted in Fig. 7. We observe a linear growth of the time with respect to the number of hops in the network. However, the additional latency is negligible compared to the time spent by the service to issue the response (order of 10-100 milliseconds). Indeed, each hop introduces a delay around 0.3 milliseconds on the virtual machine that we used to carry out our experiments. We point out that we used simple nodes with static routes and P4-compliant software switches (BMv2 with the P4 program l2_switch.p4 running on it), provided by the Kathará images we presented in Sec. III. We observed that using BMv2 software switches does not introduce more latency in the network compared to standard L3 IP-speaking routers.

As a general consideration, we argue that the service efficiency is almost not affected by the network size. Also, considering the results we show in Figs. 5 and 7, containers have good enough performance to allow parts of the production networks to run as virtualized.

VII. Conclusions and Future Work

VNFs and SDN represent both challenges and opportunities in today networks. Their adoption allows providers to issue novel services and to better exploit their infrastructure. The benefits range from programmability to flexibility and scalability. Also, the growing usage of container technologies is due to the fact that they reduce the provisioning time with respect to VMs and it is easy to find pre-built images to run a wide variety of services.

In this paper we presented Kathará, a container-based framework to deploy VNFs, SDN services, and standard routing protocols working together. We showed how the technologies we use (e.g. P4 and OVS) allow for an easy transition between virtual and physical network equipment. Also, we propose architectures to implement VNFs in the data-plane. Finally, we show how Kathará performs better with respect to VM-based solutions in terms of resource consumption and time.

In the future Kathará can be expanded to take advantage of Docker Swarm to deploy network nodes in the cloud, or add support for orchestration in order to provide more service-oriented features.

REFERENCES

[1] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 511–524. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934875

[2] J. Matias, J. Garay, N. Toledo, J. Unzilla, and E. Jacob, "Toward an sdn-enabled nfv architecture," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 187–193, 2015.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[4] "The world's fastest & most programmable networks," Sept 2017. [Online]. Available: https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[6] H. Mostafaei, G. Lospoto, R. di Lallo, M. Rimondini, and G. Di Battista, "Sdnetkit: A testbed for experimenting sdn in multi-domain networks," in *Proceedings of the 3rd IEEE Conference on Network Softwarization (IEEE NetSoft 2017)*, 2017.

[7] "Behavioral model," Sept 2017. [Online]. Available: https://github.com/p4lang/behavioral-model

[8] "Docker software container platform," Sept 2017. [Online]. Available: https://www.docker.com

[9] T. Combe, A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sept 2016.

[10] "Docker compose," Sept 2017. [Online]. Available: https://docs.docker.com/compose/

[11] K. Lu, S. Liu, F. Feisullin, M. Ersue, and Y. Cheng, "Network function virtualization: opportunities and challenges [guest editorial]," *IEEE Network*, vol. 29, no. 3, pp. 4–5, May 2015.

[12] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

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

[14] "Quagga routing suite," Sept 2017. [Online]. Available: http://www.nongnu.org/quagga/

[15] "Openvswitch," Sept 2017. [Online]. Available: http://openvswitch.org/

[16] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7665.txt

[17] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, pp. 105–111, 2014.

[18] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 25, p. 31.

[19] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[20] M. Pizzonia and M. Rimondini, "Netkit: easy emulation of complex networks on inexpensive hardware," in *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, p. 7.

[21] "Netkit tools from contributions," Sept 2017. [Online]. Available: http://wiki.netkit.org/index.php/Download_Contributions

[22] "Netkit official labs," Sept 2017. [Online]. Available: http://wiki.netkit.org/index.php/Labs_Official

[23] "Ryu: component-based software defined networking framework," Sept 2017. [Online]. Available: https://osrg.github.io/ryu/

[24] "Docker security," https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface, Sept 2017.

[25] "Simplified fsm for the wrapper of kathará," https://github.com/Kidel/Kathara/blob/master/bin/wrapper/fsm.pdf, Sept 2017.

[26] "Microsoft hyper-v," Sept 2017. [Online]. Available: https://docs.docker.com/machine/drivers/hyper-v/

[27] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[28] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, Feb 2015.

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

[30] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[31] "L4-l7 service function chaining solution architecture." Sept 2017. [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/L4-L7_Service_Function_Chaining_Solution_Architecture.pdf

[32] "P4 language specification," Sept 2017. [Online]. Available: https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[33] "What is network service chaining? definition." Sept 2017. [Online]. Available: https://www.sdxcentral.com/sdn/network-virtualization/definitions/what-is-network-service-chaining/

[34] R. di Lallo, M. Gradillo, G. Lospoto, C. Pisa, and M. Rimondini, "On the practical applicability of SDN research," in *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, M. Erol-Kantarci, B. Jennings, and H. Reiser, Eds., 2016, pp. 1–9.

[35] "Kathará repository on github," Sept 2017. [Online]. Available: https://github.com/Kidel/kathara

[36] "Server-side programming languages market position report," Sept 2017. [Online]. Available: https://w3techs.com/technologies/market/programming_language