Received: 12 September 2016 | Revised: 4 October 2018 | Accepted: 15 March 2019

DOI: 10.1002/dac.3961

RESEARCH ARTICLE

WILEY

A virtualized infrastructure to offer network mapping functionality in SDN networks

Pilar Manzanares-Lopez | Juan Pedro Muñoz-Gea | Josemaria Malgosa-Sanahuja | Adrian Flores-de la Cruz

Department of Information Technologies and Communications, Technical University of Cartagena, Cartagena, Spain

Correspondence
Pilar Manzanares-Lopez, Department of Information Technologies and Communications, Technical University of Cartagena, Antiguo Cuartel de Antigones Campus Muralla del Mar s/n, Cartagena E-30202, Spain.
Email: pilar.manzanares@upct.es

Funding information
Ministerio de Economía y Competitividad, Grant/Award Number: TEC2003-47016-C2-2R and TEC2016-76465-C2-1R

Summary

The separation of control and forwarding planes in software-defined networking (SDN) networks is a key issue of the SDN technology. This feature and the existence of the SDN controller allow the developing of dynamic, adaptable and manageable networks, networks that require adequate services, and applications. However, the separation of these planes prevents the use of existing powerful tools that were coded considering traditional networks. In this paper, we make use of the potential of network virtualization (NV) technologies to propose the use of a virtualized infrastructure that makes possible the incorporation of these existing services and/or applications to an SDN network, without the need for programming additional and complex software modules in the SDN controller. Thus, in this paper, NV is not employed to develop a network managed by SDN but to broaden and give support to the SDN control layer. As an example, we describe the incorporation of *nmap* (a versatile and powerful tool widely used by security experts for network exploration) into the SDN framework. It is only necessary to develop a simple control plane service that thanks to the proposed virtualized infrastructure allows the inclusion of this powerful management application. The result offers the complete functionality of the *nmap* utility to the network administrators, who control the SDN network through the out-of-band control plane. In addition, a northbound REST API has been defined to offer the main functionality of the tool (host discovery, port scanning, and operating system detection) to the application layer.

KEYWORDS

network mapping, network virtualization, OpenFlow, SDN

1 | INTRODUCTION

Software-defined networking (SDN) and network virtualization (NV) have changed the traditional networking paradigms. Both are different but are strongly related. SDN can be seen as a programmable and customizable framework that allows the integration and management of networks, regardless of whether they are physical or virtual networks.

The SDN controller is the core of the network acting as the control point of the SDN network. Making use of the adequate software modules, the controller will be able of managing dynamic, adaptable, and efficient networks. However,

the deployment of new and advanced networking applications that improve the network performance and functionality involves the programming of additional and complex modules.

As in the case of traditional network administrators, SDN-network administrators require adequate utilities to solve the variety of networking tasks. They include examples such as bandwidth monitoring, diagnostic tests, real time analytics, or troubleshooting network problems. The logically centralized control offered by the SDN controllers, by means of different software modules, provides networking related information to give a solution to most of these goals.

From a management perspective, the accessibility to the SDN network from the controller is obtained by the out-of-band (OOB) control plane connection through the OOB port of the network elements (routers and switches). OpenFlow protocol[1] is used for this purpose. In contrast, hosts are not accessible through the OOB. Thus, it is not possible to make use of any network tool that is performed using end hosts. Active measurement tools such as *ping* and *traceroute* or network mapping tools such as *nmap* are some examples.

Using the proposal we described in this work, it will not be necessary to implement the whole management functionality of any of these tools as a new module in an SDN controller. We make use of the potential of NV technologies to propose the creation and use of a virtualized infrastructure that allows the incorporation of these and other useful network services and applications offered by existing and well-known software tools into an SDN network, without the need for programming additional and complex software modules in the SDN controller.

NV is not used to develop the network managed by SDN but to broaden the SDN framework. Using this virtualized infrastructure, it will be easy to offer to network administrators existing software tools for network management or even to integrate these applications in more complex network management tools.

In particular, by the way of example, this work describes the incorporation of the existing *nmap* tool[2] in the SDN networks, without the need of implementing it from the scratch as a controller module. Using our proposal, the required SDN controller module will not be in charge of building the management packets, sending and receiving them, and proceeding according to the management protocol state machine. Instead, it is only necessary to code a simple control plane service that thanks to the developed virtualized infrastructure allows the inclusion of this powerful management application.

While *nmap* is commonly used for security audits, many systems and network administrators find it useful for routine tasks such as network inventory, managing service, upgrade schedules, and monitoring host or service uptime. Our solution allows network administrators and application developers to make use of the complete functionality of the command line *nmap* utility. In addition, a REST API has been defined to facilitate the communication between the agents that require network mapping information and the SDN controller.

Although the proposed network architecture has been employed to integrate the use of a particular software tool, the potential of this solution is huge. The virtual infrastructure would allow developers to define more complex scenarios where different SDN controllers and services can coexist. In this paper, the SDN network is managed by the Ryu controller,[3] exposing a northbound REST API and also offering the complete functionality of the *nmap* tool.

The rest of this paper is organized as follows. Section 2 describes the network mapping functionality and outlines SDN and NV. In Section 3, we present some related works. Section 4 describes the virtualized infrastructure proposed in this paper, the required Ryu software module, and the northbound APIs that have been defined. Section 5 presents some situations where network mapping functionality is useful to improve the network management. Finally, Section 6 offers some conclusions.

2 | BACKGROUND

2.1 | Network mapping

Network mapping is a fundamental task to help network administrators to "know" the network infrastructure that they are controlling. There are different processes involved in gathering information about computer networks, such as host discovery, port scanning, service detection, and operating system detection.[4]

Ping utility using the ICMP protocol[5] is the most commonly used method for discovering which hosts are located in a network, checking a range of IP addresses. ARP protocol[6] is an effective way of detecting hosts in most IEEE 802 network technologies, including Ethernet and Wi-Fi. On the other hand, nodes in a network often have one or more names or aliases associated with them. In order to translate these names into IP addresses, Domain Name System (DNS)[7,8] servers exist. The port scanning objective is to find out which TCP/UDP ports are open, closed, or filtered. It is conducted by simply trying to establish a socket connection with a node on a particular port. Service detection is the process of determining which service and version is running on a specific port. After the open ports are discovered, version detection interrogates

those ports to determine more about what is actually running. Operating system detection is accomplished in a similar way as service detection. The remote host is interrogated and, after performing multiple tests and analysing the responses, the OS executed on the node is identified by its behaviour.

2.2 | Overview of SDN

SDN is based on the separation between the control plane and the data plane. The network elements only perform packet forwarding (this is called the data plane). The decisions on how packets should be forwarded by the network devices and the pushing of such decisions down to the network for execution are the responsibility of the control plane.[9] The intelligence of the network is logically centralized in SDN controllers, which contains a collection of pluggable modules that perform different network tasks.

The SDN controller (control plane) communicates with the routers and switches (data plane) using a southbound interface (SBI). OpenFlow is probably the most-known SBI protocol. It offers a flexible, dynamic, and programmable interface to manage network elements from a logically centralized location.

On the other hand, the SDN controller interacts with applications and business logic (application plane) using a northbound interface (NBI). The NBI should be a clearly defined interface by which applications can access the underlying devices, coexist and interact with other applications, and use system services (eg, topology discovery and forwarding) without requiring the application developer to know the implementation details of the controller.[10] Although there have been some attempts to standardize a NBI for SDN, such as the Open Daylight Project,[11] a standard solution does not exist yet. Currently, different SDN controller solutions have proprietary NB APIs.

On that point, one of the common technologies used to define the NB APIs is REST.[12] REST APIs use the HTTP protocol to execute common operations on resources represented by URI strings. An application can use the REST APIs to send HTTP messages via the SDN controller's IP address. The messages would contain a URI string referencing the relevant network device, the HTTP method, and a JSON/XML payload and/or parameters.

2.3 | Network virtualization

Over the last years, NV has become a promising area for developing modern network technologies.[13] NV offers the ability to integrate multiple hardware and software networking resources into a logical software-based virtual network entity that can be easily consolidated and efficiently managed using commercial computing servers, network switching gears, and off-the-self equipment.[14] NV allows IT managers to consolidate multiple physical networks, divide a network into multiple segments, or create software-only networks between virtual machines.

To solve the NV objectives, different virtualization technologies are available. Layer 2 switching is typically implemented by means of kernel level virtual bridges or switches interconnecting virtual and physical interfaces. Open vSwitch (OVS)[15] is an open source and software-defined virtual multilayer switch designed to enable network automation through standard management interfaces and protocols. It has been available in the Linux kernel since version 3.3, and it is widely used within the SDN developer communities. Layer 3 routing functions can be executed by taking advantage of lightweight virtualization tools, such as Linux network namespaces.[16] Network namespaces provide the isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, routing tables, firewalls, port numbers, and so on.

3 | RELATED WORK

A wide range of network functionality is not standardized on SDN controllers, so each controller should define and implement complete software modules to solve the required tasks.

In some cases, it will be enough for the controller to interact with the network elements through the standard OpenFlow protocol, requesting some standardized information (port status, transmission/reception statistics, etc). In other cases, the resolution of the task may demand that the controller generates network traffic proactively. In these cases, the software module will require that some packets are created from the scratch. By means of the OpenFlow protocol, the switches will be instructed to how forward these packets (using PacketOut messages) and how process and forward them when received from other switches according to the adequate flow rules (that will be set using FlowMod messages). The controller will address the non-standardized network functionality by the processing the information obtained from the switches by

means of the Packet-In messages. For example, in a previous work,[17] we have implemented and evaluated a new module that offers an improved host discovery functionality.

An alternative approach has been described in this paper. The main idea is to facilitate the reuse of existing tools to make easier the integration of their functionality in SDN networks. With a similar approach but with a different proposal, SDN-Radar[18] thinks about the possibility of reusing existing network management tools in an SDN network. Their aim is to help network administrators to determine the location and type of a network fault. They propose an architecture composed of the SDN controller and SDN network devices, and users called agents that are connected to switches in the network and that will be in charge of executing the adequate network management tool. The work of the agents is coordinated by an agent controller, which also collects the user reports. Finally, it is required an application that takes into account the measurements required by the agents and other features provided by the SDN controller to identify problems in the network.

SDNIPS[19] is another solution that makes use of an existing tool to enhance offered functionality of SDN networks. They utilize Snort, a multimode packet analysis tool dominating the IDS/IPS market, and the flexibility offered by the network reconfiguration of SDN networks to present an SDN-based IPS solution.

4 | PROPOSED SOLUTION

The usual way of adding new network functionality to an SDN controller is by means of coding new modules. In this way, new modules should be coded from scratch to implement the network mapping tasks. However, there are different software tools, such as *nmap*,[2] that already offer this functionality. *Nmap* runs on all major operating systems, and official binary packages are available for Linux, Windows, and Mac OS X.

In this paper, we propose an alternative approach that clearly reduces the complexity assigned to the SDN controller. The solution consists in the development of a virtualized architecture that allows the use of the existing network mapping utilities. Thus, the *nmap* tool will be the process in charge of creating and processing all the required TCP/IP packets to offer the network mapping services.

If this approach is compared with the purely controller-based SDN solution, the number of involved packets that are generated to execute any of the nmap functionality is the same. In our proposal, the packets are created directly by the *nmap* process, while in the second case, the packets must be created by the controller.

However, the implications in terms of controller performance if the network mapping functionality were implemented as a new module would be harder, because the developed module should be in charge of sending and receiving management packets and also for building them according to the management protocol state machine.

As described in the following sections, our solution offers two levels of interaction with the *nmap* tool. On the one hand, network administrators and application developers can make use of the complete functionality of the command line *nmap* utility (the most important commands of *nmap* are shown in Table 1). On the other hand, a northbound REST API has been coded providing the most interesting *nmap* options.

4.1 | Required virtualized infrastructure

Nmap process creates raw TCP/IP packets, sends them to the target network, and processes the responses to carry out the different network mapping tasks. Therefore, there needs to be a communication between the computer where the *nmap* application is executed and the target network, in this case, the OpenFlow-based SDN network.

However, in SDN networks, the controller has only access to the management port of the switches and routers through an out-of-band network (OOB). Therefore, from a management perspective, the target network, and consequently the hosts, is not accessible from the SDN controller.

Thanks to the proposed virtualized infrastructure, this problem is solved and the complete functionality of the *nmap* utility is offered to the network administrators, who control the SDN network through the OOB control plane. The key idea is that the SDN controller benefits from the *nmap* tool, avoiding having to create all the TCP/IP traffic required to carry out the network mapping tasks. *Nmap* will be executed at the controller as usual and the virtualized infrastructure will enable the communication between the *nmap* process and the target network using the OpenFlow interface.

Figure 1 shows the virtualized network infrastructure designed to implement the proposal. Although we consider only the use of a particular software tool (*nmap*), the potential of this solution is huge. The virtual network allows defining different scenarios in which different SDN controllers and services can coexist by means of the use of different namespaces.

TABLE 1 Main *nmap* commands

Host discovery	
nmap -sn 192.168.1.0/24	ICMP Echo Request, ICMP Timestamp Request, TCP SYN to port 443, TCP ACK to port 80
nmap -PO 192.168.1.0/24	Only ICMP ping
nmap -PS 192.168.1.0/24	TCP SYN (if ICMP pings are blocked)
nmap -PA 192.168.1.0/24	TCP ACK (if ICMP pings are blocked)
nmap -PU 192.168.1.0/24	UDP ping
Port scanning	
nmap -p 22 192.168.1.1	scan a single port
nmap -p 1-100 192.168.1.1	scan a range of ports
nmap -F 192.168.1.1	100 most common ports
nmap -p T:80 192.168.1.1	scan a TCP port
nmap -p U:80 192.168.1.1	scan a UDP port
Alias resolution (reverse DNS resolution)	
nmap -sL 192.168.1.0/24 [–dns-servers=server]	
OS detection	
nmap -O 192.168.1.1	Detect remote Operating System
Service detection	
nmap -sV 192.168.1.1	Detect remote Services
If a host/network is protected by a firewall	
nmap -sA 192.168.1.254	Find out if a host/network is protected by a firewall
nmap -PN 192.168.1.254	Scan a host when protected by the firewall

Note. In this table, the target network is 192.168.1.0/24.

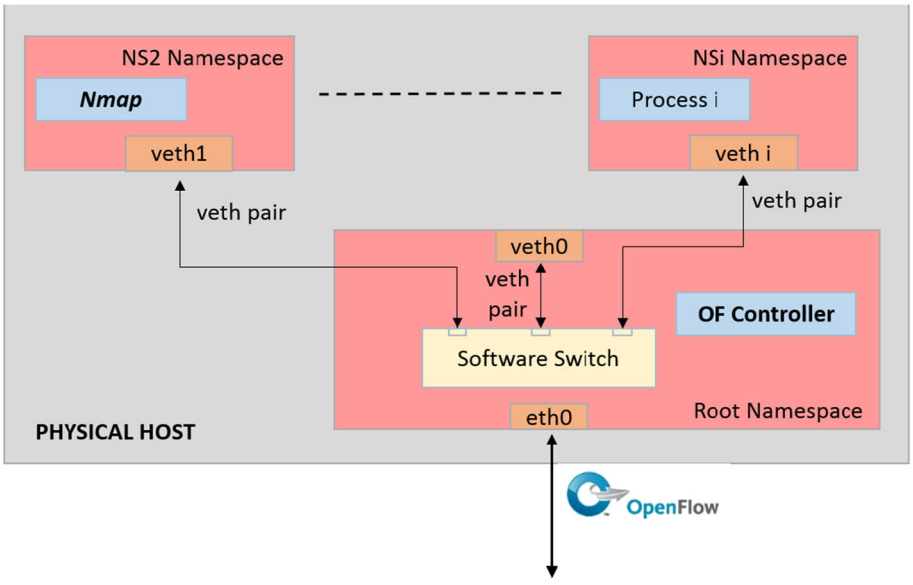

FIGURE 1 Required virtualized network infrastructure

In this case, to allow the controller to receive the packets generated by the *nmap* process, two independent networking instances are used. The key point is to allow the controller to "intercept" the packets which are generated by *nmap* and send them to the SDN network to perform all the network mapping tasks efficiently. The SDN controller will be executed in the default namespace (the root namespace in Figure 1) and the *nmap* application will be executed in a new network namespace (NS2 in Figure 1). As said before, each network namespace maintains its own ARP table and routing table. Both networking instances are interconnected through a virtual network consisting of a virtual bridge (OVS[15] has been used) and two virtual Ethernet (veth) links.

The commands to implement and configure the virtual infrastructure in a Linux machine are detailed in Appendix A.

4.2 | Ryu software components

The Ryu controller provides software components, commonly known as modules, with well-defined application program interfaces (APIs), that make it easy for developers to create new network management and control applications. Table 2 shows the main modules providing REST APIs in Ryu 4.4, and as an example, Table 3 shows the northbound REST APIs offered by the *rest_conf_switch.py* module.

4.2.1 | New controller module: sdn-nmap.py

In this paper, we have developed a new software module for the Ryu controller which, thanks to the previously described network infrastructure, provides the complete functionality of the *nmap* tool to network administrators and application developers and also offers a set of REST APIs offering the most interesting *nmap* options. We have called it *sdn-nmap.py*.

The key element of *sdn-map.py* is the use of raw layer 2 sockets. If a raw layer 2 socket is created, packets bypass the normal TCP/IP processing and are passed directly to the specific user application. That is, the application will receive all the Ethernet frames.

The implemented module creates a raw socket to listen to the virtual interface veth0. As detailed in Appendix A, the interface veth0 is the default gateway of the namespace ns2, so any packet created by the *nmap* tool will be received by the controller. Depending on the *nmap* command, the received packets will be ICMP, TCP, or UDP packets.

When ping-based host discovery commands are executed, ICMP Echo Requests (type 8) and ICMP Timestamp Requests (type 13) are created by the *nmap* application. The *sdn-nmap.py* module will receive the messages and, to allow the discovery of any host, will generate *packet_out* messages containing the received Ethernet frames. The *packet_out* messages will be sent directly to each switch in the network, indicating the action of forwarding the encapsulated frame through each port.

Notice that, as said before, the TCP/IP traffic is not created by the controller but by the *nmap* utility. Due to the virtualized network infrastructure, the ICMP requests are encapsulated in an Ethernet frame with veth1's mac address as source address and veth0's mac address as destination. For that reason, to allow the existing hosts of the SDN target network to receive and process the queries, the destination address must be changed to the broadcast address by the controller, before generating the *packet_out* messages.

Each existing host will respond to adequate ICMP requests with the corresponding ICMP replies. The replies will be received by the leaf switch, which will perform a matching lookup in its flow table. So that the *sdn-nmap.py* module will not depend on table-miss flow entries, a flow entry is proactively inserted in each switch. The flow entry will match all the packets whose destination's IP address is equal to veth1's IP address and set the instructions for sending to the controller as a *packet_in* message. In addition, another flow entry is proactively inserted in each switch. In this case, the flow entry will match all the packets whose source's IP address is equal to veth1's IP address, and set the instruction for dropping the

TABLE 2 Northbound APIs

	Provides a set of REST APIs for:
ofctl_rest.py	retrieving and updating switch statistics
rest_conf_switch.py	switch configuration
rest_topology.py	links configurations
rest_router.py	getting/deleting/setting address data and routing data
rest_qos.py	getting/deleting/setting qos rules/meter entries/queue status
rest_firewall.py	getting/setting firewall status; getting/deleting/setting firewall rules

TABLE 3 Set of REST APIs provided by rest_conf_switch.py

REST API	**URI**	**Description**
GET	/v1.0/conf/switches	G et all the switches
	/v1.0/conf/switches/{dpid}	g et all the configuration keys of the switch *dpid*
	/v1.0/conf/switches/{dpid}/{key}	get the *key* configuration of the switch *dpid*
PUT	/v1.0/conf/switches/{dpid}/{key}	set the *key* configuration of the switch *dpid*
DELETE	/v1.0/conf/switches/{dpid}	delete all the configuration of the switch *dpid*
	/v1.0/conf/switches/{dpid}/{key}	delete the *key* configuration of the switch *dpid*

packet. This last entry is inserted to reduce the switch-to-controller traffic due to forwarding of the *nmap*'s TCP/IP traffic between interconnected switches.

Thus, after receiving the ICMP reply within a *packet_in* message, the controller just needs to change the destination mac address of the Ethernet frame and send it to the *nmap* application though the raw layer 2 socket.

The UDP traffic generated by *nmap* is processed in a similar way. The destination mac address of the Ethernet frames that are sent to the switches is changed to the broadcast address. Regarding the response traffic, the destination mac address is changed to the veth1's mac address. However, for the TCP traffic, the procedure is a bit different. The Linux kernel does not allow encapsulating a TCP packet into an Ethernet frame with a broadcast destination address. The destination address must be unicast. This fact only affects the "*nmap -PS*" and "*nmap -PA*" commands (see Table 1), which are used to perform the host discovery task if ICMP pings are blocked. In the rest of the cases, the TCP traffic is generated once a host has been discovered, so its mac address is known.

As shown in Table 4, this new module has been programmed to offer four northbound REST APIs, corresponding to the most used *nmap* options.

The returned format of the host discovery and OS detection APIs is as follows:

```
HTTP/1.1 200 OK
Content-type: html
{Corresponding nmap output}
```

Figure 2 shows a screenshot after using the REST API to discover the OS of a host.

In the case of the port scanning API, the returned format is as follows:

```
HTTP/1.1 200 OK
Content-type: json
{JSON data}
```

TABLE 4 Set of REST APIs provided by sdn_nmap.py

REST API	URI	Description
GET	/nmapsdn/hosts/{net}/{mask}	Performs host discovery on the *net/mask* network
	/nmapsdn/ports/{ip}	Scans the most 1000 common ports of host *ip*
	/nmapsdn/ports-reduced/{ip}	Scans the most 100 common ports of host *ip*
	/nmapsdn/os/{ip}	Detects the operating system of host *ip*

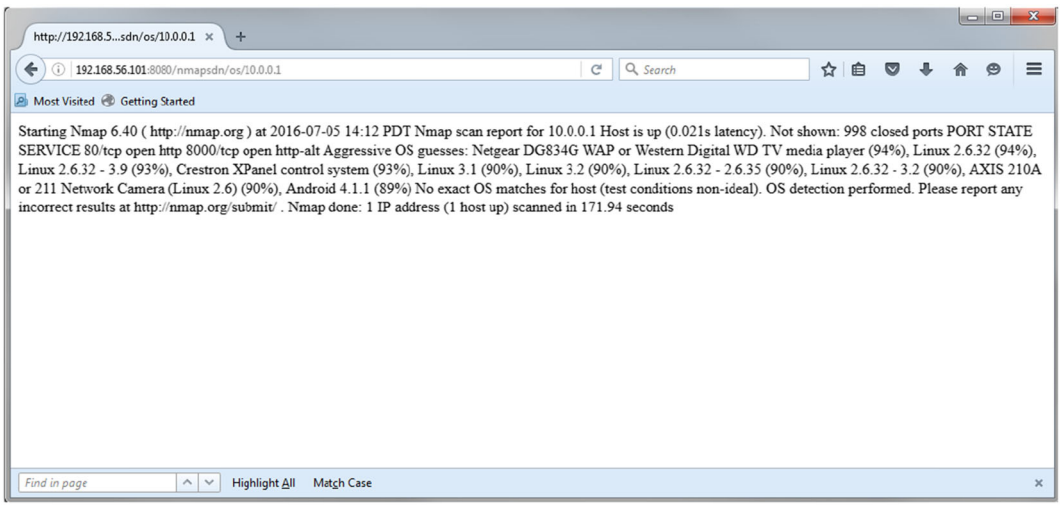

FIGURE 2 Example of the operating system detection output

where the corresponding JSON data are, for example,

```
[{"port":"80","state":"open","service":"http"},
{"port":"8000","state":"open","service":"http-alt"}].
```

Finally, as can be seen in the screenshot shown in Figure 3, we have integrated the use of this API into the topology viewer provided by Ryu (*ryu.app.gui_topology.gui_topology.py*). The original topology viewer shows the SDN network controlled by Ryu, concretely, the switches, links, and only those hosts that have already generated traffic. Additionally, when the user double-clicks a switch, the */stats/flow/dpid* API is employed to get the flow statistics of the switch. The obtained information is shown at the bottom of the website.

Using our virtualized network infrastructure and the enhanced version of the topology viewer, after performing the host discovery task, the topology viewer will be able to show the switches, links, and all the discovered hosts, even if they have not generated traffic yet. In addition, when the user double-clicks a host, the port discovery API corresponding to that host is employed. The result will be also shown at the bottom of the website (see Figure 3).

5 | EXAMPLES OF USE

As in traditional networks, network monitoring is an essential task for network management in the case of SDN networks. Whatever the solution chosen to perform the monitoring (an interesting survey can be found in[20]), it would provide useful information to reach an efficient configuration and management of the networks.

However, in addition to network monitoring, network mapping can help to improve even more the operation of the networks. The better the knowledge of the network is, the better the network can be managed and defended. Next, we are going to describe some examples of use in which network mapping tasks can help network management.

The Internet infrastructure, a university's network, and cloud datacenters are some examples of enterprise networks, where resource allocation, network management, and security are laborious to control. These networks, which typically have massive infrastructures that are tedious to manage, benefit from SDN technology.

In this scenario, workers usually connect to the enterprise network not only by desktops but also by laptops. When a worker receives a computer, the network administrator can guarantee that the computer is "clean," but after that moment, this cannot be assured. For example, the worker could click accidentally on an email attachment consisting of malware,

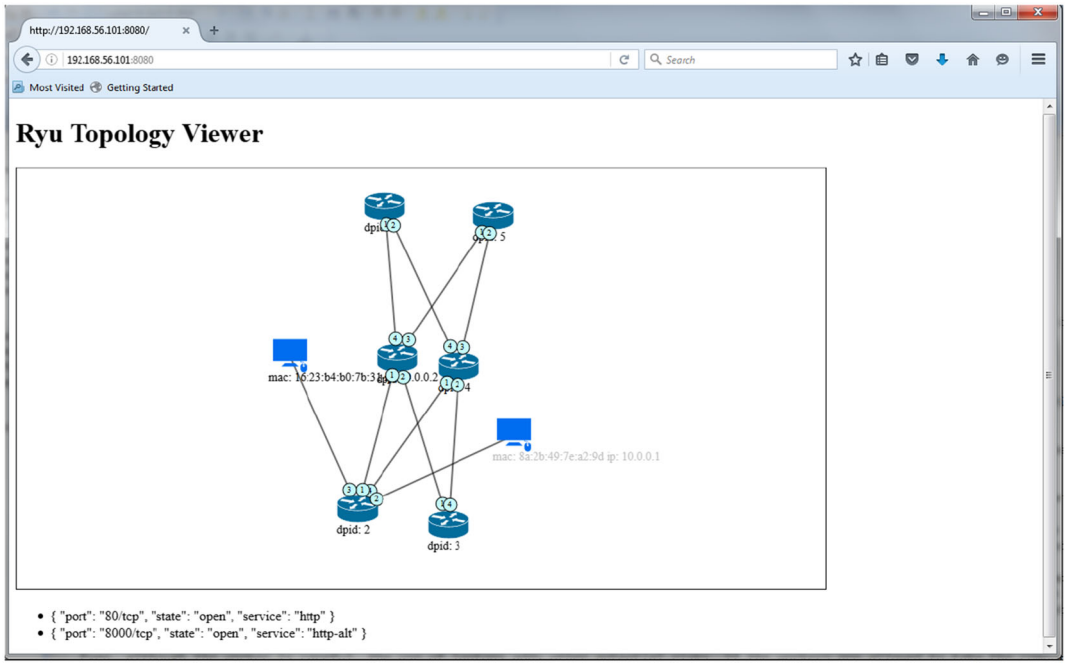

FIGURE 3 Example of the operating system detection output

which could end up installing an undesired program in the computer. Port scanning can be useful in this situation. If the worker's computer is supposed to not be running any server program, no ports on the computer should be open.

Even, if the worker is careful, the use of laptops adds other potential risks. If the workers are allowed to take the computer home, the risk of malware infection from the employee's home network should be considered. A real case of a small company that was infected by Sircam virus a few years ago is described.[21] An employee took his laptop home to continue working, and he allowed his daughter to access her emails from the laptop, which unfortunately included an infected message.

Continuing with security aspects, OS detection could also help network administrators. If the workers of a company are required to keep an updated operating system version, traffic generated from nonupdated computers can be redirected to be analysed.

Some companies have strict policies that prevent employees from using personal devices (smartphones, tablets, etc) at the workplace. Others do not. In this situation, network mapping could be useful. OS detection could allow identifying these devices. Thus, the network could be configured accordingly, routing the associated traffic through particular paths or even limiting the network bandwidth.

6 | CONCLUSIONS

In this paper, we have proposed a virtualized network architecture that allows the integration of existing software tools in the management and control of OpenFlow-based SDN networks. The proposed architecture makes use of different virtualization technologies; concretely, it employs software-defined virtual switches (OVS) and Linux network namespaces.

The proposed infrastructure allows the implementation of different scenarios, where different SDN controllers and software tools can coexist. In this paper, the defined infrastructure has been used to allow network administrators and application developers to make use of the well-known network mapping tool *nmap* in OpenFlow-based SDN networks controlled by Ryu. In the implemented solution, users can make use of the complete functionality of the command line tool utility. In addition, a northbound REST API has been coded to offer the most interesting *nmap* commands. By way of example, we have integrated the use of the defined API into the topology viewer provided by Ryu.

ACKNOWLEDGEMENT

This work was supported by the MINECO/FEDER Project Grants TEC2013-47016-C2-2-R (COINS) and TEC2016-76465-C2-1-R (AIM).

ORCID

Pilar Manzanares-Lopez https://orcid.org/0000-0003-1296-7158
Juan Pedro Muñoz-Gea https://orcid.org/0000-0001-8342-4797
Josemaria Malgosa-Sanahuja https://orcid.org/0000-0001-8137-1089

REFERENCES

1. Openflow. Available: <https://www.opennetworking.org/sdn-resources/openflow>; 2016.
2. Lyon G. *Nmap Network Scanning*. Sunnyvale CA, USA: Insecure.com LLC; 2008.
3. Build SDN agilely. Available: <https://osrg.github.io/ryu/>; 2016.
4. Senekal F, Vorster J. Network mapping and usage determination. In: Proceedings of MICSSA 2007; 2007 Pretoria, South Africa. pp. 1-12.
5. Postel J. Internet control message protocol. In: RFC 792; 1981.
6. Plummer D. C. An ethernet address resolution protocol. In: RFC 826; 1982.
7. Mockapetris P. Domain names. Concepts and facilities. In: RFC 1034; 1987.
8. Mockapetris P. Domain names. Implementation and specification. In: RFC 1035; 1987.
9. Haleplidis E, Pentikousis K, Denazis S, Hadi Salim J, Meyer D, Koufopavlou O. SDN layers and architectures terminology. In: RFC 7426; 2016.
10. Nunes BAA, Mendonca M, Nguyen X-N, Obraczka K, Turletti T. A survey of software-defined networking: past, present, and future of programmable networks. *IEEE Commun Surv Tutorials*. 2014;16(3):1617-1634.

11. OpenDayLight. A Linux foundation collaborative project. Available: <https://www.opendaylight.org>; 2016.
12. Zhou W, Li L, Luo M, Chou W. REST API design patterns for SDN northbound API. In: Proceedings of 28 th. International Conference on Advanced Information Networking and Applications Workshops; 2014. Victoria, Canada. pp. 358-365
13. Mosharaf N, Chowdhury K, Boutaba R. A survey of network virtualization. *Comput Networks*. 2010;54(5):862-876.
14. Network functions virtualisation: an introduction, benefits, enablers, challenges & call for action. Available: <http://portal.etsi.org/NFV/NFV_White_Paper.pdf>; 2012.
15. Open vSwitch: An open virtual switch. Available: <http://www.openvswitch.org>; 2016.
16. Linux namespaces. Available: <http://man7.org/linux/man-pages/man7/namespaces.7.html>; 2016.
17. Manzanares P, Muñoz J, Delicado F, Malgosa J, Flores A. Host discovery solution: an enhancement of topology discovery in OpenFlow based SDN networks. In: Proceedings of 7th. DCNET; 2016. Lisbon, Portugal. pp. 80-88.
18. Gheorghe G, Avanesov T, Palattella M, Engel T, Popoviciu C. SDN-RADAR: network troubleshooting combining user experience and SDN capabilities. In: 1st. IEEE Conference on Network Softwarization (NetSoft); 2015. London, UK. pp. 1-5.
19. Xing T, Xiong Z, Huang D, Medhi D. SDNIPS: enabling software-defined networking based intrusion prevention system in clouds. In: 10 th. International Conference on Network and Service Management (CNSM); 2014. Rio de Janeiro, Brasil. pp. 308-311
20. Yassine A, Rahimi H, Shirmohammadi S. Software defined network traffic measurement: current trends and challenges. *IEEE Instrum Meas Mag*. 2015;18(2):42-50.
21. http://www.computerweekly.com/feature/The-security-dangers-of-home-networks; 2016.

How to cite this article: Manzanares-Lopez P, Muñoz-Gea JP, Malgosa-Sanahuja J, Flores-de la Cruz A. A virtualized infrastructure to offer network mapping functionality in SDN networks. *Int J Commun Syst*. 2019;32:e3961. https://doi.org/10.1002/dac.3961

APPENDIX A

This appendix details the commands to implement the virtual infrastructure in a Linux machine.

```
ip netns add ns2  #create the new namespace
ovs-vsctl add-br snmap #create the virtual bridge
ovs-vsctl add-port snmap snmap-veth0  #add the ports
ovs-vsctl add-port snmap snmap-veth1  #to the virtual bridge
#create the veth pairs
ip link add veth0 type veth peer name snmap-veth0
ip link add veth1 type veth peer name snmap-veth1
#move the virtual interface veth1 to the ns2 namespace
ip link set veth1 netns ns2
```

And the commands to configure the virtual network are as follows:

```
ip netns exec ns2 ifconfig veth1 192.168.3.1
ip netns exec ns2 route add default gw 192.168.3.254
ip netns exec ns2 ifconfig veth1 hw ether 00:33:33:ff:ff:ff
ifconfig veth0 192.168.3.254
ifconfig veth0 hw ether 00:22:22:ff:ff:ff
ifconfig snmap-veth0 up
ifconfig snmap-veth1 up
```