Predictive Path Routing Algorithm for Low-Latency Traffic in NFV-based Experimental Testbed

Juncal Uriol, Juan Felipe Mogollón, Mikel Serón, Roberto Viola, Ángel Martín, Mikel Zorrilla, and Jon Montalbán

***Abstract*—The growth of network traffic and the rise of new network applications having heterogeneous requirements are stressing the telecommunication infrastructure and pushing network management to undergo profound changes. Network management is becoming a core research area to push the network and its performance to the limits, as it aims at applying dynamic changes across the network nodes to fit the requirements of each specific network traffic or application. Here, solutions and frameworks based on software-defined networking (SDN) and network function virtualization (NFV) facilitate the monitorization and control of both the network infrastructure and the network services running on top of it. This article identifies and analyzes different implemented solutions to perform experiments on network management. In this context, an innovative experimental testbed is described and implemented to allow experimentation. A predictive path routing algorithm is later proposed and tested by designing experiments with specific network topologies and configurations deployed through the testbed. The algorithm exploits predictions on network latency to change the routing rules. Finally, the article identifies the open challenges and missing functions to achieve next-generation network management.**

***Index Terms*—Network analytics, network function virtualization, networking, software-defined networking.**

I. Introduction

5G networks and beyond will experience profound changes to cope with the requirements of network-based applications and services. The networks should provide increased flexibility and better resource utilization to address specific application requirements and assess the demanded quality of service (QoS) [1], as well as reduced capital expenditure (CAPEX) and operational expenditure (OPEX) [2], [3]. To achieve it, the objective is to create an adaptive and automated network capable of monitoring and analyzing itself to perform operations to maintain its performance. Hence, the network should be able to automatically trigger actions in response to detected events or monitored changes in its behaviour. A network with such capabilities is usually referred to as self-organizing network (SON) [4]. A SON includes specialized functionalities that deal with configuration (self-configuration), optimization (self-optimization) and healing (self-healing) of the network, which could be complementary.

New technologies that emerged in recent years, such as software-defined networking (SDN) [5] and network functions virtualization (NFV) [6], are fundamental enablers to provide the higher levels of automation necessary to implement a SON. SDN and NFV follow the principle of decoupling softwarized functions from general-purpose hardware, usually referred commercial off-the-shelf (COTS), where they are meant to be run. SDN solutions focus on forwarding capabilities, i.e., layer 2 (L2) and layer 3 (L3) of the open systems interconnection (OSI) model, providing a centralized control to monitor and operate distributed network routers and switches. NFV technologies manage higher layers (L4-7) of the OSI model, virtualizing RAM and CPU resources and simplifying the lifecycle management of software instances of network functions, referred to as virtual network functions (VNF), or a combination of them in a complete infrastructure, referred as network service (NS), on top of them.

Consequently, SDN and NFV are profoundly changing the telecommunication infrastructures and pushing the research on network deployment, monitoring and management. The combination of SDN and NFV capabilities will create virtualized networks embedding heterogeneous softwarized functions and running on top of programmable network devices.

However, the paradigm shift introduced by SDN and NFV brings with it significant challenges. The most important one is their interoperability inside the network architecture. While their complementarity is evident, their integration in a common network infrastructure still presents major issues to overcome. SDN and NFV solutions will need further steps to integrate and achieve a fully programmable virtualized network.

In this context of virtualized networks implemented through SDN and NFV to implement SONs, the contributions of this work are the following:

- This article identifies and analyzes different solutions based on SDN and NFV to perform experiments on management of virtualized networks. For each of them, both features and limitations are studied.

Manuscript received May 5, 2022 revised December 2, 2022; approved for publication by Mubashir Husain Rehmani, Division 3 Editor, April 15, 2023.

This research was supported by the Spanish Centre for the Development of Industrial Technology (CDTI) and the Ministry of Economy, Industry and Competitiveness under grant/project CER-20191015 / Open, Virtualized Technology Demonstrators for Smart Networks (Open-VERSO).

J. Uriol is with Fundación Vicomtech, Basque Research and Technology Alliance, 20009 San Sebastián, and with the Department of Communications Engineering, University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain, email: juriol@vicomtech.org.

J. F. Mogollón, M. Serón, R. Viola, Á. Martín, and M. Zorrilla are with Fundación Vicomtech, Basque Research and Technology Alliance, 20009 San Sebastián, Spain, email: {fmogollon, mseron, rviola, amartin, mzorrilla}@vicomtech.org.

J. Montalbán is with the Department of Electronic Technology, University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain, email: jon.montalban@ehu.eus.

J. Uriol is the corresponding author.

Digital Object Identifier: 10.23919/JCN.2023.000018

1229-2370/23/$10.00 © 2023 KICS

- An innovative and experimental testbed is proposed and implemented to perform experiments dealing with network management. The testbed includes functional NFV-compliant solutions to simplify the deployment and control over the experiments. Moreover, the testbed provides the flexibility to run heterogeneous network services and applications embedded in virtual machines (VMs).
- A predicted path routing algorithm, based on latency predictions, is proposed and employed to validate the testbed when running SON experiments. The algorithm uses the matricial autoregressive (MAR) model and Dijkstra's algorithm to predict links' latencies and select the optimal end-to-end path. The design of the experiments includes specific network topologies and configurations deployed through the testbed.
- This work also discusses the state of the art concerning the management of virtualized networks and identifies open challenges and missing functions which future research activities will address.

The remainder of the article is organized as follows. Section II describes the state of the art of 5G and beyond network management. Section III analyses the tested alternatives for network management and describes the selected ones deployed in the experimental testbed proposed in this work. Section IV describes the predictive path routing algorithm for low-latency, implemented and deployed across the aforementioned experimental testbed. Section V includes the overall experimental setup and implementation details and provides numerical results of the validation. Section VI discusses the achieved results and the open challenges. Finally, in Section VII we assert our conclusions.

II. Network Management: State of the Art

A. Architecture of Network Virtualization

Network management relies on softwarization and virtualization technologies, such as SDN and NFV. The former aims at instructing distributed network nodes implementing L2 and L3 packet forwarding (network routers and switches). The latter allows instead the deployment of VNFs implementing L4-7 functionalities.

Going deeply, SDN [5] consists in centralizing the network control and the management of forwarding rules between distributed data centers and the VNFs running on them. It enables the separation between control and data planes such that the control plane is employed to operate the data plane with forwarding rules to be applied when processing data packets. An SDN controller is employed to have a global network topology view. It implements the control plane to manage all the SDN-enabled devices, such as switches and routers, representing the data plane. Therefore, the SDN paradigm creates an abstraction layer for the network administrator, who no longer needs to manually configure each network node and can applies programmed policies.

On the other side, NFV [6] enables virtualizing the physical resources available at the data centers distributed across different network locations and interconnected though SDN. In network management, the virtualization, already widely employed in cloud computing platforms, allows to easily scale or migrate a NS, composed by one or more network VNFs, depending on the demanded computational resources and the network status at any moment. Thus, a NS can be deployed as a combination of VNFs and does not need a specific hardware configuration anymore, as VNFs can run on top of general-purpose hardware.

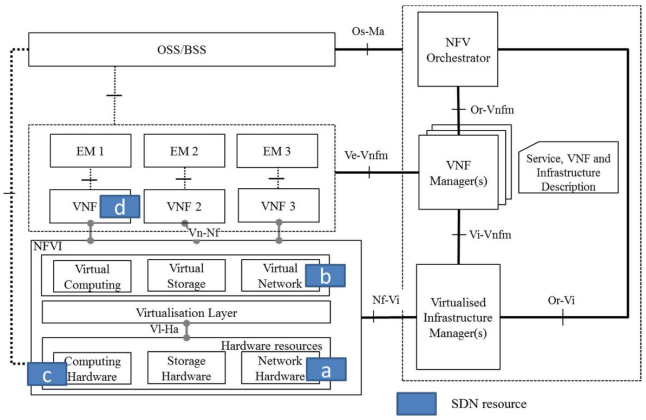

Fig. 1. NFV MANO architecture and possible integration with SDN. Source: [7, Fig. 2].

NFV management and orchestration (NFV-MANO), presented in Fig. 1, is the architecture proposed by the European telecommunications standards institute (ETSI) to cope with the needs to effectively deploy and manage NSs, VNFs and the underlying infrastructure. The NFV-MANO includes three main components: The virtual infrastructure manager (VIM), the virtual network function manager (VNFM) and the network function virtualization orchestrator (NFVO).

The VIM controls and manages the NFV infrastructure (NFVI) by virtualizing the psychical resources, including computation, storage and network ones. It creates and assigns the virtual resources needed by each VNF. The VNFM oversees the lifecycle of VNFs deployed on top of the NFVI, including the configuration, deployment, scaling operations and the termination. Finally, the NFVO is responsible for the NSs and VNFs by validating them before the deployment. It is also in charge of the lifecycle management of NSs, meaning the orchestration of the different VNFs included in the NS according to programmed networking policies.

In general, the NFV stack is expected to increase flexibility, improve the utilization of network resources, as well as to provide the ability to fit the requirements of each network application in terms of QoS with reduced CAPEX and OPEX [2], [3].

To achieve it, NFV enables the management of physical resources available at each data center within the network, such as RAM and CPU, and creates logical or virtual resources to be assigned at each VNF.

The integration of SDN and NFV enables the operation and management of VNFs instances on top of virtualized resources available at NFV-enabled data centers and interconnected

through the forwarding rules of the SDN resources, such as switches and routers, established by an SDN controller. To achieve such integration, several possibilities are envisioned [7], on top of the ETSI's NFV-MANO stack shown in Fig. 1. The SDN resources might be located depending on the nature of programmed systems:

- Physical switch or router (case a): Hardware-based implemented solution with software interfaces to program it;
- Virtual switch or router (case b): Software-based solution running on top of virtual resources;
- E-switch (case c): Software-based switch in a general-purpose physical server;
- Switch or router as a VNF (case d): Software-based solution running as VNF.

Accordingly, the SDN controller can be deployed as part of the NFVI, on top of physical or virtual resources and managed by the VIM, or as VNF, managed by the VNFM [7].

B. SDN Technologies

The main feature of the SDN paradigm is the separation of the control and data planes. To achieve it, the SDN controller provides a Northbound API to enable the communication between the network administrator application and the control plane. A Southbound API communicates between the SDN controller and network devices. Unlike the Northbound API, where each SDN controller has its REST API implementation, the Southbound API is usually based on standard and widely employed protocols, such as OpenFlow [8] and NETCONF [9] to bridge universal interoperability.

Concerning the control plane, lots of SDN controller implementations are available, where the most employed for research purposes are open network operating system (ONOS) [10], OpenDaylight (ODL) [11], RYU [12], and FloodLight [13]. A comparison of standard SDN controllers is presented in [14], where several features, such as programming language, graphical user interface (GUI) and APIs, platform support and internal architecture (modularity, distributed/centralized), are considered.

For the switches and routers implementing the forwarding plane, every OpenFlow-enabled device can be managed by all the available SDN controllers. Solutions range from vendor-specific hardware solutions (case a), e.g., Cisco systems and Juniper networks, to open-source software implementations, where the most employed is open vSwitch (OVS) [15], [16]. The software nature of OVS has made it perfect for being installed on generic hardware acting as a switch (case c). The authors of [17] employ OVS to create an SDN-enabled switch using a Raspberry Pi [18]. Furthermore, OVS can also be run inside a VM, allowing the generation of virtual networks inside a data center (case b) [19] or as a VNF on top of the virtualized environment (case d). In the last case, a VNF-based switch is not employed as it does not present any significant advantages over other solutions, but some implementations are provided [20].

In any case, having physical or virtualized (VM/VNF) systems to perform experiments is complex and costly in terms of hardware or virtualization infrastructure setup and maintenance. Mininet [21] is a tool for creating realistic virtual networks by instantiating and interconnecting several OpenFlow/OVS-enabled switches on a single machine, simplifying the setup. Nowadays, Mininet represents the most common solution to investigate SDN routing algorithms [22], [23].

Finally, the problem of interconnecting several networks, each one with its SDN controller, has been rising in the last few years and remains unsolved. An East-West API is required to enable the coordination between the different SDN controller implementations. However, it is still far from being standardized; even some proposals are already being discussed in literature [24], [25], [26].

C. NFV Technologies

The introduction of NFV is essential to increase flexibility when provisioning network resources. It enables fitting the QoS requirements of applications in a network while reducing the costs for its operation (CAPEX and OPEX).

As previously explained, NFV MANO architecture has three different components: VIM, VNFM and NFVO. Each of these components provides APIs according to ETSI specifications [27], [28] in order to intercommunicate with each other.

When considering VIM solutions, common public cloud platforms, such as Amazon web services (AWS), Microsoft Azure and Google cloud platform, already provide APIs to be used as VIMs. OpenVIM [29], hosted by ETSI, and OpenStack [30], supported by open infrastructure foundation [31], represent the reference solutions and the most used ones. Both of them are open source and have proven to be valid for managing private data centers [32].

Regarding VNFM and NFVO, it is not easy to separate the solutions between those that provide VNFM and those that provide NFVO, as in many cases, they are together in the same suite. Regarding open source software, the most relevant VNFM/NFVO implementations are open source MANO (OSM) [33], hosted by ETSI, and open network automation platform (ONAP) [34], supported by Linux Foundation. VMware vCloud director [35] is also meant for private clouds but consists of proprietary software. A comparison between them is presented in [36]. Other existing solutions are Tacker, an OpenStack project consisting of a generic VNFM/NFVO, open Baton, and Cloudify [37].

Among commercial solutions, there are Cisco network services orchestrator [38], Ericsson network manager [39] and ZTE CloudStudio [40], [41].

D. Network Monitoring

Monitoring is an essential process in modern networks to provide insights in terms of the satisfaction of service level agreement (SLA), matching the key performance indicators (KPIs) from the network and the QoS requirements from applications. Thus, the network is constantly monitored to detect network issues, virtual functions life-cycle or QoS violations to perform actions that restore the proper operation.

According to ETSI specification, network monitoring tasks may be passive, active or hybrid [42]. Passive monitoring consists in observing network traffic generated by network applications and users. It is limited to collecting data already available at the network agents. Here, the type of traffic and the duration of network flows influence the measurements [43]. However, measurements may not be available during periods when network traffic is not generated.

On the other hand, active measurement aims to perform a more extensive diagnostic of the network conditions and determine if network packets are correctly transferred between hosts. To achieve it, active monitoring involves the generation of synthetic or test traffic to validate network applications' performance and verify that the SLA is fulfilled. Then, instead of simply collecting information available at network agents, it generates and sends traffic flows to analyze the behaviour of the network [44]. This, for sure, brings some communication overheads.

Finally, as the name suggests, hybrid monitoring is an approach that uses the information obtained from both active and passive monitoring. Already existing traffic flows provide passive measurements, while added testing traffic flows enables active measurements.

In the context of the virtualized network, the set of metrics to monitor is broader compared to a legacy physical network. Nowadays, virtualized networks also provide distributed CPU, memory and disk capabilities to be shared among the network applications [45]. It means that new metrics such as CPU load or memory usage are now necessary to be considered when describing the network behaviour, together with legacy metrics limited to describing the communication channel, e.g., packet loss or network delay. It results in continuous monitoring of performance metrics on each SDN switch, NFV infrastructure or VNF instance. Moreover, specific time series databases and visual analytics tools are included in SDN/NFV environment to store, visualize and process the collected measurements. Suppose the measurements do not comply with the SLA or QoS requirements. In that case, alarms are triggered, and actions are set up according to programmed policies to modify the behaviour of the network and guarantee that the requirements are fulfilled.

Prometheus [46], InfluxDB [47], and Elasticsearch [48] represents the most employed time series database, while Grafana [49] and Kibana [50] are the worth of mention tools to visualize data through data charts and dashboards, i.e., the composition of data charts into a unique visualization.

E. Routing Algorithms

Traditional or best-effort routing treats all traffic flows equally, no matter their requirements. It means that all the traffic flows share the network resources without any prioritization or differences when choosing the delivery path, i.e., if two flows have the same source and destination, they have the same delivery path and compete for the network resources without any mediation. It considers fairness, overall throughput, and average response time as the essential performance aspects of traditional routing.

Nevertheless, traditional routing has already been replaced by QoS routing. QoS routing aims at guaranteeing the appropriate resources for each traffic flow. It is connection-oriented, providing each traffic flow with a resource reservation according to its QoS requirements. Hence, meeting the QoS requirements is the key to evaluating the effectiveness of the routing strategy. Moreover, depending on the QoS requirements, a different problem can be defined to find the optimal routing path [51], where the most common are the following ones:

- Shortest path (SP): The route has to minimize a unique end-to-end QoS metric, referred to as the *cost*.
- Constrained shortest path (CSP): The route has to minimize an end-to-end QoS metric while being constrained by a defined bound of another metric, referred to as the *constraint*.
- Multi-constrained shortest path (MCSP): Similar to the CSP problem, but in this case, multiple *constraints* have individual bound constraints.
- Multi-constrained path (MCP): It is similar to MCSP, but without the *cost* to be optimized. The route has only to keep the *constraints* below prescribed bounds.

These fundamental problems can also be extended to find k distinct paths that optimize the QoS metric and/or fulfill the constraints [52] or to optimize more than one QoS metric [53].

This work focuses on the SP problem, where Dijkstra [54] is the well-known and widest employed algorithm. It works by selecting an optimal partial path at each iteration till finding the optimal end-to-end path. A partial path is a path that starts from the source node and reaches an intermediate node which is not the ultimate destination. At each iteration, it takes the least-cost path among the available partial paths, and then it generates k new paths by extending the chosen partial path with the k outgoing edges of the node.

Different surveys [51], [55], [56] compile and study other SP algorithms, such as the Bellman-Ford and the Floyd-Warhsall ones. These are similar to Dijkstra algorithm. The Bellman-Ford algorithm finds the shortest path between a given node and all other nodes in the graph, sharing the goal with Dijkstra. Bellman-Ford is slower than Dijkstra, while it is more versatile, as it is compatible with graphs with negative weights. This implies that there is no shortest path in negative cycles where the sum of edges means a negative value, therefore the algorithm is prevented from being able to find the correct route since it terminates on a negative cycle. This algorithm can detect negative cycles and report on their existence. The Floyd-Warshall algorithm is an SP algorithm that stands out because, unlike the previous two algorithms (Dijkstra and Bellman-Ford), it is not a single-source algorithm. This means that it calculates the shortest distance between every pair of nodes in the graph rather than only calculating from a single node. It works by breaking the main problem into smaller ones and combining the answers to solve the shortest path issue. This algorithm is beneficial when generating routes for multi-stop trips as it calculates the shortest path between all the relevant nodes. For this reason, many route planning software utilize this algorithm as

it provides the most optimized route from any given location. Another SP algorithm is given by Johnson's algorithm and is a way to find the shortest paths between all pairs of vertices in an edge-weighted directed graph. It allows some edge weights to be negative numbers, but no negative-weights cycles may exist. It works by using the Bellman-Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph. Finally, as indicated at [51], the A* algorithm was proposed by Hart et al. for finding a single-destination SP by introducing a so-called guess function. At each node, this guess function provides an estimation for the cost of the SP from this node to the destination node.

III. Testbed for Network Management

This section describes the testbed employed for the experimentation with network management. For this purpose, the first subsection overviews different alternatives to perform the SON experiment for low latency routing comprising the considered characteristics. Then, the second subsection compares the alternatives, their advantages and limitations, and the selected one to perform our experiments on network management.

A. Alternatives for Network Management

To perform flexible experimentation on predictive routing, the testbed include the following elements or features:

- Deployment of the virtual networks
- Programmable routing capability
- A management interface for controlling the virtual networks
- Endpoints for monitoring the networks

The identified and evaluated testbed alternatives to set up our experiment are analyzed in this subsection.

1) SDN emulation: To provide a complete vision of alternatives for network management, the description of alternatives starts with SDN emulation tools, representing a good solution for fast prototyping. Still, their capabilities could be limited in deploying a complex network scenario.

Mininet [21] is a well-known SDN emulation tool which employs virtualization mechanisms to enable the deployment and test of networks on a single machine. It is widely employed in the research community since it comes with interesting features, such as the capacity for fast prototyping and the possibility of sharing the prototypes and results with other scientists. As backwards, it presents limitations when considering the performance fidelity between the emulated and the real environment. Each network node is implemented through a Linux namespace, which means that all the network nodes share the same hardware resources available on one single machine, also resulting in a disadvantage for experiments on a larger scale [57].

Containernet [58], Maxinet [59] and VIM-EMU [60] are more recent tools to emulate networks. All of them are based on Mininet and aim to reduce the gap between emulated and real environments. Containernet [58] substitutes Linux namespaces with containers, which means providing a separation between network nodes closer to a NFV infrastructure. Containers share kernel resources, but each has its software stack to differentiate it from the others [61]. Maxinet [59] extends Mininet to span the emulated network over several machines. It allows scaling the emulated network by enabling more network nodes and distributing them among different machines. Finally, VIM-EMU [60] evolves Containernet by emulating a multi-point of presence (multi-PoP) environment. The containers deployed through Containernet are instantiated into an emulated distributed data centers architecture. VIM-EMU also features an API to allow it to be employed as a VIM component in NFV MANO architecture.

2) SDN-controlled OpenStack: To overcome the limitations and complexity of configuration and maintenance of deployment of bare-metal machines for SDNs, NFVI solutions are employed to arrange cloud computing platforms automatically. Several solutions are available as commercial deployment, e.g., Amazon EC2 [62], Google cloud engine [63] and Azure [64], or open source software, e.g., OpenStack [65], Eucalyptus [66] and Opennebula [67]. OpenStack is the most attractive solution for our testbed deployment, as it has a more extensive foundation supporting its development, including all the major tech industry players [68], [69]. Thus, the deployment of OpenStack in our infrastructure enables the capability to deploy multiple virtual machines and virtual networks to create network topologies and configure them for any experiment.

OpenStack offers several interfaces to control and manage the infrastructure, such as a web GUI, a command line client and a REST API, and an orchestration engine, called *Heat* [70], to launch multiple composite networks and machines using a descriptor file.

OpenStack networking architecture and management are based on *Neutron* engine [71], which is the software element capable of creating, modifying and deleting different virtual network elements available on OpenStack, such as virtual networks and routers. The Neutron engine's primary technology is OVS, the element capable of creating those virtual network elements.

To manage OpenStack virtual network, we have tested the possibility of integrating it with an SDN controller, for example, the widely employed ODL solution [11]. We have researched two different ways to control OpenStack virtual networking devices using ODL:

1) OpenDaylight-Neutron integration
2) OpenDaylight controlling OVS

To get a direct integration between OpenDaylight and Neutron, there is a plugin called *Networking-ODL*, which is supposed to make available the control of OpenStack devices by ODL [72]. The interaction between ODL and Neutron allows us to monitor the network topology already deployed into OpenStack infrastructure and the traffic crossing the virtual networks. However, we cannot manage the performance of the virtual networks and create or reroute no one of those

virtual networks. Based on these limitations, we discarded this approach to manage Neutron infrastructure through ODL.

Following the approach to get a connection between OpenStack and an SDN controller, we have checked the possibility of connecting Open vSwitch through standard SDN protocols like OpenFlow directly. This way, we use tools such OpenDaylight to manage Open vSwitch virtual devices [73]. This approach brings a high limitation as OpenStack is able to manage just one Open vSwitch in each OpenStack deployment when creating virtual network devices. Therefore, a topology cannot be arranged with just one Open vSwitch.

3) VNF-enabled switches: After checking the limitations of the results provided by former approaches given in III-A2, we have tried several VNF approaches to get a working testbed environment for the management of routing on network topologies.

The first approach to set up a VNF [74] networking architecture was deploying a virtualized network topology using Ubuntu VMs in OpenStack as routers. Here, we scale the network up by adding one more node or scale it down by removing a node based on network traffic. In this setup, OSM manages the deployment of those VMs. The VNF descriptor for describing and configuring those VMs includes a *cloud init* file for each VM that composes the topology by performing *Day-0* actions in order to configure the VM once it is deployed. Here, the automated routing of the topology deployed based on those Ubuntu VMs is applied in the configuration of the *cloud init* file of each VM. Thus, the idea was to change their routing table through *Day-0* actions, ensuring connectivity in the whole topology.

The second approach for VNF architecture consists in using OVS VMs. Network topology is built by running multiple Open vSwitch instances, each consisting of a single virtual machine deployed in our OpenStack infrastructure.

The overall network architecture consists of several instances (VMs) connected by multiple virtual networks inside the OpenStack infrastructure. Each of these instances has two network interfaces and runs OVS [75] software to manage them. Thus, OVS acts as a gateway between the network interfaces available at each VM and interconnects the virtual networks where such interfaces are connected. Taking advantage of the capabilities of OVS to be managed by using SDN protocols will enable the management of the network topology by an SDN controller.

4) OpenStack native networking: OpenStack, through its network engine Neutron, provides some NFV devices like virtual networks, firewalls and routers. OpenStack virtual routers are called *qrouters*, and have capabilities to re-route incoming and outgoing traffic through the connected networks to the router. Those routing capabilities are very useful to modify the behavior of the traffic routing inside the network topology in our testbed.

Qrouters provide routing functionality into the OpenStack virtual networking infrastructure and can be configured in a fancy way in order to redirect traffic through the different networks they are connected to. That possibility, combined with native OpenStack networking capabilities, offers us a robust scenario where multiple virtual networks interconnected through multiple *qrouters* can route incoming traffic from each network in a way to arrange custom topologies for the given case.

Qrouters and virtual networks are part of OpenStack standard tools and can be managed by the standard interfaces offered by OpenStack, like API/REST, command line and orchestration services like *Heat*. This way, management of arranged topology infrastructure, in terms of deployment and configuration, can be automated.

Heat [76] orchestration engine from OpenStack can create and modify multiple bandwidth-related limitations from OpenStack networking devices. It can manage those limitations in *Day-1* and *Day-2* actions simplifying integration and application.

B. Comparison and Testbed Selection

In this subsection, we present the analysis of the different evaluated alternatives characteristics explained in the section before, as well as the proposed experimental testbed of this experiment. Accordingly, Table I compiles and classifies all the research activities exploring different approaches to create a functional testbed for network management and routing experiments. Thus, this table shows in a summarized way the capabilities, requirements and limitations of all different approaches.

After testing all the alternatives presented above, the proposed experimental testbed is given by the *OpenStack native networking*. This testbed complies with most of the characteristics mentioned above in the previous section, so this testbed implemented and used is the best option for performing SON experiments of routing techniques on top of network management tools. This testbed refers to the ETSI architectural framework presented in Fig. 1.

We evaluated this approach and made some tests in order to check its capabilities, and the results were positive when interacting with the deployed topology. We could deploy any topology based on virtual networks and *qrouters* and modify its routing operation through OpenStack Orchestrator API several times once deployed. This kind of deployment also did not require high computing resources. We use *Heat* actions above explained to deploy an initial infrastructure composed of virtual networks, *qrouters* and virtual machines inside OpenStack infrastructure, making a *Day-1* action. By modifying *Heat* description file and commanding it again with desired modifications, the testbed topology could be modified by applying routing changes into *qrouters* as a result of a *Day-2* action.

We decided to follow this approach to implement our testbed based on these results. The next step in implementing a routing SON experiment was to determine the grade of control that can be achieved using this approach. To get a working testbed, we need to control several elements of the networking infrastructure:

- Deployment of network topologies by means of *Day-1* actions
- Definition of network configuration and rerouting rules through *Day-2* actions

TABLE I
COMPARISON BETWEEN DIFFERENT APPROACHES.

Alternative	Management interface	Network deployment	Network routing	Computational resources	Architecture	Endpoints	Limitations
SDN-Emulation [21], [58], [59], [60]	SDN	Yes	Yes	Very Low	SDN	Namespaces	Gaps to real and complex NFV setups & no bindings for integration with network management systems
SDN-OpenStack (Neutron) [72]	SDN	No	No	Low	SDN/NFV	VMs	No programmable routing or topology creation
SDN-OpenStack (Open vSwitch) [73]	SDN	No	No	Low	SDN/NFV	VMs	Limited to 1 OVS instance meaning no topology possible
VNF-Networking [74]	Cloud-init	Yes	Yes	High	VNF	VMs	No programmable routing & no easy scalability & no impact on routing when adding a new node
VNF-Open vSwitch [75]	SDN	Yes	Yes	High	VNF	VMs	No efficient deployment and scalability
OpenStack native [71], [76]	Heat, API/REST, cli	Yes	Yes	Low	NFV	VMs	*No significant drawback*

- Apply configurable bandwidth limitations at different network nodes/routers to be able to impress bottlenecks on injected network traffic
- Monitor network nodes/routers metrics to predict the routing issues and feed the SON algorithm with data

The rest of the testbed alternatives mentioned in the above section were discarded for the following reasons. Regarding the SDN emulated alternative, all the evolutions listed in Section III-A1 expand the emulation and try to deploy a more realistic environment for experimentation. However, some intrinsic limitations remain. The most important is the inability of the emulation tools to manage real machines (physical or virtual). Moreover, most of them do not provide NFV-compliant REST APIs (only VIM-EMU has this interface), making the integration within the NFV MANO architecture non-viable.

On the contrary, SDN and NFV software may enable a complete experimentation environment where VMs can be deployed and tested while managing their interconnection through virtualized networks. Even with an increase in testbed complexity, SDN and NFV software guarantee the presence of REST APIs to simplify the monitoring and management of the testbed.

Regarding the SDN-controlled OpenStack alternative, to get OpenStack and an SDN controller connected, we have employed OVS, which can be managed through standard SDN protocols like OpenFlow. This way, we can manage it directly by using ODL as before. This approach was not feasible because OpenStack can only offer one OVS in each OpenStack deployment when creating virtual network devices. Therefore, arranging a topology with just one OVS is impossible.

Once we evaluated both the SDN controller and OpenStack approaches, we decided to abandon this SDN approach because of the lack of opportunities to get a customizable topology. OpenStack only offered one OVS or did not offer any way to interact with virtual network topologies created inside OpenStack.

Related to the VNF alternative, several problems were found when deploying topologies with VNFs.

- The automated routing through Day-0 actions imposed by *cloud init* file for VMs configuration was unsuccessful as the connectivity to the topologies was impossible.
- Due to the heavy consumption of computing resources from VMs, the scalability for deploying topologies with many nodes on top of an OpenStack server is quite limited. Specifically, when using OVS as an instance, each OVS instance needs significant capabilities, i.e., at least one core and 512 MB of RAM. Thus, this approach results in a bottleneck when a topology with multiple switches needs to be deployed, causing a lack of resources to deploy virtual machines that act as typical clients or server instances that communicate with each other over the deployed network topology.
- When scaling up by adding a node, there is not effect on reducing the traffic load of the topology.

To sum up, Table I resumes the main characteristics of different alternatives. Some features underline the reasons backing the selection of an OpenStack native networking testbed. The main advantage of this testbed is that it has different management interfaces and it allows interaction with the already deployed topology, reconfiguring it. Another relevant aspect is that it means low computational resources need, enabling us to deploy different topologies while saving OpenStack's assets. Finally, as the endpoints of the testbed are VMs, it eases the monitoring of the network.

In summary, as we have mentioned before, the selected testbed is given by *OpenStack native networking*. The implementation and validation of the selected testbed are explained in Section V.

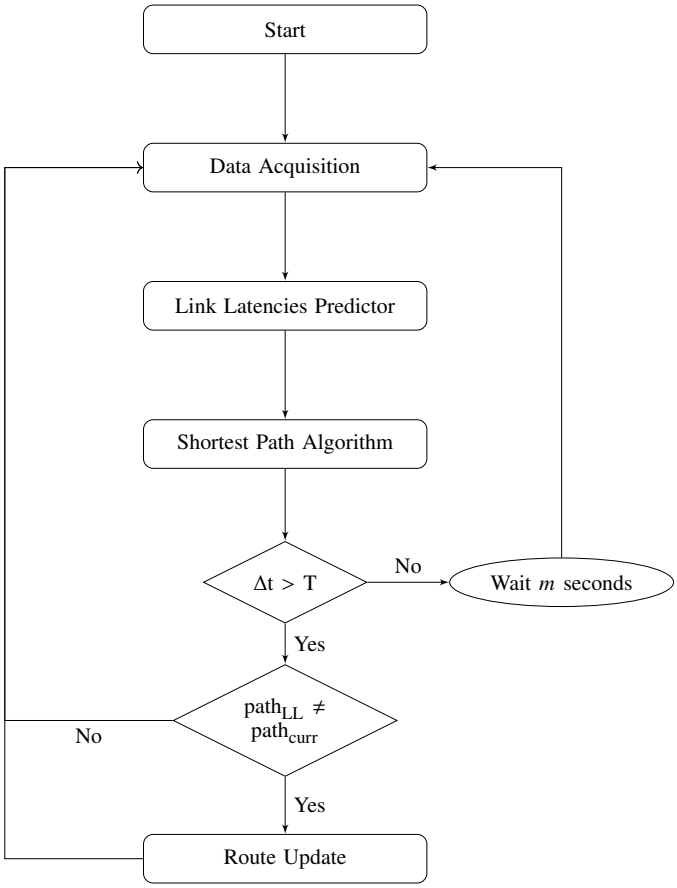

Fig. 2. Predictive path routing algorithm for low-latency flow chart.

IV. Predictive Path Routing Algorithm

This section presents the predictive path routing algorithm for low-latency, tested through the experimental testbed. The algorithm aims at predicting QoS metrics to select or update the routing path proactively. The flow chart of the algorithm is presented in Fig. 2.

This flow chart presents the following parameters, whose values need to be defined:

- m is the number of seconds configuring the update rate of network metrics in the database and the mathematical models, updating forecasts of path metrics with a Link Latencies Predictor and finding routing paths bringing low latency (path_{LL}) with a shortest path algorithm.
- T defines the timeout in minutes establishing how often the routing update is evaluated if the path providing lower latency (path_{LL}) has changed from the current routing setup ($\text{path}_{\text{curr}}$).
- Δt is the elapsed time t from the last evaluation.

$$\Delta t = t - t_{LastEval} \tag{1}$$

Going into details of the evaluated predictive path routing algorithm, Algorithm 1, the link latencies predictor used is the matricial autoregressive model (MAR Model) due to both the input for feeding this model, and the output of the predictor is a matrix. The shortest path algorithm selected and implemented in the predictive routing path algorithm is the Dijkstra algorithm due to that this algorithm calculates the shortest path in a graph given a known source and destination.

The first step is to initialize all the variables related to the algorithm, such as the time scheduled for the subsequent routing evaluation, the traffic source and destination, in order to calculate the critical path and the current routing path, which is initially defined according to the default routing rules of the deployed topology. The initialized parameters are:

- $\text{t}_{\text{NextEval}}$ is the scheduled time in which the subsequent route evaluation is performed.
- $\text{path}_{\text{curr}}$ is the default routing path for the topology.
- src is the traffic source of the network topology.
- dst is the traffic destination of the network topology.

Then, the algorithm executes the process and decision blocks shown in the flow chart of Fig. 2. The overall workflow is composed of these steps:

Step 1: The first step is to construct the hyper-matrix (HipMat_{L}) with the latency metrics in order to input the matrix into the link latencies predictor.

Step 2: Then, the next step is to calculate the predicted latencies between different adjacent links from the adjacency matrix ($\widehat{\text{Mat}}_{\text{L}}$).

Step 3: The third step is to calculate the shortest path routing ($\text{path}_{\text{curr}}$), feeding the shortest path algorithm with the predicted link latencies adjacency matrix, the traffic source and the destination.

Step 4: The next step is to wait for the next evaluation time $\text{t}_{\text{NextEval}}$. Otherwise, the algorithm keeps capturing metrics for n seconds and evaluates mathematical models with the new data.

Step 5: As far as the scheduled time for evaluating the routing change comes, the current ($\text{path}_{\text{curr}}$) and shortest routing paths (path_{LL}) are compared. The algorithm does not update the topology routing rules if both routing paths are equal. But if both routing paths are different, the topology routing rules are updated. In both cases, the $\text{t}_{\text{NextEval}}$ is updated by adding the timeout T.

All the blocks that compose the predictive path routing algorithm for low latency are presented and individually explained in the following sections.

A. Data Acquisition

A network topology can be easily associated with a finite graph. A graph [77] is a pair $G = (V, E)$, where V is a set of vertices and E is a set of paired vertices called edges. A graph may be fully specified by its adjacency matrix, which is an $n \times n$ square matrix, where n is the number of nodes of the graph, with A_{ij} specifying the number of connections from vertex i to vertex j. For a simple and unweighted graph, $A_{ij} \in \{0, 1\}$, indicating disconnection or connection, respectively. Furthermore, $A_{ii} = 0$ (as an edge can not start and end at the same vertex), as can be seen in Fig. 3.

Algorithm 1 Predictive path routing algorithm

```
function MAIN()
    t_NextEval ← t + T                          ▷ Timeout setup
    path_curr ← path_default                    ▷ Route init
    src ← S                                     ▷ Source init
    dst ← D                                     ▷ Destination init
    while true do                               ▷ Infinite loop
        HipMat_L ← DataAcquisition()
        Mat_L^ ← MARModel(HipMat_L)
        path_LL, dist ← Dijkstra(Mat_L^, src, dst)
        if t > t_NextEval then                  ▷ Every timeout
            if path_LL ≠ path_curr then
                RouteUpdate()                   ▷ Update rules
                path_curr ← path_LL             ▷ Update path
            else
                path_curr ← path_curr           ▷ Unchange path
            end if
            t_NextEval ← t + T                  ▷ Schedule evaluation time
        end if
        sleep (m)                               ▷ Wait m seconds
    end while
end function
```

Algorithm 2 Data acquisition algorithm

```
function DATAACQUISITION()
    for k ← t - N to t do                       ▷ Past events
        L ← GetLatency(k)                       ▷ Get metrics
        Mat_L ← L                               ▷ Adjacency matrix
        HipMat_L ← HipMat_L(Mat_L)              ▷ Hyper-matrix
    end for
    return HipMat_L
end function
```

Consider now a weighted graph, that is, a graph in which a certain weight is assigned to each edge. The adjacency matrix changes indicating now the *weight* in a direct connection of the graph, i.e. $A_{ij} \in \{0, weight\}$, indicating disconnection or connection metric respectively, meanwhile $A_{ii} = 0$.

This block of the algorithm presented in Algorithm 2 is responsible for reading the metrics from the database and composing the adjacency matrix associated with the virtualized network topology. The corresponding metrics we use for creating the adjacency matrix are the link latencies between nodes of the virtualized topology, so our adjacency matrix is composed of 0 and the latency of each edge in a direct connection of the topology. Moreover, it is responsible for creating the hyper-matrix of past history/records for link latencies, which are necessary for the predictor.

The N parameter is referred to the number of past samples needed by the predictor to forecast the following values accurately.

B. Matricial Autoregressive Model

The MAR model [78] is an analytical method to predict the values of a data matrix. In our case, the link latencies at the current time t are stored in an adjacency matrix (Mat_L). Those

(a) Square 2x2 Topology ⟶ (b) Adjacency matrix

Fig. 3. Adjacency matrix associated to an unweighted graph.

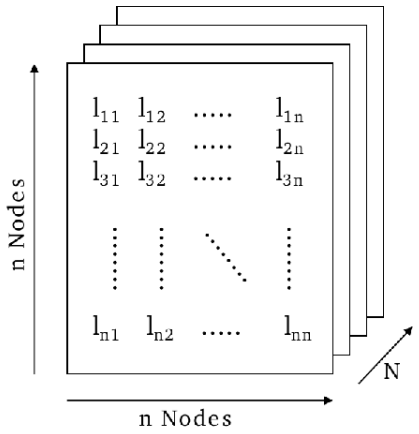

Fig. 4. Latencies hyper-matrix for the MAR model.

stored metrics get into a MAR model to predict the latencies at the next time $t + 1$. The first step is to acquire samples of N past time instants to feed the predictor. The predictor uses them to forecast the adjacency matrix of link latencies at the next scheduled time. The example shown in Fig. 4 depicts how the latencies samples compose a hyper-matrix where n is the number of nodes in the topology. The N is the number of samples that compose the HipMat_L.

The MAR model presented in [78] employs a bilinear structure as follows:

$$X_t = AX_{t-1}B^T + E_t. \tag{2}$$

The matrix resulting from the MAR processing ($\widehat{\text{Mat}_\text{L}}$) is also an adjacency matrix of the considered network topology, where the values correspond to the predicted link latencies. These predicted link latencies adjacency matrix is the input for the next block of the predictive routing algorithm, i.e., the Dijkstra algorithm.

C. Dijkstra Algorithm

Dijkstra algorithm [54] is an algorithm for finding the shortest path between nodes in a graph depending on the weight of the edges of the graph itself. The Algorithm 3, runs over each vertex and considers the weight of each edge as a measurement of the distance between vertices. Thus, it selects the neighbour vertex connected with the minimum distance to the current vertex. This operation is performed recursively to find the shortest path between the source (S) and destination (D) vertices.

First, the algorithm initializes two vectors:

Algorithm 3 Dijkstra algorithm

```
function DIJKSTRA(Mat_L, src, dst)
    dist[src] ← 0                                  ▷ Initialization
    for v in Lat_Mat do                            ▷ Visit all nodes
        if v ≠ src then
            dist[v] ← ∞                            ▷ Distance init
            path_short[v] ← empty                  ▷ Short path init
        end if
        Q ← Q.append(v)                            ▷ Add vertex in queue
    end for

    while Q is not empty do                        ▷ The main loop
        u ← Q.extract_min()                        ▷ Get vertex with minimum distance
        for each (v,u) do                          ▷ Vertex in queue
            d ← dist[u] + Mat_L[u][v]              ▷ Calculate d
            if d < dist[v] then
                dist[v] ← d                        ▷ Update dist
                path_short ← u                     ▷ Update minimum path
                Q ← Q.rmv(v)                       ▷ Remove visited vertex
            end if
        end for
    end while
    path_LL ← path_short                           ▷ Shorter provides Low Latency
    return dist[dst], path_LL
end function
```

- *dist*: It contains the minimum distances from the source to all possible destinations.
- $path_{short}$: It contains the vertices to be crossed from the source to each destination along the shortest path.

Then, the algorithm needs to run over all the graph vertices to calculate their distances from the source. For that, the weights to the neighbour vertices are captured and stored in *dist*. This is done in two steps:

1) Extract the neighbour vertex with the smallest distance from the vector *dist*.
2) Check the different distances between neighbour vertices and keep just the one with the lower weight.

Finally, both the value of *dist* with the minimum distance calculated from the shortest path given by $path_{short}$ and the shortest path itself are updated.

The use of this algorithm allows us to predict the shortest path between a source and a destination of the virtual network topology based on the lowest end-to-end predicted latency. The final objective is to update the routing rules based on that predicted shortest path.

D. Routing Update

The last block of the Predictive Path Routing Algorithm, is in charge of updating the routing rules inside the topology when needed. It is responsible for changing the current routing rules of the topology ($path_{curr}$) based on the shortest path given in this case by the Dijkstra algorithm ($path_{short}$). If the shortest path is different to the current one, the update of the topology routing process is activated to grant low latency ($path_{LL}$). Otherwise, the algorithm starts the flow again.

V. Validation

This section describes all the experiment validation, including the description of the employed setup, the performance metrics to analyze the topology routing and, finally, the obtained results.

A. Experimentation Setup

The testbed to carry out the experiment has several components, as shown in Fig. 5. On the one hand, a server runs an OSM instance which executes Day-1 and Day-2 actions through JUJU charms [79]. JUJU software is installed on the same server to create and update the topology routing. An OpenStack server is configured and managed by the OSM/JUJU server to deploy the necessary VNFs to create the desired topology. Moreover, on another server, a Prometheus database [46] and a Grafana data analytics tool [49] are configured in order to store the network metrics obtained from the topology and to analyze them at a visual level. A Prometheus PushGateway [80] system is also deployed on this server. Prometheus works in pull mode by design to retrieve the metrics from the monitored nodes. It means that Prometheus should discover the probed endpoints (IP addresses) to pull the metrics from the monitored nodes every time the routing configuration of the topology changes. To enable Prometheus to gather all the metrics even when the IPs configuration changes, we introduce an intermediate service such as Prometheus PushGateway. Then, Prometheus pulls the metrics from the Prometheus PushGateway, whose IP address is maintained even when topology or routing rules are updated. Finally, an external management computer executes a Python implementation of the predictive shortest path routing algorithm. Its objective is to predict the shortest routing path between the source and destination inside the topology from the compiled metrics. Table II shows the specifications of each component of the setup.

B. Deployment of Virtual Topologies

We have developed three components to deploy a network topology automatically.

- Square topology matrix engine generator
- Alternative routing descriptor generator
- Routing engine generator

The *square topology matrix generator* is an engine which creates a squared topology matrix with nodes (V) and edges (E) from a given description file. This system creates the appropriate instructions for OpenStack's orchestration engine to deploy a squared topology matrix of a given size.

The *alternative routing descriptor generator* is the component that, when a node does not have any routing rule in order to connect it inside the rest of the topology, creates routing rules for that node. Once that specific routing rule for each node has been created, the system loads it into the

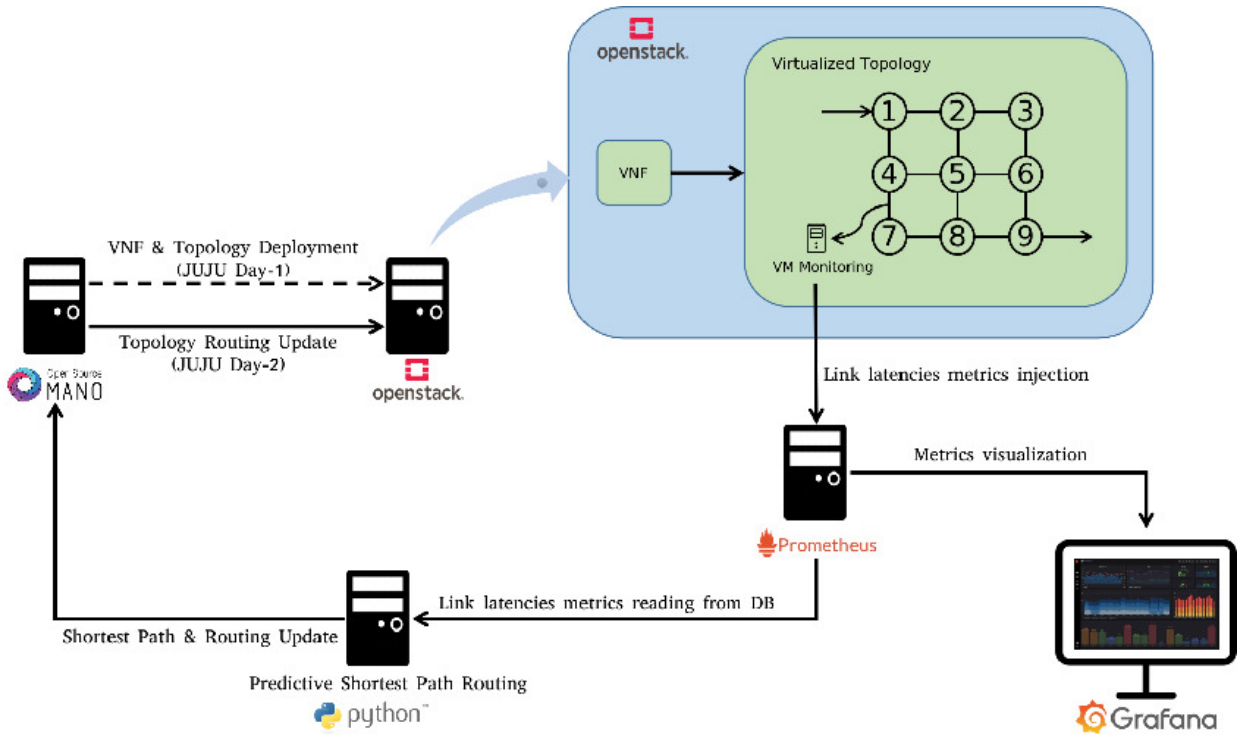

Fig. 5. Experimentation setup for virtualized network topologies deployment and routing.

TABLE II
TESTBED COMPONENTS SPECIFICATIONS.

Component	System specification
OSM/JUJU	OS: UBUNTU 18.04 RAM: 8 GB vCPU: 2 OSM version: 10.0.3 JUJU version: 2.8.13
OpenStack	Version: Victoria Intel(R) Xeon(R) Gold 6230R 26 CPUs 192 GB RAM
Grafana/Prometheus	OS: UBUNTU 18.04 RAM: 4 GB vCPU: 2 Prometheus version: 2.2.20 Grafana version: 8.4.3
Predictive path routing	OS: UBUNTU 20.04 RAM: 16 GB CPU cores: 6 Python version: 3.8.10

routing engine generator to generate appropriate instructions to OpenStack's orchestrator to implement it. This way, all the nodes, even when they are part of the shortest path or not used for traffic forwarding, are still available, connected and pushing inactivity metrics. The routing to be applied is defined by a description file employed as the input for the *routing engine generator*.

The *routing engine generator* is the component which creates appropriate instructions to route traffic between nodes

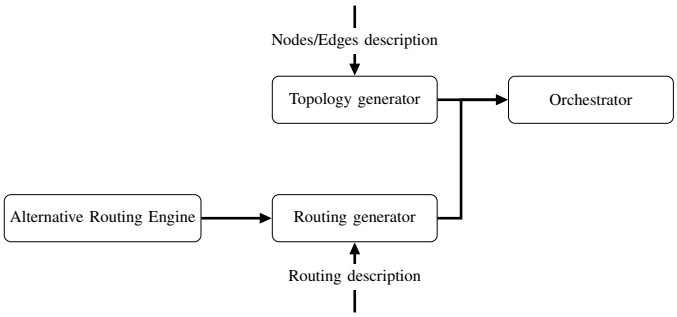

Fig. 6. Initial topology setup.

deployed into the *Topology matrix* of a given size. Like in the former step, those instructions are commanded into the OpenStack's Orchestration engine.

This workflow allows virtual topologies to be deployed in an initialization and reconfiguration strategy gaining flexibility for network management experiments. A default virtual topology is deployed in the initialization step, Fig. 6.

Accordingly, the squared topology matrix is defined and sent to the *topology generator* that creates a set of instructions for the *orchestrator*. An initial routing description is created and sent to the *routing generator* to create another set of instructions for the *orchestrator*. Based on the initial routing description, the *alternative routing engine* creates a subset of routing descriptions for the nodes excluded from the initial routing description and sends them to the *routing generator* in order to create a complete instruction set for the *orchestrator*.

When a virtual topology needs to be updated, as shown in Fig. 7, the component called *predictive routing path* for low-latency sends a new routing description to both the

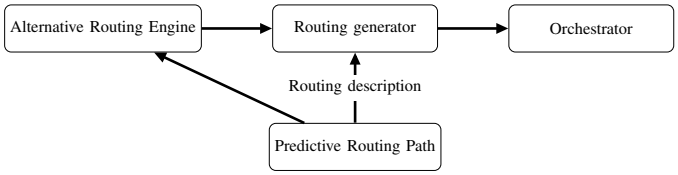

Fig. 7. Routing update.

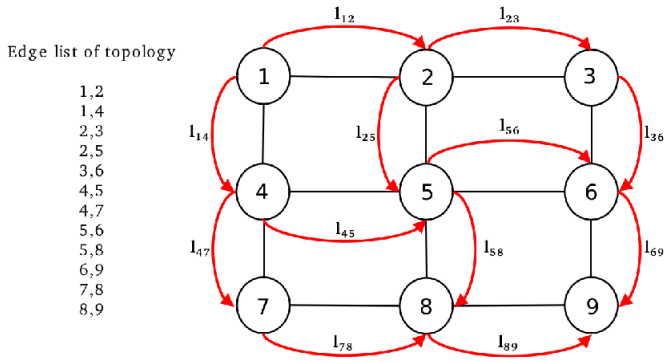

Fig. 8. Client-server definition order of each monitoring VM of each node of the topology.

router generator and the *alternative routing engine*. Here, the *routing generator* creates a subset of instructions from the new routing description and from the alternative routing description produced from the *alternative routing engine* and sends them to the *orchestrator* in order to execute new routing commands into the OpenStack virtual network infrastructure.

C. Performance Metrics

The metrics used in the experiment are the link latencies of all the nodes in the virtualized topology. The tool *Qperf* [81] is being used in each VM of the virtualized topology to assess the values of such metrics. Those metrics are sent to Prometheus. The Prometheus Time Series database stores latency metrics along the tests. Every time a network link is created in the topology, a VM will be automatically deployed and start measuring the performance of the link.

The configuration of *Qperf* command in each VM of each node of the topology, i.e. which nodes act as servers and which nodes act as clients which send metrics to the database, is defined and presented below.

The order of the *edges* (E) for a 3×3 grid topology, as it is shown in Fig. 8, also implies the roles meaning which nodes act as servers and which nodes act as clients for the automation of the metrics probing and sending to the central database. Nodes of the first column of the file act as clients of the edges, while nodes of the second column of the file act as servers of the edges when configuring the *Qperf* setup. The *Qperf* client VM sends the link latency to Prometheus every 5 seconds. Moreover, for the unequivocal processing of the link latencies captured at each edge, usually, as some network topologies can have more edges than nodes, both the source and the destination of the monitored link's metrics are also sent to the Prometheus API. Thus, any query from the predictive path routing algorithm for low-latency retrieves the map of latencies to the edges.

TABLE III
RMSE AND MAE OF END TO END AVERAGE LATENCY ANALYSIS FOR SELECTING THE BEST NUMBER OF PAST SAMPLES N FOR THE PREDICTOR.

N	RMSE	MAE
9	0.285 ms	0.144 ms
10	0.251 ms	0.124 ms
11	0.240 ms	0.122 ms

D. Application of Routing Update

This section presents how the predictive path routing algorithm performs the routing update of the virtualized topology for low latency once the virtualized topology is deployed in OpenStack and the definition of the parameters presented in Section IV.

As it is explained, the MAR predictor needs some past events samples for predicting the link latencies for the next period time $t + 1$. The number of past samples N in the predictive routing algorithm is set to $N = 10$. The predictor's accuracy is analyzed concerning the number of past samples needed to forecast the link latencies in $t + 1$. Suppose the number of samples is much lower than 10, i.e. $N = 2$. In that case, the predictor is inaccurate with an RMSE of end-to-end average latency, calculated over all possible paths, equal to 181.455 and an MAE equal to 20.13. If the same analysis is made with N close to 10, both the RMSE and MAE decrease significantly, getting the results shown in Table III.

As seen in the table, setting the employed past samples to a number higher than ten does not make any significant difference in terms of the accuracy of the forecast. The time parameters m and T, explained in Section IV, are now defined. On the one hand, the window of time in seconds (m) applied in the predictive path routing algorithm for low latency is defined as ten seconds. On the other hand, the time in minutes T, which establishes how often new routing rules are evaluated, is set to five minutes so that, every T minutes, the algorithm is going to compare the current route with the candidate shortest one. It has been set to five minutes because OpenStack needs some time to establish the routes in each router, so if every time the optimal route does not match the current route changed, the system would be all the time changing routes, and the system needs to stabilize once the routing rules have been changed. Thus, OpenStack can change the *qrouter's* routing rules of the virtualized topology correctly before evaluating both the optimal and current routing paths again.

A comparison of different SP algorithms is now analyzed. As described in Section IV, the SP algorithm employed in this experiment is the Dijkstra one. The Table IV shows the differences between other shortest path algorithms. The computation time is the time elapsed to perform the algorithm processing when a new evaluation over time is triggered. This is the most important parameter to consider when deciding the algorithm to implement in our experiments. As the Dijkstra algorithm, is clearly the most efficient one, in terms of computation time, is the one chosen. Furthermore, Dijkstra

TABLE IV
SHORTEST PATH ALGORITHMS COMPARISON.

SP algorithm	Graph type	Edge weight sign	Computation time	Output
Dijkstra	Directed/Undirected	Positive	0.05769 ms	Shortest path
Bellman Ford	Directed/Undirected	Positive/Negative	0.12373 ms	Shortest path
Floyd Warshall	Directed	Positive/Negative	0.20456 ms	Shortest path
A*	Directed/Undirected	Positive	0.25916 ms	Shortest path
Johnson	Directed	Positive/Negative	0.54250 ms	Shortest path

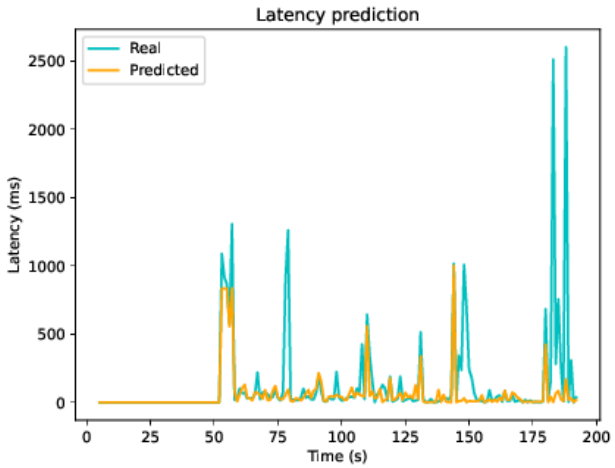

Fig. 9. End-to-end average latency through all possible nodes prediction of the MAR predictor.

is compliant with the graph type planned to use in the tests, where routing of traffic can be bidirectional. This means a the algorithm needs to work with undirected graphs, discarding some algorithms of Table IV.

The predictive routing algorithm calculates the shortest routing path, bringing lower latency. If the result is different from the current route, the algorithm sends an OSM/JUJU Day-2 action to the VNF deployed in OpenStack to change the current route to the new one. In that OSM/JUJU Day-2 action, the input for the routing update is the shortest routing path calculated from the Dijkstra algorithm from the link latencies forecasted by the MAR predictor.

E. Results

This section shows the results obtained during the experiment for predicting low-latency path routes from a source to a destination in different topologies.

The first results, shown in Fig. 9, are related to the accuracy of the MAR predictor when forecasting the average end-to-end latency across all possible paths in a given network topology.

It is observed that the predictor detects some network saturation peaks in terms of the average end-to-end latency of all the paths from source to destination. The predictor cannot predict some significant changes tied to spontaneous saturation.

Concerning the validation of the prediction, we present another result in Fig. 10. Here, it can be observed the influence in the network latency under edge saturation conditions when changing the routing rules with and without prediction. This graph shows two saturation zones in which the network suffers a traffic injection in two different edges of the network. It can be seen that the routing path change is more efficient when the prediction is applied instead of changing the routing path without prediction in bot saturation zones A and B.

TABLE V
END-TO-END AVERAGE LATENCY AND SHORTEST PATH LATENCY DEPENDING ON TOPOLOGY COMPLEXITY.

Nodes	Edges	Average latency	Lowest latency
4	4	2.4798 ms	0.1274 ms
9	12	11.0935 ms	0.5481 ms
16	24	28.5481 ms	1.5498 ms

The influence of adding nodes and edges in a topology in terms of latency improvement is now analyzed. As it can be seen in Table V, it is compared the end-to-end average latency of all the paths from a source to a destination in the topology. When applying the low latency paths to routing rules, shorter paths are ensured from a source to a destination.

Fig. 11 shows the end-to-end latency improvement from Table V. Here, the blue line is referred to the average latency of all the paths from a source to a destination, and the black line is referred to the end-to-end shortest path gaining low latency.

It is observed that the higher the complexity of the topology, i.e. higher number of nodes and edges in the topology, the more improvement we have on calculating the shortest routing path, based on the end-to-end latency metrics when applying the predictive algorithm.

In terms of latency improvement, the impact of saturating an edge in topology is discussed below. Considering a 9-node closed square network topology with 12 edges, where each link is limited to 3000 kbps, the link latencies determine the weight of each edge of the network topology for constructing the adjacency matrix at time *t*. If the source of the data packets is set to node 11 and the destination is set to node 9, there are 12 possible paths for the data packets to flow from source to destination. One of those possible paths is the shortest one, with the lowest end-to-end latency. In the initialization of the topology route setup to perform the necessary tests, a routing rule that connects *1,2,5,8,9* nodes has been set. When edge 2–5 of the topology has a traffic bottleneck due to flooding with UDP traffic through the *ffmpeg*, [82] tool, both the latencies of that edge and the adjacency edges are affected. This results in a significant increase in the latencies of those edges. While the edges not adjacent to the 2–5 edge are not affected by the ffmpeg's UDP flow, as shown in Fig. 12.

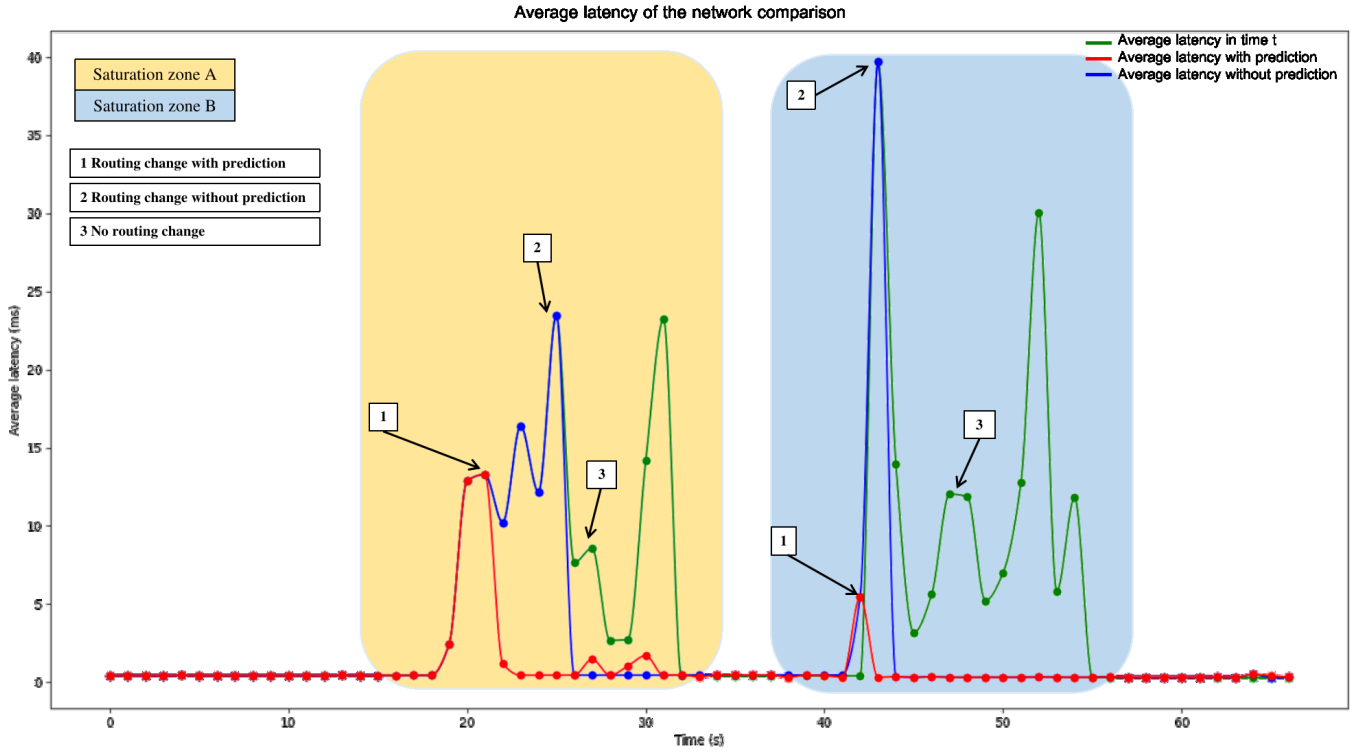

Fig. 10. Network average latency in terms of changing its routing rules with or without prediction.

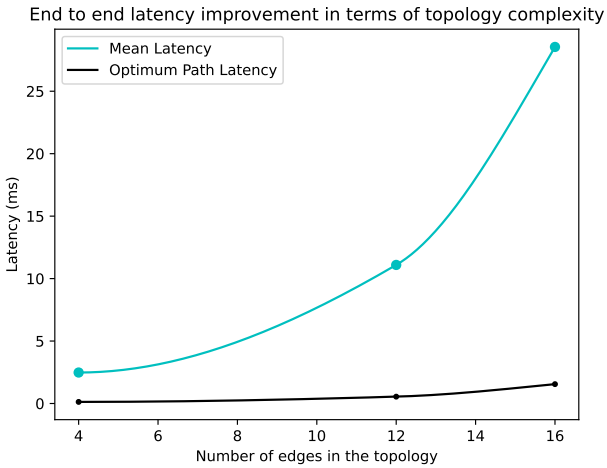

Fig. 11. Topology complexity influence on the end-to-end latency improvements.

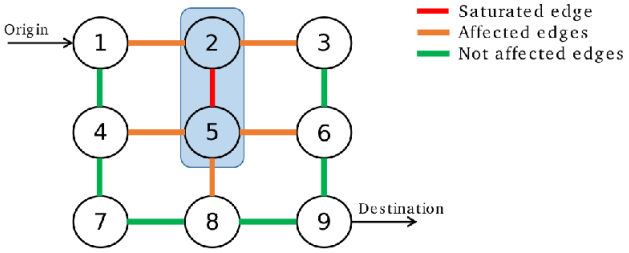

Fig. 12. Affected edges due to the saturation of edge between nodes 2 and 5 of the topology.

To overview all the results for mechanism stages, Fig. 13 summarizes latency performance for three different zones.

- In zone A, there is no network saturation, and the data packets cross the nodes *1,2,5,8,9* being the end-to-end current and shortest path latency similar.
- Zone B is affected due to the UDP flooding saturation on edge 2–5 of the virtualized topology. Therefore, the end-to-end latency of the current path is increased. When the predictive routing algorithm detects the network saturation, it selects the shortest routing path avoiding the saturated edge of the topology. In this case, the path that connects the nodes *1,4,7,8,9* is the shortest one along the edge 2–5 is being saturated. Once the routing path of the topology is updated, the current route and the shortest one coincide.
- The last zone, C, represents that even when the edge 2–5 is still saturated, the current route of the topology across the nodes *1,4,7,8,9* is not being affected by performing a minimum end-to-end latency.

As a final result, the continuity of the network reconfiguration is analyzed. While updating the routing rules and applying actions and file descriptors, the virtual network has no connectivity, but once the orchestrator completes the configuration, the network is again forwarding packets. This reason is a significant limitation of management technologies of virtual networks.

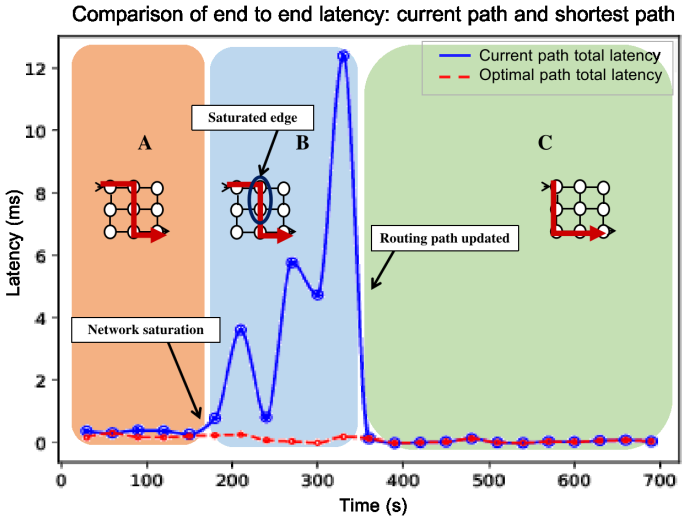

Fig. 13. Comparison of end to end latency: Current and shortest path.

VI. Discussion and Open Challenges

Monitoring systems enable advances in automated network management, log reporting from software functions and standard interfaces for programmed operation and interoperable integration. It is evident that software and the virtual nature of network functions are pushing the network limits demanded by specific applications and specialized scenarios. The blurred frontiers between applications and networks brought by 5G network technologies and widely employed software frameworks favour the progress and breakthroughs from a bilateral perspective.

The networks target challenging application domains, heavily dominated by multimedia streaming protocols, such as cooperative, connected and automated mobility (CCAM) or demanding immersive applications such as holographic and extended reality (XR) or multi-party virtual reality, which meta worlds expect to create. In these scenarios, mobility and daily human life cycles can shape traffic patterns around mutable distributions, which network topologies need to bridge dynamically.

The network topology can mutate to accommodate a traffic demand better [83]. However, the transition impacts the ongoing sessions challenging to minimize or absorb, making this option ideal for long-term traffic distribution. Instead, for prompter reactions or quicker preventions to changeable traffic demands, which could produce bandwidth bottlenecks or late delivery, the automated, programmed routing policies can make the difference.

To implement this feature, different technologies and software frameworks can be employed. However, different implementation limits make some technology candidates, listed in Table I, challenging to apply dynamic and programmable routing configurations, bringing significant scalability overheads or lacking interoperability between them the run out of the box. The analyzed technologies are not able to immediately apply a new routing configuration or to transparently and seamlessly deploy or retire virtual router instances. So, the need to smoothly modify configurations or allocate resources is a significant concern where artificial intelligence algorithms trained to penalty situations that damage QoS or require session reboot could come into play.

Our solution brings some underperformance for highly symmetric topologies and a limited ability to forecast situations in advance. The ability to create light and flexible algorithms that can quickly scale for higher network cardinality and accurately respond to previously unseen situations is something where reinforced learning techniques could foster the self-organizing network leap. To avoid underperformance with some virtualized network topologies, the network analysis performed with the predictive path routing algorithm for low latency will focus on the affected edges of the network.

Additionally, the measurement and monitoring systems take too many resources as we need to actively get the link performance even when no application traffic is there. This implies an unnecessary overhead that could be probed directly from the traffic if the monitoring systems would work as the OSM and OpenStack systems promised.

The last aspect that would mean a significant feature is reducing deployment times of routing policies and applying the dynamic rules. The proposed solution takes time in minutes, with equivalent scores to instantiating routers with a specific configuration through OpenStack or Kubernetes. However, this is mainly limited by OSM and JUJU, which will be more easily reduced than the container lifecycle times.

VII. Conclusion

This paper evaluates, identifies and analyzes the different implemented solutions to perform experiments on network management. For this purpose, an innovative experimental testbed and a predictive routing path algorithm are proposed.

For the evaluation of the proposed approaches, OpenStack has been used in order to deploy virtualized topologies through its orchestration service.

The obtained results from the proposed approaches show that the testbed proposed using OpenStack as the primary technology for deploying the virtualized topology is functionally obtaining the possibility of deploying any virtualized topology on top of it. Moreover, it can be seen that the implementation of our Predictive Path Routing for low latency detects the saturated edge of the topology and calculates the shortest path to change the routing rules of the deployed topology to improve the end-to-end topology latency.

Future lines of research should focus on the reduction in terms of the time of both the deployment and the changing of the topology routing rules, the introduction of real-time traffic in order to saturate any edge of the topology, the possibility of integrating containers instead of VMs for the network monitorization and the improvement of the predictive path routing algorithm for low-latency for analyzing the only the affected part of the network.

References

[1] H. Yu, H. Lee, and H. Jeon, "What is 5G? Emerging 5G mobile services and network requirements," *Sustainability*, vol. 9, no. 10, p. 1848, 2017.

[2] C. Bouras, P. Ntarzanos, and A. Papazois, "Cost modeling for SDN/NFV based mobile 5G networks," in *Proc. IEEE ICUMT*, 2016.

[3] E. Hernandez-Valencia, S. Izzo, and B. Polonsky, "How will NFV/SDN transform service provider opex?" *IEEE Netw.*, vol. 29, no. 3, pp. 60–67, 2015.

[4] O. G. Aliu, A. Imran, M. A. Imran, and B. Evans, "A survey of self organisation in future cellular networks," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 1, pp. 336–361, 2012.

[5] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[6] R. Mijumbi *et al.*, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. surveys tuts.*, vol. 18, no. 1, pp. 236–262, 2015.

[7] ETSI, "Network functions virtualisation; ecosystem; report on SDN usage in NFV architectural framework," *Technical report, ETSI, Tech. Rep.*, 2015.

[8] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *ACM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[9] R. Enns, M. Bjorklund, and J. Schoenwaelder, "NETCONF configuration protocol," RFC 4741, December, Tech. Rep., 2006.

[10] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. ACM HotSDN*, 2014.

[11] "Opendaylight," https://www.opendaylight.org/, [Online; accessed 03-January-2022].

[12] F. Tomonori, "Introduction to ryu SDN framework," *Open Netw. Summit*, pp. 1–14, 2013.

[13] R. Wallner and R. Cannistra, "An SDN approach: Quality of service using big switch's floodlight open-source controller," in *Proc. APAN*, 2013.

[14] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "SDN controllers: A comparative study," in *Proc. IEEE MELECON*, 2016.

[15] B. Pfaff *et al.*, "Extending networking into the virtualization layer." in *Proc. Hotnets*, 2009.

[16] B. Pfaff *et al.*, "The design and implementation of Open vSwitch," in *Proc. NSDI*, 2015.

[17] V. Gupta, K. Kaur, and S. Kaur, "Developing small size low-cost software-defined networking switch using raspberry pi," in *Proc. CSI-2015*, 2018.

[18] E. Upton and G. Halfacree, *Raspberry Pi user guide*. John Wiley & Sons, 2014.

[19] P. R. Srivastava and S. Saurav, "Networking agent for overlay L2 routing and overlay to underlay external networks L3 routing using OpenFlow and Open vSwitch," in *Proc. IEEEAPNOMS*, 2015.

[20] O. vSwitch, "Open vSwitch docker hub," https://hub.docker.com/r/openvswitch/ovs, 2008, [Online; accessed 20-December-2021].

[21] K. Kaur, J. Singh, and N. S. Ghumman, "Mininet as software defined networking testing platform," in *Proc. ICCCS*, 2014.

[22] H. Zhang and J. Yan, "Performance of SDN routing in comparison with legacy routing protocols," in *Proc. CyberC*, 2015.

[23] S. Sendra *et al.*, "Including artificial intelligence in a routing protocol using software defined networks," in *Proc. IEEE ICC*, 2017.

[24] F. Benamrane *et al.*, "An east-west interface for distributed SDN control plane: Implementation and evaluation," *Comput. Electr. Eng.*, vol. 57, pp. 162–175, 2017.

[25] H. Yu, K. Li, H. Qi, W. Li, and X. Tao, "Zebra: An east-west control framework for SDN controllers," in *Proc, ICPP*, 2015.

[26] B. Almadani, A. Beg, and A. Mahmoud, "DSF: A distributed sdn control plane framework for the east/west interface," *IEEE Access*, vol. 9, pp. 26735–26754, 2021.

[27] ETSI, "Network functions virtualisation (NFV) release 3; management and orchestration; Vi-Vnfm reference point - interface and information model specification," *Technical report, ETSI, Tech. Rep.*, 2018.

[28] ETSI, "Network functions virtualisation (NFV) release 3; management and orchestration; Or-Vnfm reference point - interface and information model specification," *Technical report, ETSI, Tech. Rep.*, 2018.

[29] ESTI, "OpenVIM," https://osm.etsi.org/gitweb/?p=osm/openvim.git, [Online; accessed 03-January-2022].

[30] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42, 2012.

[31] "Open infrastructure foundation," https://openinfra.dev/, [Online; accessed 03-January-2022].

[32] T. Sechkova, M. Paolino, and D. Raho, "Virtualized infrastructure managers for edge computing: Openvim and openstack comparison," in *Proc. IEEE BMSB*, 2018.

[33] "Open source mano (OSM)," https://osm.etsi.org/, [Online; accessed 03-January-2022].

[34] "Open network automation platform (ONAP)," https://www.onap.org/, [Online; accessed 03-January-2022].

[35] S. Gallagher, *VMware private cloud computing with vCloud Director*. John Wiley & Sons, 2013.

[36] G. M. Yilma, Z. F. Yousaf, V. Sciancalepore, and X. Costa-Perez, "Benchmarking open source NFV MANO systems: OSM and ONAP," *Comput. Commun.*, vol. 161, pp. 86–98, 2020.

[37] N. F. S. De Sousa, D. A. L. Perez, R. V. Rosa, M. A. Santos, and C. E. Rothenberg, "Network service orchestration: A survey," *Comput. Commun.*, vol. 142, pp. 69–94, 2019.

[38] Cisco, "Network services orchestrator data sheet - cisco," https://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/network-services-orchestrator/datasheet-c78-734576.html, [Online; accessed 03-January-2022].

[39] Ericsson, "Ericsson network manager," https://www.ericsson.com/en/portfolio/digital-services/automated-network-operations/network-management/network-manager, [Online; accessed 03-January-2022].

[40] ZTE, "Cloudstudio nfvo," https://www.zte.com.cn/global/products/core_network/201903151447/201707261124, [Online; accessed 03-January-2022].

[41] ZTE, "Cloudstudio VNFM," https://www.zte.com.cn/global/products/core_network/201903151447/201707261125, [Online; accessed 03-January-2022].

[42] ETSI, "Network functions virtualisation (NFV); assurance; report on active monitoring and failure detection," *Technical report, ETSI, Tech. Rep.*, 2016.

[43] C. Yu *et al.*, "Flowsense: Monitoring network utilization with zero measurement cost," in *Proc. PAM*, 2013.

[44] R. Hofstede *et al.*, "Flow monitoring explained: From packet capture to data analysis with netflow and ipfix," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 4, pp. 2037–2064, 2014.

[45] H. Kim, S. Yoon, H. Jeon, W. Lee, and S. Kang, "Service platform and monitoring architecture for network function virtualization (NFV)," *Cluster Comput.*, vol. 19, no. 4, pp. 1835–1841, 2016.

[46] Prometheus. [Online]. Available: https://prometheus.io/

[47] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi, "Time series databases and influxdb," *Studienarbeit, Université Libre de Bruxelles*, p. 12, 2017.

[48] C. Gormley and Z. Tong, *Elasticsearch: The definitive guide: A distributed real-time search and analytics engine*. "O'Reilly Media, Inc.", 2015.

[49] M. Chakraborty and A. P. Kundan, "Grafana," in *Monitoring Cloud-Native Appl.*, pp. 187–240, 2021.

[50] Y. Gupta, *Kibana essentials*. Packt Publishing Ltd, 2015.

[51] J. W. Guck, A. Van Bemten, M. Reisslein, and W. Kellerer, "Unicast QoS routing algorithms for SDN: A comprehensive survey and performance evaluation," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 388–415, 2017.

[52] D. Wischik, M. Handley, and C. Raiciu, "Control of multipath TCP and optimization of multipath routing in the Internet," in *Proc. NET-COOP*, 2009.

[53] R. G. Garroppo, S. Giordano, and L. Tavanti, "A survey on multi-constrained optimal path computation: Exact and approximate algorithms," *Comput. Netw.*, vol. 54, no. 17, pp. 3081–3107, 2010.

[54] J.-C. Chen, "Dijkstra's shortest path algorithm," *J. Formalized Math.*, vol. 15, no. 9, pp. 237–247, 2003.

[55] K. Magzhan and H. M. Jani, "A review and evaluations of shortest path algorithms," *Int. J. Sci. Technol. Research*, vol. 2, no. 6, pp. 99–104, 2013.

[56] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah, "A survey of shortest-path algorithms," *arXiv preprint arXiv:1705.02044*, 2017.

[57] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *Proc. IEEE COLCOM*, 2014.

[58] M. Peuster, J. Kampmeyer, and H. Karl, "Containernet 2.0: A rapid prototyping platform for hybrid service function chains," in *Proc. IEEE NetSoft*, 2018.

[59] P. Wette *et al.*, "Maxinet: Distributed emulation of software-defined networks," in *Proc. IFIP Networking*, 2014.

[60] M. Peuster, H. Karl, and S. van Rossem, "Medicine: Rapid prototyping of production-ready network services in multi-pop environments," in *Proc. IEEE NFV-SDN*, 2016.

[61] C. Rotter *et al.*, "Using linux containers in telecom applications," in *Proc. ICIN*, 2016.

[62] "Amazon ec2," https://aws.amazon.com/ec2/, [Online; accessed 03-January-2022].
[63] "Google cloud compute engine," https://cloud.google.com, [Online; accessed 03-January-2022].
[64] "Azure," https://azure.microsoft.com/, [Online; accessed 03-January-2022].
[65] Prometheus. [Online]. Available: https://openstack.org/
[66] "Eucalyptus cloud," https://eucalyptus.cloud/, [Online; accessed 03-January-2022].
[67] "Opennebula," https://opennebula.io, [Online; accessed 03-January-2022].
[68] S. Yadav, "Comparative study on open source software for cloud computing platform: Eucalyptus, openstack and opennebula," *Int. J. Eng. Sci.*, vol. 3, no. 10, pp. 51–54, 2013.
[69] R. Kumar, N. Gupta, S. Charu, K. Jain, and S. K. Jangir, "Open source solution for cloud computing platform using openstack," *Int. J. Comput. Sci. Mobile Comput.*, vol. 3, no. 5, pp. 89–98, 2014.
[70] "Opennebula," https://wiki.openstack.org/wiki/Heat, [Online; accessed 03-January-2022].
[71] J. Denton, *Learning OpenStack Networking (Neutron)*. Packt Publishing Ltd, 2014.
[72] O. Tkachova, M. J. Salim, and A. R. Yahya, "An analysis of SDN-OpenStack integration," in *Proc. PIC S&T*, 2015.
[73] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven SDN controller architecture," in *Proc. WoWMoM*, 2014.
[74] B. Yi *et al.*, "A comprehensive survey of network function virtualization," *Comput. Netw.*, vol. 133, pp. 212–262, 2018.
[75] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of virtual switching with open vSwitch: A quantitative analysis," *J. Netw. Syst. Manag.*, vol. 26, no. 2, pp. 314–338, 2018.
[76] A. A. Siddiqui, *OpenStack Orchestration*. Packt Publishing Ltd, 2015.
[77] R. Diestel, "Extremal graph theory," in *Graph Theory*. Springer, 2017, pp. 173–207.
[78] R. Chen, H. Xiao, and D. Yang, "Autoregressive models for matrix-valued time series," *J. Econometrics*, vol. 222, no. 1, pp. 539–560, 2021.
[79] C. Butler, "Automating orchestration in the cloud with Ubuntu Juju," in *Proc. UCMS*, 2014.
[80] Prometheus. Prometheus pushgateway. [Online]. Available: https://github.com/prometheus/pushgateway
[81] Qperf. Qperf. [Online]. Available: https://linux.die.net/man/1/qperf
[82] Ffmpeg. Ffmpeg. [Online]. Available: https://ffmpeg.org/
[83] A. Martin *et al.*, "Network resource allocation system for QoE-aware delivery of media services in 5G networks," *IEEE Trans. Broadcast.*, vol. 64, no. 2, pp. 561–574, 2018.

Juncal Uriol is with the Department of Digital Media, Vicomtech. She received the Electronic Communications Engineering degree and the M. Sc. in Telecommunication Engineering in 2017 and 2019, respectively, in TECNUN University of Navarra. She is a Research Assistant at Vicomtech, where she works on projects concerning 5G virtualized networks.

Juan Felipe Mogollón is with the Department of Digital Media, Vicomtech, Spain. He received his Telecommunication Engineering degree in 2006 from Universidad de Cantabria, Spain. Former developer at Zitralia Security Solutions (October 2006 - June 2008). Currently, he is working at Vicomtech in multimedia services and 5G infrastructures projects.

Mikel Serón is with the Department of Digital Media in Vicomtech. He received his Telecommunications Engineering B. Sc and M. Sc. from University of the Basque Country in 2016 and 2022 respectively. He worked as an Internship Researcher (2018) in Telefónica R+D and as a System Engineer (2018-2021) in Talio Solutions. Since 2021 he is Research Assistant at Vicomtech, where he works on projects concerning 5G, MEC, virtualized networks and infrastructures.

Roberto Viola is with the Department of Digital Media, Vicomtech. He received his advanced degree in Telecommunication Engineering in 2016 from University of Cassino and Southern Lazio and his PhD degree in 2021 from University of the Basque Country. He is Research Associate at Vicomtech, where he works on projects concerning multimedia services and network infrastructures.

Ángel Martín is with the Department of Digital Media, Vicomtech. He received his PhD degree (2018) from UPV/EHU and his engineering degree (2003) from University Carlos III. He developed in Prodys an standard MPEG-4 AVC/H.264 codec for DSP (2003-2005). He worked in media streaming and encoding research (2005-2008) in Telefónica. He worked in the fields of smart environments and ubiquitous and pervasive computing (2008-2010) in Innovalia. Currently, he is on Vicomtech working in multimedia services and 5G infrastructures projects.

Mikel Zorrilla is head of the Digital Media department, Vicomtech. He received his Telecommunication Engineering degree (2007) from Mondragon Unibertsitatea, and an advanced degree (2012) and PhD degree (2016) in Computer Science from UPV/EHU. He has participated in many international research projects, such as MediaScape or Hbb4All European Projects. Previously he has held positions at IK4-Ikerlan as an Assistant Researcher (2002-2006) and at Deusto Business School (2014) as an Associate Professor in media.

Jon Montalbán received the M.S. Degree (2009) and PhD (2014) in Telecommunications Engineering from the University of the Basque Country (UPV/EHU). Since 2009 he is part of the TSR (Radiocommunications and Signal Processing) research group at UPV/EHU, where he is a Postdoctoral Researcher involved in several projects in the Digital Terrestrial TV broadcasting. His current research interests include digital communications and digital signal processing for mobile reception of broadband wireless communications systems in 5G.