Megalos: A Scalable Architecture for the Virtualization of Network Scenarios

Mariano Scazzariello, Lorenzo Ariemma, Giuseppe Di Battista, and Maurizio Patrignani
Roma Tre University – Rome, Italy

***Abstract*—We introduce an ETSI NFV compliant, scalable, and distributed architecture, called Megalos, that supports the implementation of virtual network scenarios consisting of virtual devices (VNFs) where each VNF may have several L2 interfaces assigned to virtual LANs. We rely on Docker containers to realize VNFs and we leverage Kubernetes for the management of the nodes of a distributed cluster. Our architecture guarantees the segregation of each virtual LAN traffic from the traffic of other LANs, from the cluster traffic, and from Internet traffic. Also, a packet is only sent to the cluster node containing the recipient VNF. The allocation of the VNFs to the nodes of the cluster is performed by Megalos Scheduler, taking into account the network topology in order to reduce the traffic among nodes. We produce an example application where we emulate a large network scenario, with thousands of VNFs and LANs, on a small cluster of 50 nodes. Finally, we experimentally show the scalability potential of Megalos by measuring the overhead of the distributed environment and of its signaling protocols.**

***Index Terms*—NFV, Kubernetes, containers, orchestrators**

I. Introduction

One of the main trends in the Internet Service Providers (ISPs) ecosystem is to move several functions performed by their managed networks to a virtual environment. This goes under the name of Network Function Virtualization (NFV). To give some examples: firewall, NAT, and even fully-fledged routing services quite often move from network devices landing into general-purpose hardware. This tendency keeps growing fast due to the reduced costs that a virtual network has in terms of deployment, scalability, and maintainability and due to the presence on the market of cloud providers that allow to run software functions into thousands of virtual machines or containers in their clouds.

Implementing (even partially) a complex network architecture in terms of the NFV paradigm requires the support of tools that can be used to transform the description of a complex network scenario into a software architecture composed of several interacting virtual entities. This pushes an awesome effort of standardization and implementation.

However, the currently available platforms that allow to implement network functions into a cloud environment have the following limitations. (i) They usually focus only on the virtualization of specific network functions rather than allowing the virtualization of an entire network device. (ii) They do not allow massive virtualization activities, where large portions of an existing network have to become virtual in just one step on an as-is basis, i.e. preserving their interconnections and address plans. (iii) They use scheduling facilities to assign Virtual Network Functions (VNFs) to cluster nodes that are mostly aimed at balancing the workload on the nodes and do not take into account the network topology in order to reduce the traffic among nodes.

978-1-7281-4973-8/20/$31.00

We present Megalos (from the greek word $\mu\varepsilon\gamma\acute{\alpha}\lambda o\varsigma$ that stands for "big" or "large"). It is an architecture that overcomes the above limitations and can be used to virtually implement complex network scenarios. Megalos has the following main features. (1) It is ETSI NFV compliant, scalable, and distributed. (2) It supports the implementation of complex virtual network scenarios consisting of several virtual devices (VNFs) where each VNF may have several L2 interfaces assigned to virtual LANs. (3) It relies on Docker containers and leverages Kubernetes for the management of the nodes of a distributed cluster. (4) It guarantees the segregation of each virtual LAN traffic from the traffic of other LANs, from the cluster traffic, and from Internet traffic exploiting Virtual Extensible LAN (VXLAN) and EVPN BGP. (5) The allocation of the VNFs to the nodes of the cluster is performed by schedulers that take into account the network topology.

The paper is organized as follows. Sec. II discusses the state-of-the-art related to Megalos. In Sec. III we model our container-based virtual network and its scalability requirements. In Sec. IV we illustrate the Megalos architecture. In Sec. V we propose the usage of two distinct schedulers. Sec. VI illustrates several use cases involving large network scenarios with thousands of VNFs and LANs. In Sec. VII we experimentally show the scalability potential of Megalos by showing the results of several experiments on a cheap cluster, including the measure of the overhead of the distributed environment and of its signaling protocols. Sec. VIII concludes the paper with final considerations and open problems.

II. Related Work

NFV [1] is currently standardized by the European Telecommunications Standards Institute (ETSI). In particular, we refer to the terminology and concepts introduced by the ETSI NFV architecture [2] (see Fig. 1). The literature on NFV is particularly rich. In [3] it is addressed the difficulty of managing network functions that are strongly integrated with the hardware: updating them implies purchasing new hardware while NFV aims to overcome this limitation. Several other surveys about NFV are available, such as [4]–[6].

Independently from the ETSI standard, several architecture have been proposed and developed for emulating networks

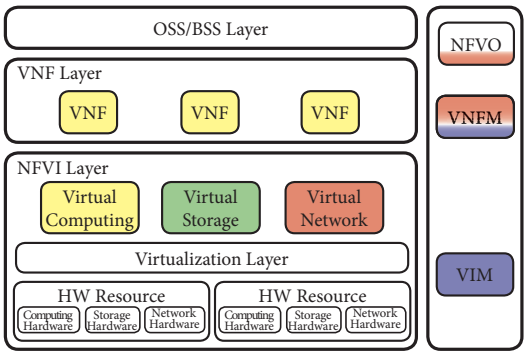


Fig. 1: ETSI MANO NFV Architecture and its relationship with Megalos. In yellow the parts implemented using Docker, in green the Kubernetes volume drivers, in blue Kubernetes itself, and in red the Megalos Controller and CNI.

on a single host: NetKit [7] is a tool capable of emulating arbitrary network topologies using User-Mode Linux (UML) virtual machines. Kathará [8] is a variation of NetKit which uses Docker containers instead of UML virtual machines.

Some network emulators allow to distribute the workload among different hosts independently from the ETSI standard. In [9] a system is proposed, called ClickOS, to provide NFs in lightweight middleboxes (e.g. firewall, load balancer, NAT, ...) running on minimal virtual machines. GLANF [10] allows to run network functions using containers and Software Defined Networking (SDN). DVCL [11] is a project, based on NetKit, that allows the emulation of network topologies distributed across multiple hosts.

A large variety of NFV orchestrators allows the Management And Network Orchestration (MANO) of virtualized network services. Open Source MANO (OSM) [12] (coordinated by ETSI) consists in a set of softwares for the management and orchestration of NFV, fully compliant to the ETSI specifications. However, as for most of the related work described below, the approach of OSM is meant to connect VNFs through direct virtual links, which are not necessarily provided with a full protocol stack and cannot be replaced by an arbitrary switched L2 network. Open Network Automation Platform (ONAP) [13] enables the design, creation, and orchestration of NFV services. It is the main competitor of OSM and has been adopted by big companies (e.g. Nokia, Cisco and Huawei). As OSM, it is fully compliant to the ETSI's architectural specifications. Tacker [14] is an official OpenStack [15] project building VNF Manager (VNFM) and NFV Orchestrator (NFVO) to deploy and operate VNFs on OpenStack. It is strongly based on ETSI MANO Framework. However, Tacker relies on VNF Forwarding Graph network model, while a more general scenario would need to implement plain L2 networks between VNFs without any additional configuration. Tacker, by default, relies on fully-fledged Virtual Machines (VMs), but it can also leverage Kubernetes as a Virtual Infrastructure Manager (VIM) to deploy containerized VNFs. OpenBaton [16] is open-source and implements the NFVO and VNFM modules. The user can select the preferred VIM through various plug-ins. It is fully compliant to the ETSI MANO specifications and allows the support for multi-tenancy. It mainly uses VMs through OpenStack or Amazon AWS, but can also leverage containers with Docker Swarm. Gigaspaces' Cloudify [17] is another NFV orchestrator that provides the NFVO and the VNFM components, with the freedom to choose the preferred VIM. This allows the usage on public clouds or on OpenStack. Unfortunately, as far as we understood, Cloudify does not clarify the level of detail for network configurations, so it is unclear which kind of settings is possible to manage (e.g., MAC Addresses, IP Addresses, ...) or which ISO/OSI levels are mandatory. NFV over Open DC/OS [18] is a concept developed by a research institute in Taiwan with the goal of use Mesosphere's Open DC/OS as an orchestrator for the NFV architecture. It uses Marathon to orchestrate containers executed in a Open DC/OS cluster. NFV over Open DC/OS is only compliant to the VIM and VNFM blocks of the architecture. As far as we know, this architecture still lacks an implementation. As described extensively in [5], there are many other NFV orchestrators currently maintained such as: CloudNFV, OPNFV, CloudBand or HP OpenNFV.

III. A Model of Container-Based Virtual Network and its Scalability Issues

In this section we describe the virtual network model adopted in this paper and its scalability requirements.

A. The Model

What follows is the list of the main features of the virtual network we aim to support with our architecture.

Interconnected Containers: A virtual network consists of several interconnected containers each representing a VNF and each implementing either a complex virtual device (e.g., a router or a switch) or a simpler atomic network function.

Multiple Interfaces: Each container has several L2 interfaces. Each interface is assigned to exactly one virtual LAN. Conversely, a virtual LAN can be connected to several interfaces.

Traffic Segregation between LANs: The traffic between interfaces assigned to a certain virtual LAN is kept separated from the traffic of any other LAN, from the cluster traffic, and from the Internet traffic.

Traffic Integrity: Packets sent by an interface must be received by the destination interface with no modification.

Partial Virtualization: For the sake of generality, we assume that part of the network is virtualized while the remaining part consists of physical devices and physical LANs.

B. Scalability Requirements

We assume to have at disposal a cluster of several *nodes* (physical or virtual computers). Each container is allocated to a specific node and nodes can be anywhere in the Internet.

We face several scalability issues. Some of them are well-known for any distributed architecture and will not be discussed. Some others are specific of our domain. Namely, consider two interfaces i_1 and i_2 of containers c_1 and c_2,

respectively. Suppose that i_1 and i_2 are assigned to the same virtual LAN and that c_1 is allocated to node n_1 and c_2 is allocated to node n_2. The traffic between i_1 and i_2 should not be received by any node different from n_1 and n_2. If this requirement is not satisfied, the traffic would risk to flood the entire infrastructure, impairing any scalability goal. In order to meet this requirement, every node should be able to determine whether a LAN is intra-node or inter-node and which is the node that hosts the container that has assigned a specific interface.

We also consider the following scalability requirements.

High Cohesion: Ideally, containers with interfaces connected to the same virtual LAN should be allocated to the same node, to reduce intra-node communication.

Low Coupling: The number of virtual LANs whose interfaces are allocated in different nodes is minimized.

Load Balancing: The physical allocation of containers should be balanced among the nodes.

IV. The Megalos Framework

In this section we illustrate the Megalos framework.

A. A Reference Architecture

The architecture that we are going to present is conceived to meet the goals illustrated in Sec. III. It is compliant with the ETSI specifications and implements part of the NFV architecture (BSS/OSS, NS Catalog, VNF Catalog, NFVI Resources, and NFV Instances are not implemented).

In order to implement VNFs, our solution leverages on containers. Containers are distributed among a cluster of physical computing nodes called *worker nodes*. These nodes are controlled by a *master node*, in terms of scheduling/unscheduling the containers into them and checking the state of the cluster. The master node also exposes an API to interact with it.

The nodes can be geographically distributed, as long as full mesh L3 connectivity between them is guaranteed.

We use Docker container engine [19] to provide the Virtual Computing module (yellow colour in Fig. 1). To implement the VIM component we used Kubernetes, a container-based orchestrator. Since modifying the Kubernetes logic would be unfeasible, we rely on Kubernetes also for some of the VNFM functionalities, such as VNF failure handling, scaling or updating. This is represented with the blue colour in Fig. 1. The Virtual Storage module, marked in green, is provided by Kubernetes through a multitude of supported drivers [20].

Regarding the Virtual Network module of the NFV Infrastructure (NFVI) layer, none of the available Kubernetes' networking plug-ins matches all the goals of the model described in Sec. III. Therefore, we implemented a custom network plug-in described in Sections IV-C and IV-D (in red in Fig. 1).

The VNFM component of the NFV MANO is realized by an entity called *Megalos Controller*. The main task of the Controller (in red in Fig. 1) is to manage the lifecycle of each VNF, in particular the instantiation and termination of them. Moreover, our Controller has some features of the NFVO that allow it to manage the *network scenario* and to assign each VNF to a physical node according to a scheduler, described in Sec. V. The network scenario is composed by the virtual network topology, the network devices, and their configurations. This information is transmitted in a suitable format by the Controller to Kubernetes which manages the containers accordingly.

B. Containers and their Interfaces

Our VNFs are Docker Containers managed by Kubernetes. Kubernetes doesn't allow to execute raw containers but needs to wrap them into Pods. A Kubernetes *Pod* is a logical host, which can enclose one or more containers that can share volumes and network. To place containers into Pods, we had to solve two main problems, one related to network interfaces and the other one related to the Pod life-cycle.

Regarding interfaces, we have that Kubernetes implements the so called "IP-per-pod" model which means that a Pod has only one network interface, while our VNFs may have several of them. Hence, we decided to have just one container per Pod and exploited a specific technology to provide a Pod with more than one interface. Namely, the single Pod interface provided by Kubernetes is configured by Container Network Interface (CNI) [21]. In order to have multiple interfaces we exploited the Multus CNI [22] plug-in for Kubernetes.

Regarding the Pod life-cycle problem, as suggested in [23], Pods that are directly scheduled (the so called "naked" Pods) are not recommended because they do not reschedule on failure. To ensure the availability of the containers we used *Deployment objects*. These objects represent a set of identical Pods managed by a Deployment controller, which organizes multiple replicas of the same service and automatically reschedules them whenever they fail or whenever the number of running replicas does not match the desired amount.

C. Networking: Data Plane

Several strategies are possible for realising a data plane for the virtual LANs, provided that the cluster network is an L3 network (see Sec. IV-A) and the packets sent by VNFs are not modified (see Sec. III-A).

Pods have routable IP addresses: The IP addresses assigned to Pod interfaces are routable in the physical cluster network, i.e., in the same IP realm. This strategy does not meet our goals since it rules out the existence of interfaces without any IP address and forces each virtual LAN to be an L3 network.

Map each virtual LAN to a specific VLAN: Worker nodes add specific VLAN tags to packets sent from Pod interfaces according to their virtual LAN. However, this strategy would require the cluster network to be a L2 network and a manual configuration on the physical switches.

Custom encapsulation: A custom encapsulation protocol is devised to encapsulate L2 frames into L3 (or above) packets. This strategy is proposed in [11]. We discarded this solution since it relies on non-standard protocols.

VXLAN [24]: A standard to transport Ethernet frames through an IP network based on MAC-in-UDP encapsulation.

Our solution uses VXLAN, because it is widely adopted in real networks, because it is suitable in a possible hybrid network scenario (see Sec. VI), and because of its L4 encapsulation, which guarantees NAT and L4 firewalls traversal.

Each virtual LAN is mapped into a VXLAN Segment and it is identified by a VXLAN Network Identifier (VNI). It is possible to manage about 16M VNIs. The overlay network endpoint is called VXLAN Tunnel End Point (VTEP). VXLAN behavior is the same of a switched L2 LAN: there is a MAC-to-VTEP table that is filled with a flood-and-learn policy. After this phase, Ethernet frames generated by interfaces of a particular VXLAN Segment are only received by the interface matching the same VXLAN Segment and the L2 Destination Address. Thanks to this table, the VTEP itself can also reply to ARP requests (acting as a proxy) without forwarding them to the whole network.

Observe that VXLAN payload is not encrypted. If confidentiality is required, IPSec could be used to encapsulate the entire VXLAN packet (IPSec does not allow to encapsulate a raw L2 packet or to specify a VNI).

D. Networking: Control Plane

The VXLAN flood-and-learn process has two drawbacks. The first is that it relies on IP multicast to populate the MAC-to-VTEP table, which may be not possible as the Internet usually does not guarantee IP multicast packets forwarding. The second problem is that a MAC address is not added to all the MAC-to-VTEP tables until it is used as source address for some packet. In order to overcome these drawbacks we need to replace the flood-and-learn process with an alternative control plane. For example VXFLD [25] implements a custom control plane for VXLAN. In our solution we preferred using Ethernet VPN (EVPN) Border Gateway Protocol (BGP), since it is standardized in [26] and widely used in large data centers [27].

EVPN is a model for exchanging control plane information (in our case MAC-to-VTEP tables) using BGP updates. The BGP daemon is able to read and inject information into the VXLAN data plane so that packets will be correctly encapsulated and routed. When a container c is started on a worker node w, the BGP daemon associated to w announces the MAC address, VTEP, and VNI of all the interfaces of c, so that all the other peers can update their tables.

Kubernetes does not natively support either VXLAN or EVPN BGP. There are already available CNI plug-ins which leverage on VXLAN, but they are designed to manage the inter-node network in the cluster (e.g. [28]–[30]). All these plug-ins require to setup an IP subnet for each Kubernetes node, forcing it to be an L3 network. So, we implemented a custom CNI plug-in, called *Megalos CNI*, which is capable of automatically creating software VTEPs to support VXLAN. Each virtual LAN specified in the desired scenario is mapped to a VXLAN Segment, each one identified by a different VNI.

For supporting EVPN BGP and to keep Kubernetes unaware of the presence of such protocol into the cluster, we put the BGP daemon into a dedicated container, instead of installing it directly into the worker node. To make this solution work, the container needs visibility of the worker network namespace. In this way, it is able to read and inject information directly into the MAC-to-VTEP tables. An instance of this container is deployed on each worker node, while another container configured as a BGP Route Reflector [31] is deployed on the master node. This significantly simplifies the BGP setup because it avoids the full mesh BGP peerings between worker nodes. The solution described above is shown in Fig. 2. Zero configuration is also ensured by dynamic BGP peer discovery. Generally, each BGP peer should statically specify its neighbors in the configuration. In our solution, there is a Kubernetes listener in the Route Reflector that catches every event of addition/removal of a worker node and changes the BGP peerings accordingly.

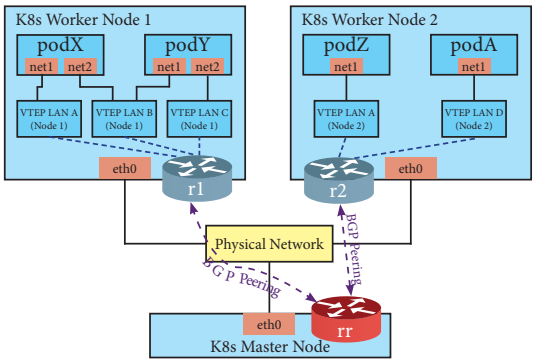


Fig. 2: The Megalos networking Architecture.

E. The Megalos Controller

The network scenario can be described in any format, as long as the Megalos Controller logic is able to convert the chosen format into a Kubernetes understandable language. Once the topology and its configuration are deployed into the cluster, the Controller is no longer required, unless the scenario needs modifications. Hence, it can be detached from the architecture since all the managing is delegated to Kubernetes.

V. A Topology-Aware Scheduler

A. Scheduler Requirements

Megalos scheduler is a modular entity, allowing the user to pursue the high cohesion and low coupling requirements discussed in Sec. III-B by distributing the Pods among the cluster nodes based on a variety of inputs (e.g., traffic matrix, policies). A simple version of the scheduler could take into account the network topology only, assuming that all VNFs exchange a similar amount of traffic. Since Kubernetes does not allow to hot-migrate a Pod after it has been spawned, the scheduler must compute a definitive Pod placement before deploying the scenario and such a placement can not be changed during execution.

B. The Megalos Scheduler and its Interactions

There are two options for interacting with Kubernetes in terms of scheduling: (1) Replacing the Kubernetes scheduler with a custom one or (2) Using the native Kubernetes

scheduler as is. The first option would not yield an effective solution as custom schedulers process a Pod at a time and are not aware of the whole network scenario. On the other side, the native Kubernetes scheduler only allows to balance the usage of available computing resources (CPU and RAM) of the worker nodes. Luckily enough, the Kubernetes scheduler allows to express the preferred worker node where each Pod should be allocated. This preference is enforced if possible. In order to match the requirements described in Sec. V-A, our solution is to execute an external scheduler, called *Megalos Scheduler*, that gives Kubernetes preferences to schedule the Pods in specific worker nodes.

The only entity that is aware of the whole scenario is the Megalos Controller (see Sect. IV-E) which, hence, is in charge of executing the Megalos Scheduler process and to send the resulting preferences to a Kubernetes master. Once the scenario is deployed, the Controller can be detached from the architecture. In fact, in the eventuality of a worker failure, Pods are rescheduled using the default Kubernetes scheduler, which tries to minimize the Pod downtime as much as possible.

C. The Basic Megalos Scheduler Logic

As described in Sec. V-A, in order to compute the optimal placement of Pods, different types of data could be considered and hence different schedulers could be used. The basic Megalos Scheduler computes the placement considering only the network topology. Namely, it builds an undirected graph in which vertices represent VNFs and an edge is added between two VNFs if they share a virtual LAN. After the graph is created, the scheduler performs a Spectral Clustering [32] algorithm producing an approximate solution of the graph partitioning into k-balanced partitions, which is NP-hard.

We refined the basic Megalos Scheduler to also take into account some semantic information. In particular, VNFs have their own configuration files in order to exploit their tasks and parsing these files it is possible to infer how VNFs interact each other. Consider for example a VNF acting as a routing daemon. Parsing its configuration files, it is easily possible to extract information about the ASN of the router or its OSPF area. With this information we can alter the edge weights of the topology graph and execute the Spectral Clustering algorithm on the obtained matrix.

VI. Use Cases

In this section we show use cases for Megalos. In particular, we focus on using Megalos for setting up network scenarios that can be: (i) large, (ii) heterogeneous, and (iii) hybrid.

Large Networks: Megalos can be used to emulate large scale network infrastructures. These scenarios may require a huge quantity of VNFs and virtual LANs. Hence, running them on a single node would be unfeasible. We give two examples.

A large network that we emulated with Megalos is the one of a data center, structured as a Clos [33], [34]. Details on the emulation of this kind of networks are provided in Sec. VII.

Provided that the needed resources are available, Megalos can be used to perform a high-level emulation of the entire Internet interdomain routing. More precisely, suppose to map each Autonomous System (AS) to a single BGP router and use CAIDA ASRank [35] to create a set of BGP customer-provider peer policies, the generated network scenario will count about 68k VNFs [36] and about 300k virtual LANs [35]. In this scenario, ASes need to announce their real IP prefixes, so it is not possible to emulate it without a tool that allows the usage of globally routed IP addresses and that isolates the virtual LANs from the cluster network. A feasible Megalos solution would be to place about 1K Pods per worker, for a total amount of 68 workers. We remark that the emulation in a testbed of the interdomain routing of the Internet has been a challenge for years [37]. Observe that DRMSim [38] tries to solve this problem with a discrete event simulator.

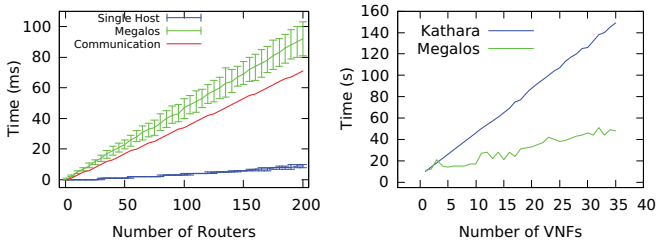
Heterogeneous Networks: The flexibility of Megalos allows to emulate (by creating proper Docker Images) a variety of devices. As an example, we used Megalos for emulating networks with SDN and P4-compliant switches [39].

Hybrid Networks: One limit in the adoption of the NFV paradigm is that migrating in just "one move" from a physical network to a virtual one is unfeasible. Hence, transition methods and tools allowing the co-existence [4] of physical and virtual devices are especially useful. Megalos can be exploited to support this migration. In fact, it is possible to configure a physical router to announce the physical network's MAC Addresses to the virtual portion of the network allowing the communication between them. To achieve this, it is needed to set up EVPN BGP on a physical router with a peering to the route reflector of the Kubernetes master. With this hybrid scenario, it is possible to transpose each physical device into its (one or more) VNF counterpart without any changes in the configuration of the other portion of the network.

VII. Evaluation

To assess the efficiency and effectiveness of Megalos we implemented the Architecture described in Sec. IV and the Scheduler of Sec. V (source code and testbed used in this paper are available at [40]). As we said in Sec. IV-E, any suitable language can be used to describe a network scenario that has to be realized with Megalos. In order to perform the experiments described in this section we used the same language of Kathará. Hence, the Megalos Controller is realized using a modified version of Kathará that uses Kubernetes API.

We performed the following four experiments. *Virtual LAN Scalability:* We performed tests to check how many virtual LANs can be concurrently managed by Megalos. *VNF Scalability:* We designed Megalos in order to manage, even on cheap clusters, large network scenarios where each Virtual Network Function can be as complex as a network device. Hence, we verified that Megalos can emulate a large number of VNFs each equipped with a fully fledged network protocol stack. *Latency:* A weak point of complex architectures (like Megalos) is that they might introduce unnecessary latency. Hence, we performed tests to measure the impact of Megalos on the latency of the virtual network. *Startup:* We measured



(a) (b)

Fig. 3: (a) Results of the latency test. (b) Test results comparing startup times of Megalos and Kathará.

the time required to startup a network scenario first by just using Kathará (single host) and then by using Megalos.

Virtual LAN Scalability: To demonstrate that Megalos can achieve scalability in terms of number of virtual LANs, we measured the amount of virtual LANs that can be deployed in a cluster. The results of the test were very positive. Indeed, we were able to start on a single worker more than 500K different virtual LANs distributed among 1,000 VNFs. However this doesn't come for free: starting a high number of virtual LANs on the same worker implies spending a considerable startup time. Indeed, our experiment took about 90 minutes on a worker equipped with dual Intel Xeon Gold 6126 CPU, 768GB of RAM, 9x SAS SSDs Raid 5, and Debian 9 as OS.

VNF Scalability: The cluster used in the tests described in this and in the following tests is composed of 50 cheap physical nodes, each one equipped with Intel Core i5 4590T CPU, 8GB of RAM, mechanical HDD and Debian 9 as OS. To test the VNF scalability, we used Megalos to implement the virtual network of a small data center consisting of about 2K VNFs. The topology is the one of a Clos Network, analogous to that described in Sec. VI. In the network there are 2,048 servers, each containing an Apache Webserver (using 20 MBytes of RAM), and 128 ToRs, 128 Leafs, 64 Spines, and 2 Exit Nodes, each equipped with a fully-fledged FRRouting [41] daemon (using 10 MBytes of RAM). Hence, we have an average of 48 VNF per worker. The test was successful, we were able to smoothly run the entire topology and we ensured that all hosts were reachable each other. Megalos took about 25 minutes to startup the network scenario in our low-performing cluster.

Latency: To measure the latency overhead induced by Megalos, we realized a network scenario consisting of a chain of $n+2$ VNFs, where: (i) The first and the last VNFs of the chain are a client and a server, respectively; (ii) All the intermediate VNFs are routers; and (iii) Two VNFs that are consecutive in the chain are connected by a virtual LAN. Also, we used a scheduler that never allocates two VNFs that are consecutive in the chain to the same worker. Using `ping` we measured the Round Trip Time (RTT) between the client and the server. We performed the above experiment ranging n from 0 to 200. The measured average RTT times are in Fig. 3a with the green curve (standard deviation is also shown).

In order to obtain the RTT overhead actually introduced by Megalos, we need to subtract from the above described values the intrinsic latency of the containers and the latency due to the physical network. Therefore, we performed the following additional measures. (1) We repeated the above experiment placing the entire network scenario on a single host without using Megalos and implementing each VNF with a single Docker container. The measured average RTTs, reported in Fig. 3a with the blue curve, provide us with an estimation of the intrinsic latency of the containers. (2) We measured the RTT in the cluster between each pair of workers and computed their average. This provided us with an estimation of the average latency of our cluster physical network.

Finally, for each value of n, we summed up the RTT measured in the single host scenario with the average latency of the physical network multiplied by n. The red curve of Fig. 3a shows the result. The difference between the green curve and the red curve is the overhead latency introduced by Megalos. It appears that such overhead is linear with the number of consecutive VNFs traversed by the traffic that are allocated to different workers. This justifies the heuristic approach that we adopted for Megalos Scheduler (see Sec. V-C).

Startup: We focus on measuring Megalos performance with respect to the amount of time needed to deploy network scenarios on a single worker node compared with Kathará (single host, version 0.36.1). We ran this test over multiple network scenarios, each one with a growing number of VNFs. We used a custom scheduler that places every Pod on the same worker node, in order to have comparable results with a single host solution. The result, shown in Fig. 3b, highlights the ability of Kubernetes to schedule Pods in a concurrent way. The results are much better than those of Kathará.

VIII. Conclusions & Future Work

We presented Megalos, an ETSI NFV compliant, scalable, and distributed architecture, that supports the implementation of virtual network scenarios consisting of several networking components (VNFs) where each VNF: (1) can implement a complex networking equipment and (2) can have several L2 interfaces, each connected to a virtual LAN. Some problems, mainly of technical nature, remain open.

Currently, the Megalos Scheduler is executed inside the Megalos Controller. It might be interesting to design a distributed algorithm and to execute it on dedicated Pods of the cluster itself.

Megalos could be expanded to take advantage of High Availability [42] in a multi-master cluster. Currently, this is unfeasible because there is just one Route Reflector of the EVPN BGP that needs to be hosted in the master node. It would be interesting to use Megalos in public clouds like Amazon Web Services, Microsoft Azure, or Google Cloud Platform. However, in the current version of Megalos this is difficult since public clouds do not permit to access the master node where Megalos places its Route Reflector. A possible solution of the above two problems could be that of moving the Route Reflector to a worker node, automatically chosen by a democratic election algorithm.

REFERENCES

[1] NFV White Paper, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. Issue 1," Oct. 2012, unpublished.

[2] ETSI. (2013, October) Network Functions Virtualisation (NFV); Architectural Framework. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf

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

[4] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018.

[5] R. Mijumbi, J. Serrat, J. Gorricho, S. Latre, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, January 2016.

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

[7] M. Pizzonia and M. Rimondini, "Netkit: Easy Emulation of Complex Networks on Inexpensive Hardware," in *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, ser. TridentCom '08. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 7:1–7:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1390576.1390585

[8] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, "Kathará: A Container-Based Framework for Implementing Network Function Virtualization and Software Defined Networks," in *Proc. IFIP/IEEE Network Operations and Management Symposium (NOMS 2018)*, Y.-K. Tu, Ed., 2018.

[9] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 459–473. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616491

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

[11] J. Haag, "DVCL: A Distributed Virtual Computer Lab for Security and Network Education," Ph.D. dissertation, Open University of the Netherlands, 6 2018.

[12] ETSI. Open Source MANO. [Online]. Available: https://osm.etsi.org/

[13] Linux Foundation's OPEN-Orchestrator Project. ONAP. [Online]. Available: https://www.onap.org/

[14] OpenStack. Tacker. [Online]. Available: https://wiki.openstack.org/wiki/Tacker

[15] OpenStack. [Online]. Available: https://www.openstack.org

[16] The Open Baton project. OpenBaton. [Online]. Available: https://openbaton.github.io/

[17] Gigaspaces. Cloudify. [Online]. Available: https://cloudify.co

[18] Y.-H. Chen. Network Function Virtualization over Open DC/OS. [Online]. Available: https://www.communications.org.tw/news/item/download/191_c4756c757cc7c5eeba3a1b11dd241106.html

[19] Docker Inc. Enterprise Container Platform. [Online]. Available: https://www.docker.com/

[20] Kubernetes. Volumes - Types of Volumes. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes

[21] Cloud Native Computing Foundation. Container Network Interface. [Online]. Available: https://github.com/containernetworking/cni

[22] Intel. Multus CNI. [Online]. Available: https://github.com/intel/multus-cni

[23] Kubernetes. Kubernetes Configuration Best Practices. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/overview/#naked-pods-vs-replicasets-deployments-and-jobs

[24] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," Internet Requests for Comments, RFC Editor, RFC 7348, August 2014, http://www.rfc-editor.org/rfc/rfc7348.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7348.txt

[25] Cumulus Networks. VXFLD: VXLAN BUM Flooding Suite. [Online]. Available: https://github.com/CumulusNetworks/vxfld

[26] A. Sajassi, R. Aggarwal, N. Bitar, A. Isaac, J. Uttaro, J. Drake, and W. Henderickx, "BGP MPLS-Based Ethernet VPN," Internet Requests for Comments, RFC Editor, RFC 7432, February 2015, http://www.rfc-editor.org/rfc/rfc7432.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7432.txt

[27] Dinesh G. Dutt, *EVPN in the Data Center*. O'Reilly Media, Inc., July 2018.

[28] CoreOS. Flannel - Network fabric for containers, designed for Kubernetes. [Online]. Available: https://github.com/coreos/flannel/

[29] Tigera. Project Calico - Secure Networking for the Cloud Native Era. [Online]. Available: https://www.projectcalico.org

[30] Weaveworks. Weave Net: Network Containers Across Environments. [Online]. Available: https://www.weave.works/oss/net/

[31] T. Bates, E. Chen, and R. Chandra, "Bgp route reflection: An alternative to full mesh internal bgp (ibgp)," Internet Requests for Comments, RFC Editor, RFC 4456, April 2006, http://www.rfc-editor.org/rfc/rfc4456.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4456.txt

[32] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On Spectral Clustering: Analysis and an Algorithm," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01. Cambridge, MA, USA: MIT Press, 2001, pp. 849–856. [Online]. Available: http://dl.acm.org/citation.cfm?id=2980539.2980649

[33] C. Clos, "A study of non-blocking switching networks," *The Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, March 1953.

[34] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for Routing in Large-Scale Data Centers," Internet Requests for Comments, RFC Editor, RFC 7938, August 2016.

[35] Center for Applied Internet Data Analysis. ASRank. [Online]. Available: http://as-rank.caida.org/

[36] Geoff Huston. The 32-bit AS Number Report. [Online]. Available: http://www.potaroo.net/tools/asn32/

[37] J. Pan, S. Paul, and R. Jain, "A survey of the research on future internet architectures," *IEEE Communications Magazine*, vol. 49, no. 7, pp. 26–36, July 2011.

[38] A. Lancin and D. Papadimitriou, "DRMSim: A Routing-Model Simulator for Large-Scale Networks," *ERCIM News*, vol. 94, pp. 31–32, Jul. 2013. [Online]. Available: https://hal.inria.fr/hal-00924952

[39] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[40] Kathará Framework. [Online]. Available: https://github.com/KatharaFramework

[41] FRRouting. The FRRouting Protocol Suite. [Online]. Available: https://frrouting.org/

[42] Kubernetes. Highly Available Kubernetes Cluster. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/