# Secure and Dependable Multi-Cloud Network Virtualization

Max Alaluna Eric Vial Nuno Neves Fernando M. V. Ramos

LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

{malaluna, evial}@lasige.di.fc.ul.pt, nuno@di.fc.ul.pt, fvramos@ciencias.ulisboa.pt

## Abstract

Existing multi-tenant network virtualization platforms have so far focused on the offer of conventional networking services by a single cloud provider. As such, they face limitations in terms of security and dependability, both in terms of the infrastructure itself and of the services offered to its customers. To address these challenges we present the design and implementation of Sirius, a network virtualization platform for multi-cloud environments. Contrary to existing solutions, Sirius considers not only connectivity and performance, but also security and dependability as first class citizens, leveraging from a substrate infrastructure composed of both public clouds and private data centers.

***CCS Concepts*** • **Networks** → *Programmable networks*

***Keywords*** network hypervisor; cloud computing; virtualisation; Network topology

## 1. Introduction

The advances in computing and storage virtualization that enabled cloud computing have not been met by networking. Traditional forms of network virtualization (VLANs, etc.) do not present the scalability and flexibility that is necessary in current cloud environments. The reason lies fundamentally in the complexity of network management and control. In particular, networking has lacked unifying abstractions to enable network-wide visibility and control. As a result, network provisioning is typically orders of magnitude slower when compared to its computing and storage counterparts [10].

The current state of affairs has recently started to change with the emergence and rapid adoption of Software-Defined Networking (SDN). By decoupling the networking planes and by logically centralizing control, SDN offers operators network-wide visibility and direct control over traffic in the network [11]. These capabilities have led to the development of production-quality network virtualization platforms [10] that allow the creation of virtual networks, each with independent service models, topologies, and addressing schemes, over the same substrate network.

Current multi-tenant network hypervisors target single-provider deployments and traditional services, such as flat L2 or L3, as their goal is to enable tenants to use their existing cloud infrastructures. Such single-cloud paradigm has inherent limitations in terms of scalability, security, and dependability, which may potentially dissuade critical systems to be migrated to the cloud. For instance, a tenant may want to outsource part of its compute and network infrastructure to a public cloud, but may not be willing to trust the same provider to store its confidential business data or to run sensitive services, which should stay in a more trusted environment (e.g., a private datacenter). To avoid cloud outages disrupting its services – a type of incident increasingly common [15] – the tenant may also wish to spread its services across clouds, to avoid Internet-scale single points of failures.

To address this challenge, we propose Sirius, a multi-cloud network virtualization platform. Contrary to previous approaches, Sirius leverages from a substrate infrastructure that entails both public clouds and private datacenters. This brings with it several important benefits. First, it increases resilience. Replicating services across providers avoids single points of failure, making a tenant immune to any datacenter outage. Second, it can improve security, for instance by exploring the interaction between public and private clouds. A tenant that needs to comply with privacy legislation may demand certain data or specific services to be placed in trusted locations. In addition, it can improve performance and efficiency. For example, the placement of virtual machines may consider service affinity to reduce latencies. Specific workloads can also be migrated to clouds which consume less energy [6]. Dynamic pricing plans from multiple cloud providers can also be explored to improve cost-efficiency [17]. The multi-cloud model has been successfully applied in the context of computation [16] and storage [7] recently. To the best of our knowledge, this is the first time the model is applied for network virtualization.

In our platform users can define virtual networks (VN) with arbitrary topologies, while making use of the full address space. Sirius further improves over existing network virtualization solutions by allowing users to specify security and dependability requirements for all virtual resources. In this paper we present the design of Sirius along with its multiple challenges. We also include an initial evaluation of the current prototype.



*XDOMO'17* April 23, 2017 - Belgrade, Serbia



ISBN 978-1-4503-4937-6/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3071064.3071066

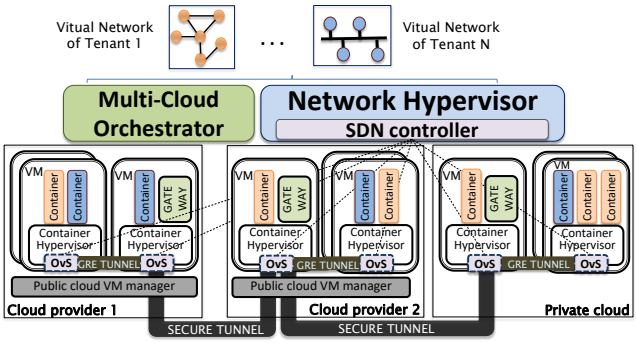


**Figure 1.** Sirius architecture.

## 2. Design of Sirius

Sirius allows an organization to manage resources belonging to multiple clouds, which can then be transparently shared by various users (or tenants). Resources are organized as a single substrate infrastructure, effectively creating the abstraction of a cloud that spreads over several clouds, i.e., a cloud-of-clouds [12]. In this paper, the considered resources are interconnected virtual machines (VM) that are either acquired from public cloud providers or are placed in local facilities (i.e., private clouds). Envisioned extensions include other cloud resources, such as storage services.

Users can define virtual networks composed of a number of containers interconnected according to an arbitrary topology. Sirius deploys these virtual networks in the substrate infrastructure, ensuring isolation of the traffic by setting up separated datapaths (or flows). While specifying the virtual network, it is possible to indicate several requirements for the nodes and links, for example with respect to the needed bandwidth, security properties, and fault tolerance guarantees. These requirements are enforced during embedding by laying out the containers at the appropriate locations, where the substrate infrastructure still has enough resources to satisfy the particular demands. In addition, the datapaths are configured to follow adequate routes through the network.

### 2.1 Architecture

The architecture of Sirius is displayed in Figure 1 and is a significant evolution of the solution proposed in [5]. This paper focuses on the main advances since its previous incarnation: the multi-cloud orchestrator, the network embedding module, and the isolation mechanisms. An important improvement was the addition of the orchestrator module to the design. The cloud orchestrator is responsible for the dynamic creation of the substrate infrastructure by deploying the VMs and containers. It also configures secure tunnels between gateway modules, normally building a fully connected topology among the participating clouds. A gateway acts like an edge router, receiving local packets whose destination is in another cloud and then forwarding them to its peer gateways, allowing data to be sent securely to any container in the infrastructure. Intra-cloud communications between tenant containers use GRE tunnels setup between the local VMs, to ensure isolation.

The network hypervisor runs as an application on top of an SDN controller. It takes all decisions related to the placement of the virtual networks, and setups the network paths by configuring software switches (OvS [14]) that are installed in all VMs (along with OpenFlow hardware switches that may exist in private clouds, not shown in the figure). The hypervisor intercepts the control messages between the substrate infrastructure and the users' virtual networks, and vice-versa, thus enabling full network virtualization.

The hypervisor was developed using a shared controller approach (the solution also adopted in [10]). Alternative solutions, including OVX [3], assume one controller per tenant. Ours is a more lightweight solution, as only one logically-centralized component is needed for all tenants. It is also simpler to implement as it can take advantage of the high-level APIs offered by the SDN controller, instead of having to deal with "raw" Openflow messages when interacting with the switches. Finally, this architecture follows a fate sharing design as the controller and the network hypervisor reside in the same host. This facilitates replication for fault-tolerance.

### 2.2 Overview of Sirius operation

The deployment of a virtual network in the platform involves the execution of a few tasks. The first task is to assemble the substrate infrastructure. The administrator of Sirius within the organization needs to indicate the resources that are available to build the infrastructure. She interacts with a graphical interface[1] offered by the cloud orchestrator that allows the selection of the cloud providers, the type and number of VMs that should be created, and the provision of the necessary access credentials. The network topology is also specified, pinpointing for instance the connections between clouds. For each provider, it is possible to specify a few attributes, such as the associated trust level.

Based on such data, the orchestrator constructs the substrate infrastructure by interacting with the cloud providers and by setting up the VMs. In each VM a few skeleton containers are started with minimal functionality. The gateways are also interconnected with the secure tunnels. The next step is for the hypervisor to be initialized by obtaining, from the orchestrator, information about the infrastructure. Then, it contacts each network switch to obtain data about the existing interfaces, port numbers and connected containers. After populating the hypervisor's internal data structures, Sirius is ready to start serving the users' VN requests.

The second task is run on demand, whenever a user of the organization needs to run an application in the cloud. The user employs a graphical interface of the orchestrator to represent a virtual network with the various containers that implement the application. Containers are then interconnected with the desired (virtual) switches and links. Complete flexibility is given on the choice of the network topology and addressing schemes. Attributes may be associated with the containers and links, specifying particular require-

[1] The same sort of information can also be provided through configuration files, to simplify the use of scripts.

ments with respect to security and dependability. For example, certain links may need to have backup paths to allow for fast fail-over, while certain containers may only be deployed in clouds with the highest trust levels.

The orchestrator receives the VN request and forwards it to the hypervisor to perform the virtual network embedding. The embedding algorithm decides on the location of the containers and network paths considering all constraints, namely the available resources in the substrate infrastructure and the security requirements. The computed mapping is transmitted to the orchestrator so that it can be displayed upon request of the Sirius administrator. Hereafter, the orchestrator and the hypervisor work in parallel to start the VN. The orchestrator downloads and initializes the containers images in the chosen VMs, and configures the IP and MAC addresses based on the tenant's request. The hypervisor enables connectivity by configuring the necessary routes by setting up the flows in the switches, while enforcing isolation between tenants.

### 2.3 The multi-cloud orchestrator

The multi-cloud orchestrator combines three main features. First, it manages interactions with users through a web-based graphical interface. Users with administrator privileges can design the substrate infrastructure topology (Admin GUI), indicating the kind of VMs that should be deployed in each cloud provider. Similarly, normal users can represent virtual networks of containers (User GUI), and later request their deployment. The graphical interface also displays the mappings between the containers and links in the substrate infrastructure and the status of the various components.

Second, it keeps information about the topologies of the substrate and virtual networks and their mappings. This information is kept updated, as virtual networks are created and destroyed, thus offering a complete view of how the infrastructure is currently organized. In addition, it maintains in external storage a representation of the different networks that were specified, allowing their re-utilization when users want to run similar deployments.

Third, it configures and bootstraps VMs in the clouds in cooperation with the network hypervisor and setups the tunnels for the inter-cloud connections. Apart from that, when a virtual network is started, it also initiates the containers in the VMs selected by the hypervisor. A storage of VM and containers is kept locally, in case the users prefer to work and save the images within the organization.

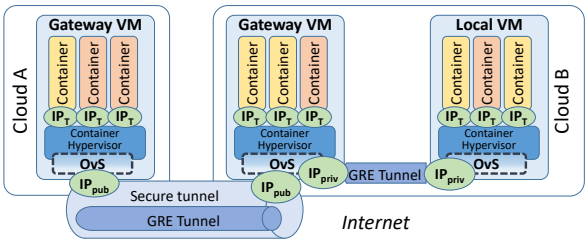


**Figure 2.** Intra- and inter-clouds connections.

Figure 2 shows the main connections that are managed within the infrastructure. Gateways have public IPs that work as endpoints of secure tunnels between the clouds. In our current implementation, OpenVPN with asymmetric key authentication is employed as the standard solution as it presents the advantage of being generic and independent from the provider's gateway service (e.g. VPC service for Amazon EC2). Links between VMs rely on GRE tunnels. We chose this simple approach as intra-cloud communications are expected to be performed within a controlled environment and inter-cloud traffic is protected by the secure tunnel. The containers use the IP addresses defined by the tenants (without restrictions), and isolation is achieved by the network hypervisor properly configuring the switches' flow tables (an aspect to be detailed in Section 2.5).

### 2.4 Hypervisor architecture and components

The design of the hypervisor software follows a modular approach. We present its building blocks in Figure 3.

The **Embedder** addresses the problem of mapping the virtual networks specified by the tenants into the substrate infrastructure [8]. As soon as a virtual network request arrives, the secure Virtual Network Embedding (VNE) module finds an effective and efficient mapping of the virtual nodes and links onto the substrate network, with the objectives of minimizing the cost of the provider and maximizing its revenue. This objective takes into account, firstly, constraints about the available processing capacity of the substrate nodes and of the available bandwidth resources on the links. Moreover, we consider security and dependability constraints based on the requirements specified by the tenants to each virtual resource. These constrains address, for instance, concerns about attacks on virtual machines or on substrate links (e.g., replay/eavesdropping). As such, each particular node may have different security levels, to guarantee for instance that sensitive resources are not co-hosted on the same substrate resource as potentially malicious virtual resources. In addition, we consider the coexistence of resources (nodes/links) in multiple clouds, both public and private, and assume that each individual cloud may have distinct levels of trust from a user standpoint. As future work we plan to include latency as an additional requirement. In the current version of the hypervisor the VNE problem is solved using a MILP formulation. For more details we invite the interested reader to [4].

The **Substrate Network (sNet) Configuration** module is responsible for maintaining information about the substrate topology. It reaches its goals by performing two main functions. First, it retrieves information from the orchestrator about the substrate nodes and links, alongside their security and dependability characteristics. Second, it interacts with each switch to set itself as its master controller, and to collect more detailed information, including switch identifiers, port information (e.g., which ports are connected to which containers), etc. This information is maintained in efficient data structures to speed up data access.

The **Virtual Network (vNet) Configuration** module is responsible for maintaining information about the virtual

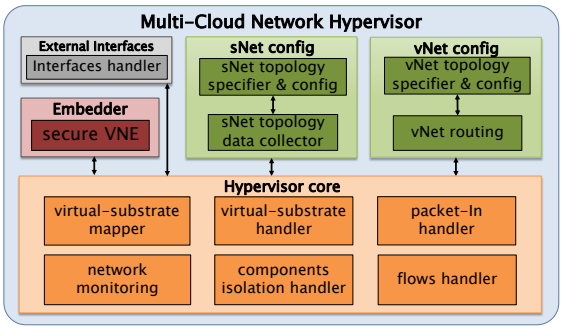


**Figure 3.** Modular architecture of the network hypervisor.

network topologies. This includes both storing tenant requests and the mapping that is result of the embedding phase. As the embedding module outputs only the substrate topology that maps to the virtual network request, this module runs a routing algorithm to define the necessary flow rules to install in the switches (without populating them – that's left for the next module).

The **Hypervisor core** module is configured as a controller module (in our case, Floodlight). Its first component is the virtual-substrate mapper that, after interacting with the substrate topology and virtual topology modules, requests a specific mapping to the embedder. When the output of the VNE returns successfully, the mapping is stored in specific data structures of the core module and this information is shared with other interested modules (namely, the vNet configuration module).

Network monitoring is responsible to detect changes in the substrate topology when a reconfiguration occurs (e.g., due to failures in the substrate network). This module then sends requests to the virtual-substrate handler to update its data structures accordingly. As ongoing work we are implementing mechanisms to respond to network changes.

Isolation is handled by several sub-modules, including the isolation handler, the packet-in handler and the flows handler. These components' goal is to guarantee that each tenant perceives itself as the only user of the infrastructure. We currently use four main techniques for this purpose. First, as we have control over the entire infrastructure, from the core to the edge, we uniquely identify each tenants' host by its precise location. Second, based on this unique identification and on the tenant ID we perform address translation at the edge from the tenant's MAC to an ephemeral MAC address (eMAC) and install the required flows based on the eMAC. The flows for communication between all virtual nodes are initially installed pro-actively by the flow handler module in such a way as to guarantee isolation between tenant's traffic. For efficiency reasons, flows are installed with predefined timeouts. When a timeout expires (which means a particular pair of nodes has not communicated during that period) the flow is removed from the switches to save flow table resources. If communication ensues between those nodes afterwards, the first packet of the flow generates a packet-in that is sent to the hypervisor, triggering the packet-in handler to install the required flows in switches. Third, we perform traffic isolation during the initial steps of communication, namely, by treating ARP requests and replies. Finally, flow table isolation is guaranteed by each virtual switch having its own virtual flow table, with a predefined size limit. We detail these techniques further in the next section.

### 2.5 Virtualization runtime: achieving isolation

The main requirement of our multi-tenant platform is to provide full network virtualization. To achieve this goal it is necessary to virtualize the topology, addressing, and service models, and guarantee isolation between tenants' networks. Topology virtualization is achieved in our system by means of the embedding procedure already described. In this section we focus on the other three aspects.

Sirius allows tenants to configure their VMs with any L2 and L3 addresses. Tenants thus have complete autonomy to manage their address space. They can also retain their preferred L2 and L3 service models (for instance, they can use VLAN services). Giving tenants with these options precludes the use of labeling techniques for virtualization, such as using VLAN tags to identify tenants (as this would break the L2 service model) or inserting tenant-based tags in the L2 or L3 address (as this would restrict the addressing choices).

To achieve these two goals and guarantee isolation we create a unique identifier for each tenant's hosts based on their location. We then perform edge-based translation of the host MAC address to an ephemeral MAC address that includes this ID. Finally, we setup tunnels between every OvS (i.e., between every VM of the substrate infrastructure).

An alternative solution that would also fulfill our requirements would be to setup tunnels between all tenant's hosts (in our solution this would mean setting up tunnels between containers). This would avoid the need to maintain host location information and of edge-based translation. The problem of this option is scalability. The number of tunnels would grow with the number of containers (i.e., with the number of tenant's hosts), whereas our solution scales much better, as it grows with the number of provider VMs (in a production setting, each VM is expected to run hundreds or even thousands of containers).

**Uniquely identifying hosts.** The tenant's hosts of our solution are containers. We opted for this operating system virtualization technology as it provides functionality similar to a VM but with a lighter footprint [9]. Each container (i.e., each tenant's host) has its own namespace (IP and MAC addresses, name, etc.) and its own resources (processing capacity, memory), and as such can be seen as a lightweight VM.

To uniquely identify a tenants' host, at this stage we use its network location (we do not yet consider host migration – that's part of future work). Each container is connected to a specific software switch (identified by a *DatapathID*), being attached to a unique port. As such, we use as *hostID* the tuple $\langle switch\ port, DatapathId \rangle$. Figure 4 shows an example.

**Edge address translation.** Packets generated in a virtual network cannot be transmitted unmodified in the substrate

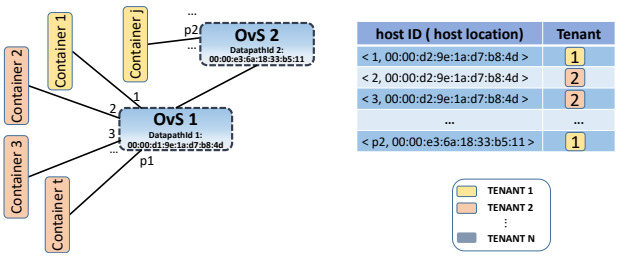


**Figure 4.** (Switch port, DatapathId) = host ID

network. As different tenants can use the same addresses, collisions could occur. For this reason, we perform edge-based address translation to ensure isolation. We assign an *ephemeral* MAC address – eMAC – at the edge, to replace the host's MAC address. The translation occurs at the edge switch. Every time traffic originates from a container its host MAC is converted to the eMAC. Before the traffic arrives at the receiving container the reverse operation occurs at the edge switch. The eMAC is composed of a tenant ID and a shortened version of the *hostID*, unique per tenant.

This mechanism guarantees isolation in the data plane. The control plane guarantees are provided by the hypervisor, as it has network-wide control and visibility. For this purpose the hypervisor populates the flow tables with two types of rules: translation rules in the edge switches, as just explained; and forwarding rules that enable communication between all hosts from a single tenant.

**ARP handling.** Hosts use the ARP protocol to map an IP address to an Ethernet address. As we want unmodified hosts to run in our platform, Sirius emulates the behavior of this protocol. When an ARP message arrives at a switch, it is forwarded directly to the destination host. Flooding is never needed as the switches are configured by the hypervisor. Even in those cases where the packet arriving at the switch does not match any flow rule – because it has expired – a packet-in is sent to the hypervisor, which populates the required tables with the necessary flow rules for the packet to be forwarded to the destination.

**Flow table virtualization.** As forwarding tables have limited capacity, in terms of TCAM entries (hardware switches) or memory (software switches), in Sirius each tenant has a finite quota of forwarding rules in each switch. This is important because the failure to isolate forwarding entries between users might allow one tenant to overflow the number of forwarding rules in a switch and prevent others from inserting their flows. Our hypervisor maintains a counter of the number of flow entries used per tenant switch, and ensures that a preset limit is not exceeded.

The hypervisor controls the maximum number of flows allowed per tenant, in both physical and virtual switches. This control is performed using the OpenFlow field *cookie* (an opaque data value that allows flows to be identified [13]). When the hypervisor inserts a new flow in a switch (which only occurs if the limit was not exceeded), the cookie field is properly set to identify its tenant owner, and the counter for the number of flows in this switch that belong to this particular tenant is incremented. When a flow is removed the hypervisor is informed, extracts from the cookie the tenant owner of the flow just removed, and decrements the corresponding counter.

## 3. Implementation

The Sirius network hypervisor is implemented in Java as a Floodlight controller module. The orchestrator runs in an Apache Tomcat server. The client GUI is written in $Javascript/\ JQuery$ and implements the *vis.js* [2], an open-source library for network visualization. Communication between the HTTP client and server is performed using *Servlet* technology. We have deployed the Linux VMs and the Docker containers in two cloud infrastructures: Amazon EC2 as public cloud, and a private platform based on a set of VMs running in VirtualBox. In order to interconnect clouds we use openvpn tunnels installed in each gateway VM (as illustrated in Figure 1). To interconnect the OvS of each VM we use GRE tunnels. We manage the public cloud using *jclouds*. *Apache jclouds* [1] is a library that offers a simple interface to manage VMs running in public clouds. More importantly, it supports a large number of cloud providers and its generic API assures higher portability, which will facilitate future integration of other public clouds into the substrate infrastructure.

## 4. Evaluation

The evaluation of Sirius addresses two questions: First, how long does it take to bootstrap a set of containers, in both public and private clouds? This question is important to understand the cost to integrate the computation elements in the virtual infrastructure. Second, how long does it take to setup a virtual network, and what is the influence of the different components (embedding, container configuration, setting up flows)?

We consider one public (Amazon EC2 in Germany – Frankfurt) and one private (the private cloud is our datacenter in Portugal – Lisbon) cloud in the evaluation. In Amazon EC2 we use t2.medium as gateway VMs and t2.micro as normal VMs. The private cloud is based on a rack of Dell R420, with 2 Intel Xeon E5520 quad-core, 2.2 GHz, and 32 GB RAM. VirtualBox managed the VMs, which were configured with 1 CPU and 2GB RAM. The VMs run Ubuntu with Docker (1.13.1) and OvS (2.5.0). The containers were also based in Ubuntu.

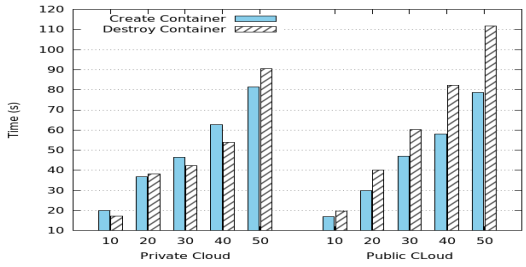


**Figure 5.** Time to create and destroy containers.

For the first experiments we assume that 10 VMs had already been created, and measure the time it takes to set up

and destroy a group of basic containers. These operations occur to scale up or scale down the substrate infrastructure. In the experiments, the location of the containers was selected randomly. Container bootstrapping involves two fundamental steps: the build operation of Docker with an available image in the VM, and linking the containers' network interface to OvS. Container destruction corresponds to the reverse operations. Figure 5 shows the results from this experience. We can observe that the time to create and destroy containers is relatively similar in private and public clouds. Moreover, it is possible to observe that the time for both operations grows linearly in either cloud. Roughly, it takes around one second to setup an additional container.

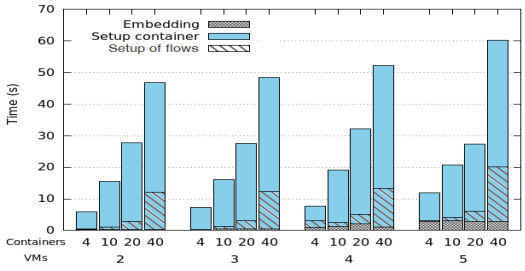


**Figure 6.** Time to setup a virtual network.

Figure 6 presents the time to setup a virtual network. This graph shows the main overheads associated with the deployment of a virtual network in VMs distributed in public and private clouds. Recall that the first step of the process is virtual network embedding. Then, it is necessary to initiate the container image and perform network configuration (setting IP and MAC addresses, etc.). Finally, it is necessary to populate the flow table of switches with the required rules.

Overall, the observed performance is acceptable, taking approximately 1 minute to startup a virtual network with around 40 containers. The time to setup containers is the main component. As concluded before, this time increases linearly. Setting up flows is the second main contributor to the total time. The table insertion time increases with the number of containers, as more flow table entries have to be configured, but this is still well below container setup. However, its growth rate (considering an increasing number of containers) seems higher, so we anticipate the need to optimize this process for larger scale networks. The time for embedding is the smallest contributor to overall time, which means that for these relatively small scale networks using a MILP formulation to address this problem is acceptable. Anyway, although this may not be immediately visible in the graph, the time for embedding grows very fast with the number of switches (from about 300 milliseconds with 2 switches to 3 seconds with 5 switches). This increase is a consequence of using a MILP solution for optimal embedding. For this reason, an important future direction of work is to develop efficient heuristics to the secure VNE problem.

## 5. Conclusions

In this paper we presented Sirius, a multi-cloud network virtualization platform. The design of Sirius extends the guarantees of connectivity and performance of virtual networks offered by existing solutions with security and dependability. This is achieved by leveraging from multiple cloud infrastructures (both public and private) and by considering resiliency requirements from users during the virtual network embedding process. The evaluation of the current prototype demonstrates its feasibility and sheds light on some of the necessary future work.

## Acknowledgments

This work was partially supported by the EC through project SUPERCLOUD (H2020-643964), and by national funds of Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/00408/2013 (LaSIGE).

## References

[1] jclouds. `https://jclouds.apache.org/`, 2017. Accessed: 2017-02-20.

[2] VIS.JS. `http://visjs.org/`, 2017. Accessed: 2017-02-20.

[3] A. Al-Shabibi et al. OpenVirteX: Make your virtual SDNs programmable. In *HotSDN*, 2014.

[4] M. Alaluna, L. Ferrolho, J. Rui Figueira, N. Neves, and F. M. V. Ramos. Secure Virtual Network Embedding in a Multi-Cloud Environment. *ArXiv*, 2017.

[5] M. Alaluna, F. Ramos, and N. Neves. (Literally) Above the Clouds: Virtualizing the Network Over Multiple Clouds. In *Proc. IEEE NetSoft*, 2016.

[6] J. Baliga, R.Ayre, K. Hinton, and R. Tucke. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 2011.

[7] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. Scfs: A shared cloud-backed file system. In *Proc. USENIX ATC*, 2014.

[8] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys Tutorials*, 2013.

[9] J. Higgins, V. Holmes, and C. Venters. Orchestrating docker containers in the HPC environment. In *Proc. ISC High Performance*, 2015.

[10] T. Koponen et al. Network virtualization in multi-tenant datacenters. In *Proc. USENIX NSDI*, 2014.

[11] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 2015.

[12] M. Lacoste, M. Miettinen, N. Neves, F. Ramos, M. Vukolic, F. Charmet, R. Yaich, K. Oborzynski, G. Vernekar, and P. Sousa. User-Centric Security and Dependability in the Clouds-of-Clouds. *IEEE Cloud Computing*, 3(5), 9 2016.

[13] ONF. OpenFlow Switch Specification, 2015.

[14] B. Pfaff et al. The design and implementation of open vswitch. In *Proc. USENIX NSDI*, 2015.

[15] USA TODAY. Massive amazon cloud service outage disrupts sites, February 2017.

[16] D. Williams, H. Jamjoom, and H. Weatherspoon. The xen-blanket: Virtualize once, run everywhere. In *Proc. ACM EUROSYS*, 2012.

[17] L. Zheng, C. Joe-Wong, C. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *Proc. ACM SIGCOMM*, 2015.