D4.3
Proof-of-concept Prototype of the Multi-Cloud Network Virtualization Infrastructure

Project number:	643964
Project acronym:	**SUPERCLOUD**
Project title:	User-centric management of security and dependability in clouds of clouds
Project Start Date:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Demonstrator
Reference Number:	ICT-643964-D4.3/ 1.0
Work Package:	WP 4
Due Date:	May 2017 - M28
Actual Submission Date:	23^{rd} June, 2017
Responsible Organisation:	FFCUL
Editor:	Fernando M. V. Ramos, Nuno Neves
Dissemination Level:	PU
Revision:	1.0
Abstract:	In this deliverable, we describe the software components that form the multi-cloud network virtualisation infrastructure. We give an overview of the structure of the network framework, we detail the APIs of its main components, and we give information on how to access and use the software developed.
Keywords:	network virtualisation, multi-cloud, software-defined networking, self-management, security

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643964.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

Editor

Fernando M. V. Ramos, Nuno Neves (FFCUL)

Contributors (ordered according to beneficiary numbers)

Ruan He, Pascal Legouge, Marc Lacoste,
Nizar Kheir, Redouane Chekaoui, Medhi Boutaka (ORANGE)
Eric Vial, Max Alaluna (FFCUL)
Khalifa Toumi, Rishikesh Sahay, Gregory Blanc (IMT)

Disclaimer

Executive Summary

In this deliverable, we describe the software components of the multi-cloud network virtualisation infrastructure. We start with an overview of the network virtualisation architecture. We then detail the APIs of its main components, including the core elements (network hypervisor and orchestrator) and the self-management security services running on top (security monitoring, appliance chaining, and network security). For each component, we include details on how to access and use the software developed.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The objective of SUPERCLOUD Work Package 4 (WP4) is to develop a platform that creates a virtual network abstraction to the SUPERCLOUD user, spanning multiple heterogeneous Cloud Service Providers (CSPs). Previously, we have focused on the design of the SUPERCLOUD network virtualisation architecture and on prototyping its various components. The preliminary architecture was presented in Deliverable D4.1, and in Deliverable D4.2, we presented its evolution alongside a detailed description of its main components and the techniques used to improve the dependability, scalability, and security of the platform.

1.1 Objective of the document

This deliverable consists of the proof-of-concept prototype of the multi-cloud network virtualisation infrastructure. This is a demonstrator ("software") deliverable that is accompanied by the present document. Our purpose here is to describe the structure of the network framework and the APIs of its main components, namely:

- The multi-cloud orchestrator that manages interactions with users through a web-based graphical interface, keeps information about the topologies of the substrate and virtual networks and their mappings, and configures and bootstraps VMs in the clouds in cooperation with the network hypervisor.
- The network hypervisor that performs the embedding (i.e., defines the mapping) of the virtual network to the substrate infrastructure, sets up the required forwarding state, and guarantees isolation between the users' virtual networks.
- The security monitoring component that detects and responds to security incidents.
- The service chaining component that allows end-users to compose their own security service chains in a multi-cloud environment.
- The network security module that manages and deploys network security policies automatically, by interacting with the security monitoring tool.

1.2 Outline

The rest of this document is organized as follows. Chapter 2 starts with an overview of the network virtualisation architecture. Then, each of the next two chapters gives a short overview of each software component and their internal and external APIs. In particular, Chapter 3 focuses on the API of the core components: the orchestrator and the network hypervisor. Then, Chapter 4 describes the details of the self-management security services, including security monitoring, service chaining, and network security. These chapters also include information about how to obtain and use each component. We finish this report with a discussion on integration aspects and conclusions in Chapter 5.

Chapter 2 Network virtualisation architecture overview

Current multi-tenant network hypervisors target single-provider deployments and traditional services, such as flat L2 or L3 networks, 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 [16] – the tenant may also wish to spread its services across clouds, to avoid Internet-scale single points of failures.
To address this challenge, in SUPERCLOUD, we are developing Sirius, a multi-cloud network virtualisation platform. Contrary to previous approaches, Sirius leverages a substrate infrastructure that entails both public clouds and private datacenters. This brings several important benefits. First, it increases resilience. Replicating services across providers avoids single points of failure and therefore makes a tenant immune to any datacenter outage. Secondly, 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 [7]. Dynamic pricing plans from multiple cloud providers can also be explored to improve cost-efficiency [18]. The multi-cloud model has been successfully applied in the context of computation [17] and storage [8] recently. To the best of our knowledge, this is the first time the model is applied for network virtualisation.
In our platform, users can define virtual networks with arbitrary topologies, while making use of the full address space. Sirius further improves over existing network virtualisation solutions by allowing users to specify security and dependability requirements for all virtual resources. In this chapter, we present the Sirius architecture.

2.1 General design and operation

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 chapter, 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 on 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

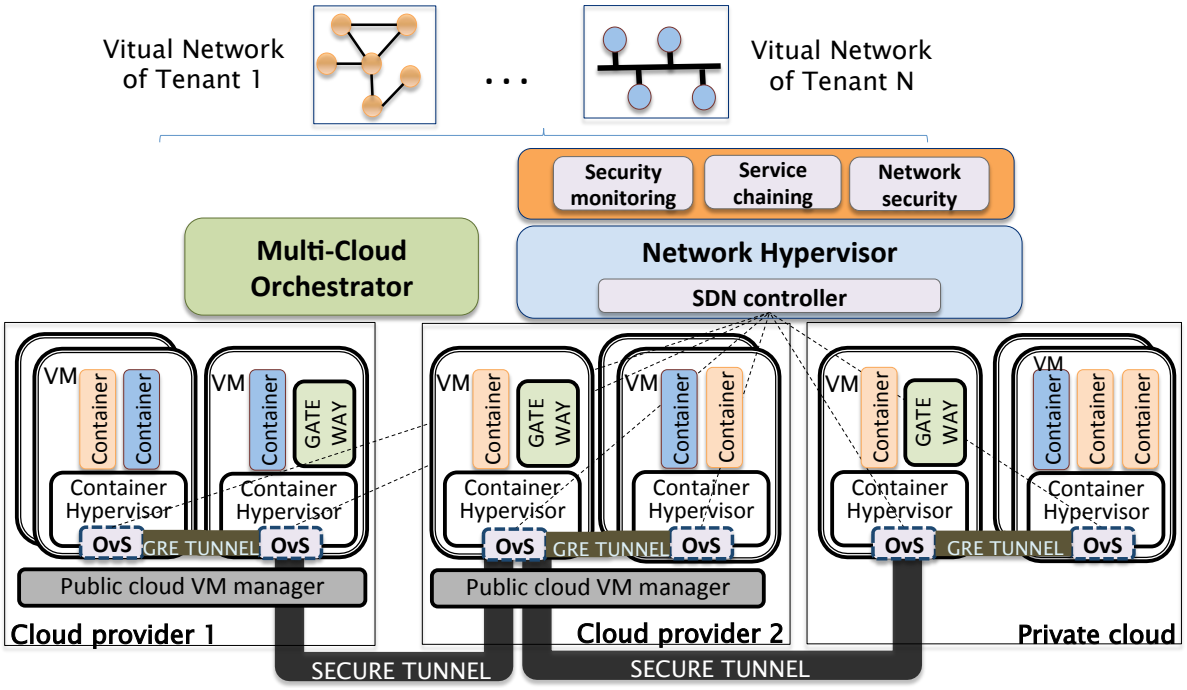

Figure 2.1: Sirius architecture.

datapaths are configured to follow adequate routes through the network.
In the rest of this section, we present the design of Sirius. First, we describe the architecture of the platform and give a step-by-step overview of its operation while creating a virtual network. Next, we elaborate on the two main components of Sirius, the network hypervisor and the cloud orchestrator.[1]

2.1.1 Architecture

The architecture of Sirius is displayed in Figure 2.1. This chapter focuses on the main advances since its previous incarnation (Deliverable D4.2 and [6]): 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 (Section 2.2.1). 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 (Generic Routing Encapsulation) tunnels setup between the local VMs, to ensure isolation.
The network hypervisor (Section 2.2.2) runs as an application on top of a Software-Defined Networking (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 virtualisation.
The hypervisor was developed using a shared controller approach (the solution also adopted in [11]). Alternative solutions, including OVX [4], 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

[1]Like in the Sirius star system, our platform also has two companion components.

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.
The self-management security services run on top of the network hypervisor. They include a security monitor to detect security incidents, a network security service that responds to these incidents, and a service chaining component that allows users to compose their own security service chains. In Section 2.3, we give an overview of each of these services.

2.1.2 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[2] 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' virtual network (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 requirements 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.

2.2 Network virtualisation core components

2.2.1 Multi-cloud orchestrator

The main modules of the multi-cloud orchestrator are detailed in Figure 2.2. 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

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

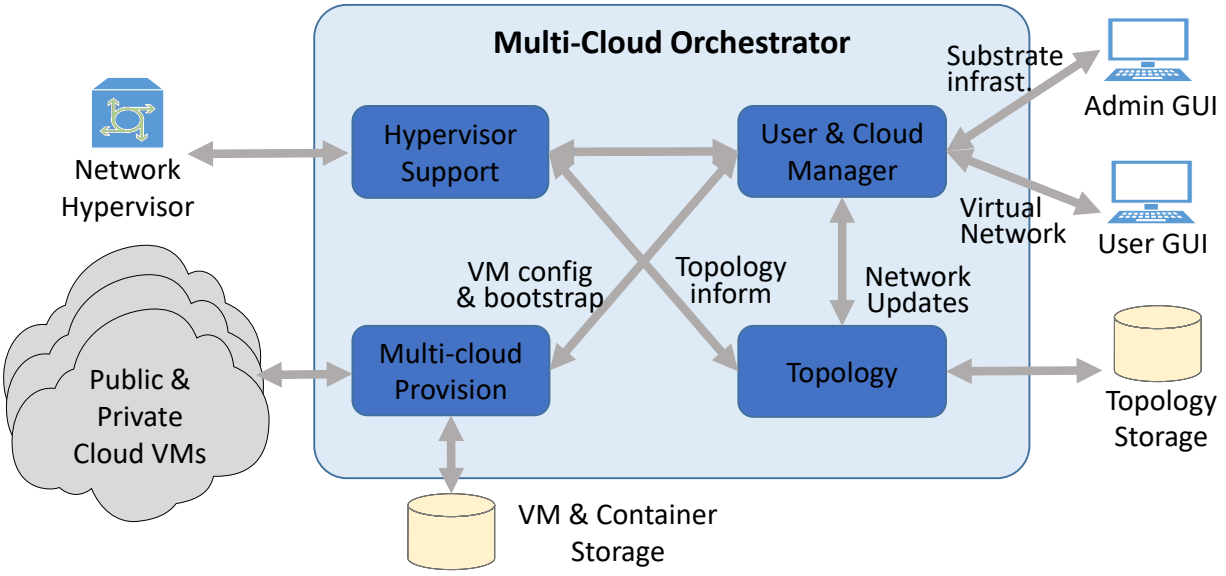

Figure 2.2: Orchestrator's main modules.

topology (Admin GUI), indicating the kind of VMs that should be deployed in each cloud provider. Similarly, normal users can represent virtual networks of virtual hosts (e.g., containers), and later request their deployment (User GUI). 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.

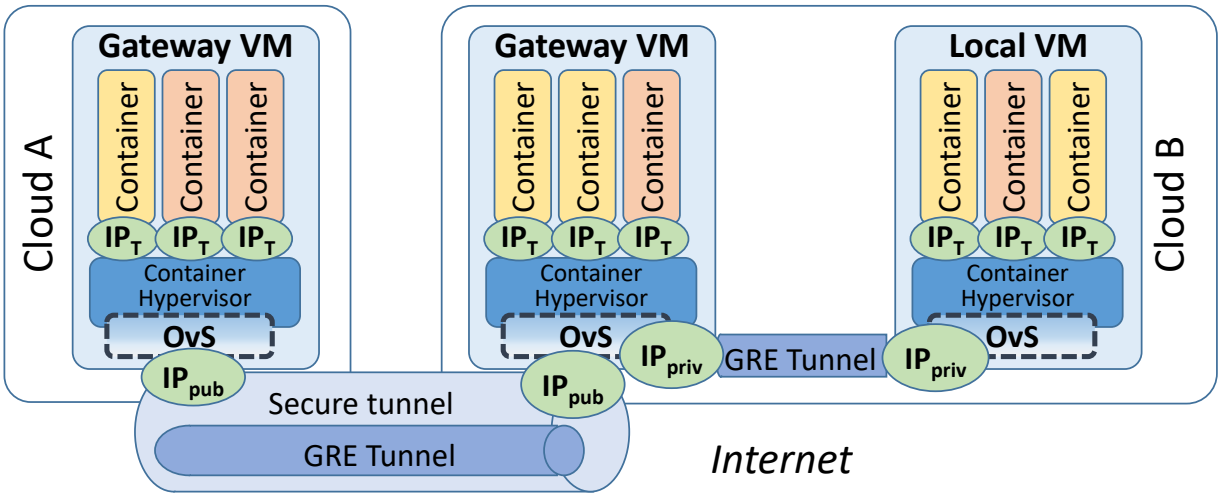

Figure 2.3: Intra- and inter-clouds connections.

Figure 2.3 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 default 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 2.2.2).

2.2.2 Hypervisor

The design of the hypervisor software follows a modular approach. We present its building blocks in Figure 2.4.

The **Embedder** addresses the problem of mapping the virtual networks specified by the tenants into the substrate infrastructure [9]. 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 constraints 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 Mixed Integer Linear Programming (MILP) formulation. For more details we invite the interested reader to consult [5].

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 network topologies. This includes both storing tenant requests and the mapping that results from 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, which is 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 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).

The network monitoring component 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

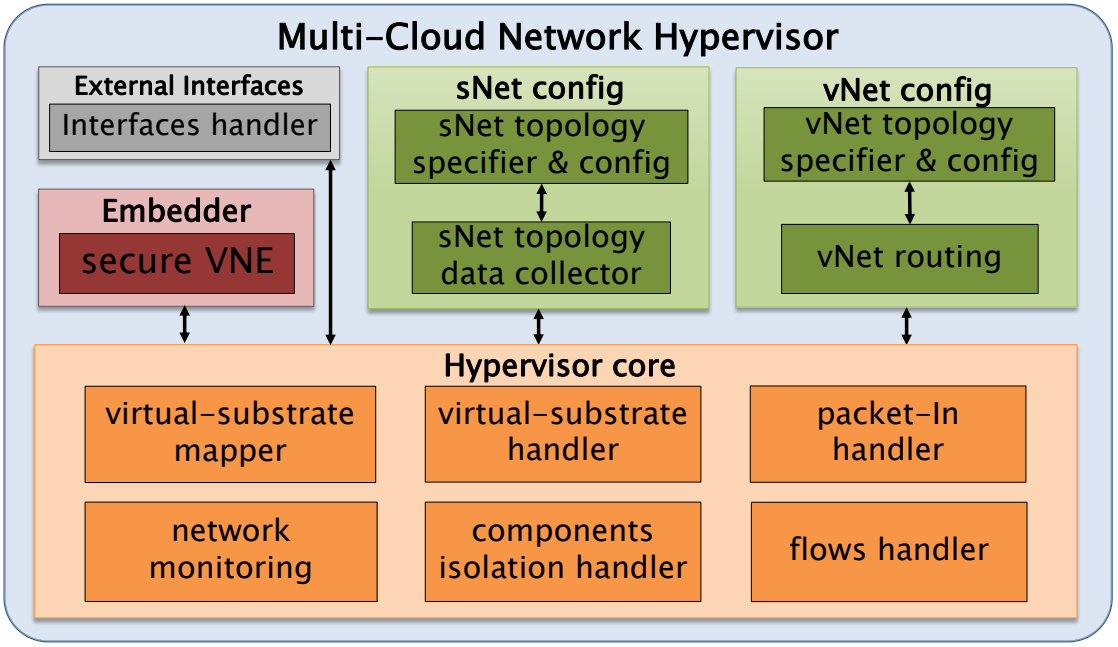

Figure 2.4: Modular architecture of the network hypervisor.

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 network 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 network edge from the tenant's MAC to an ephemeral MAC address (eMAC) and install the required flows based on the eMAC. For communication between all virtual nodes, a set of flows is initially installed pro-actively by the flow handler module in such a way as to guarantee isolation between tenants' 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 tables, with predefined size limits.

We detail these techniques further in the next section.

2.2.3 Virtualisation runtime: achieving isolation

The main requirement of our multi-tenant platform is to provide full network virtualisation. To achieve this goal it is necessary to virtualize the topology, addressing, and service models, and guarantee isolation between tenants' networks. Topology virtualisation 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). Offering tenants these options precludes the use of labeling techniques for virtualisation, 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. As explained, the tenant's hosts of our solution are containers. We opted for this operating system virtualisation technology as it provides functionality similar to a VM but with a lighter footprint [10]. 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.

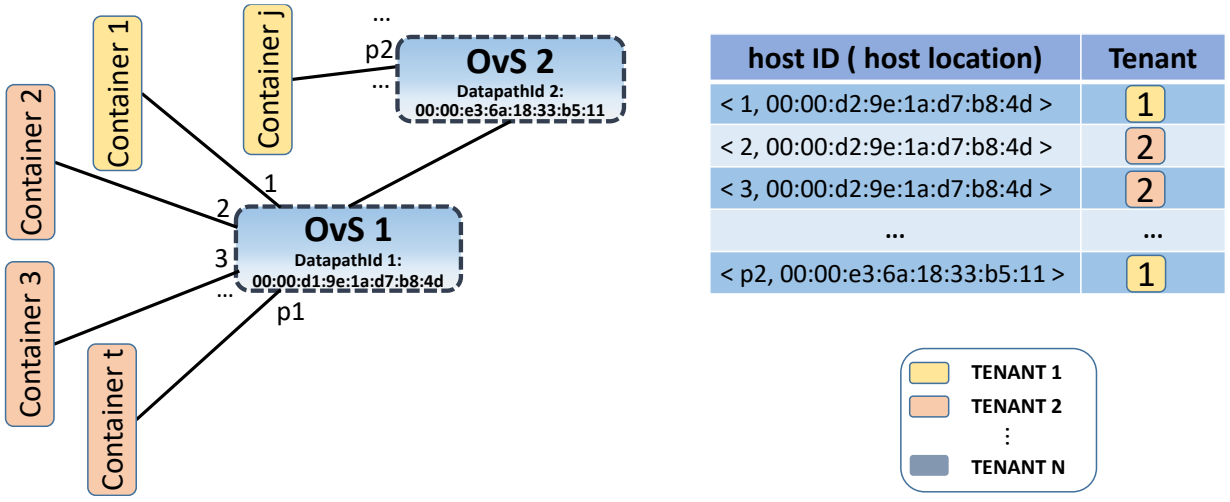

Figure 2.5: <Switch port, DatapathId> = host ID

To uniquely identify a tenant's 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 2.5 shows an example.
Edge address translation. Packets generated in a virtual network cannot be transmitted unmodified in the substrate 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 network 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 virtualisation. As forwarding tables have limited capacity, in terms of TCAM (Ternary Content Addressable Memory) 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.

2.2.4 Additional implementation details

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 uses *vis.js* [3], 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 2.1). To interconnect the OVS of each VM, we use GRE tunnels.
We manage the public cloud using *Apache jclouds* [2], 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.

2.3 Self-management network security

The self-management security module is composed of three components: security monitoring, network security, and service chaining. The security monitoring component allows the detection of security incidents in a tenant network hosted over a multi-cloud. For this purpose, it collects and processes information from these infrastructures to have a complete view of the state of the network, enabling automatic response to security incidents. These responses can be materialised by means of the network security component, that manages and deploys network security policies automatically by interacting with the security monitoring module. Finally, the service chaining component can be used as a response, and also to allow users to customize the composition of security services. Chapter 4 further details each of these services.

Chapter 3 Network virtualisation core interfaces

In this chapter, we will describe the APIs of the core network virtualisation components, namely the orchestrator and the hypervisor. We consider two types of communication interfaces – internal and external (Figure 3.1). The internal interfaces refer to the communication between the core components whereas the external interface allows an authorized HTTP application such as the self-management security modules (Chapter 4) to send requests (over the Internet or locally) to the hypervisor.

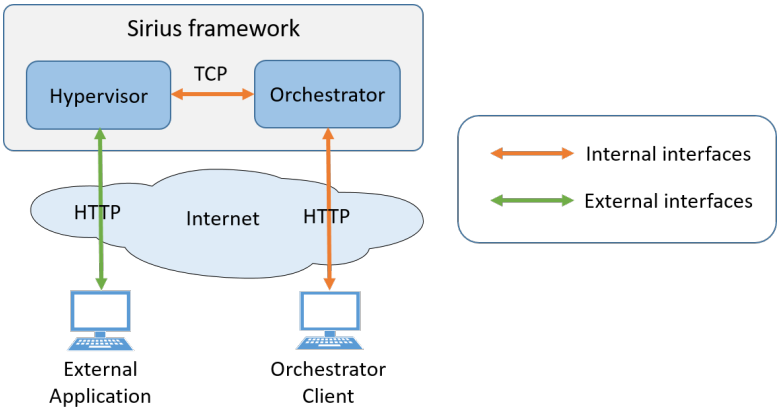

Figure 3.1: Sirius flow communication

3.1 Internal interfaces

There are two main internal interfaces, one between the hypervisor and the orchestrator Java VMs; the second between the orchestrator HTTP client (running the graphical user interface) and the orchestrator server. The Java VMs can run on the same machine or on different platforms, and we assume this communication to be made in a controlled environment. As such, communications are not secured.

3.1.1 Hypervisor-orchestrator communication

The two main functions of this interface are as follows:

- **Substrate initialization and update**: The orchestrator is in charge of the local topology database. At bootstrap, the hypervisor requests the substrate topology. Similarly, any modification of the substrate topology will be sent to the hypervisor as a topology update.
- **Tenant topology and embedding result**: The orchestrator allows the creation or modification of a tenant's topology. This request is sent to the hypervisor, which returns the result of the embedding process.

The communication between the hypervisor and the orchestrator uses a TCP channel, with the orchestrator configured as TCP server. The data is XML-encoded, and a simple protocol was developed to support packet reception in both synchronous and asynchronous modes.

Although both the hypervisor and the orchestrator were written in Java, we do not use the native Java serialization mechanisms: data is XML-encoded and is sent as an array of bytes (UTF8 charset).

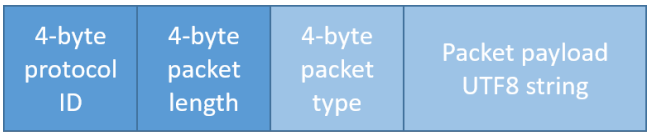

Figure 3.2: Packet's header and payload

As shown in Figure 3.2, the header of our protocol is composed of a protocol ID field, packet length (length of the payload + 4 bytes) and the packet type. The type field allows having several types of requests and replies, all of which are listed in Table 3.1.

Value	Source	Payload	Description
1	Hypervisor	None	Request to get the substrate topology
2	Orchestrator	XML topology	Reply containing the substrate topology
3	Orchestrator	Error string message	Request has failed
4	Orchestrator	XML topology	Request to send the tenant topology
5	Hypervisor	XML topology	Reply containing the embedding result
6	Hypervisor	Error string message	Request has failed

Table 3.1: Request/reply types

Reply types 3 and 6 are failure codes. As with the XML topology, the error message is coded as an array of bytes (UTF8 charset).

3.1.2 Orchestrator client-server communication

Communication between the orchestrator's server and client relies on the HTTP protocol. The server runs a Web servlet to handle the HTTP requests sent by the browser (see Figure 3.3).

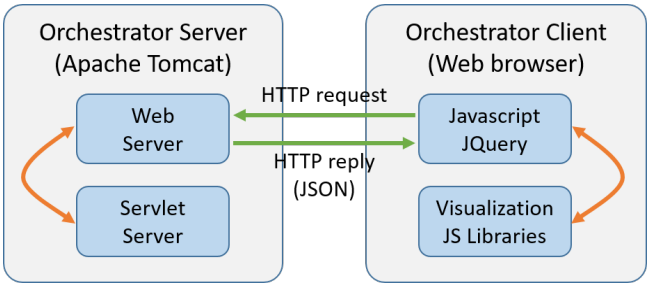

Figure 3.3: Orchestrator client-server communication

The information sent by the orchestrator server is JSON-encoded. Requests are mainly GET HTTP methods. HTTP response status codes are standard. The common error/success codes can be returned to inform the client that a request has failed. For instance, HTTP code *200* means the request has been processed properly; client error code *400* informs of a bad request; and server error code *500* represents a server internal error.

3.2 External interfaces

The external Sirius REST API allows a remote client application to connect to the hypervisor synchronously. Client requests fall into two categories:

- Requests to get topology information (about substrate and tenant networks).
- Requests to perform redirection of all traffic from one virtual host to another.

The API contains only *GET HTTP* methods. The response status codes are the same as for the orchestrator client-server communication. A request failure may be due to an invalid syntax (malformed arguments) or because the server could not complete the requested action (e.g., action not authorized). In the following we detail each category of requests.

3.2.1 Topology request

The topology request syntax, arguments, and responses are given in Table 3.2.

Request	Arguments	Response
GET URL/topology?	id=ID	JSON encoded object

Table 3.2: Topology request overview

The request arguments are as follows. The identifier of the source topology is $id = 0$ for the substrate topology and $id > 0$ for a given tenant topology. Next, we present the attributes of the main components of the substrate and tenant topologies, followed by the reply content returned by the server.

3.2.1.1 Substrate attributes

We consider the following items as part of the substrate topology definition: *Clouds*, *VMs*, *Containers*, *OVS Switches* and *Links*. Each item has a set of configured attributes, used by the Hypervisor in the routing process. Tables 3.3, 3.4, 3.5, 3.6, and 3.7 present the main attributes for each item class.

Attribute name	Description	Type and range
id	Cloud's identifier (key attribute)	*Integer* ≥ 1
name	Cloud's name	*String*
provider	Cloud provider's name	*String*
securityLevel	Cloud's security level	*Integer* ≥ 1

Table 3.3: Cloud attributes

Attribute name	Description	Type and range
id	VM identifier (key attribute)	*Integer* ≥ 1
name	VM name	*String*
pid	VM identifier in public provider's network	*String*
cid	Cloud identifier	*Integer* ≥ 1
publicIp	Public IP address	*String*
privateIp	Private IP address	*String*
location	VM geographical location	*String*
gateway	Whether the VM is a cloud gateway	*Boolean*
deployed	Whether the VM is currently in use	*Boolean*

Table 3.4: VM attributes

Note that some attributes exist only for specific components. For instance, in Table 3.4, the *pid* attribute identifies the VM inside the public cloud provider. Due to its different characteristics, this attribute does not exist in private clouds. Note also that in tables 3.5, 3.6, and 3.7, we use the

concept of "node". In these tables, a node is either a container or an OVS switch, depending on the context. Importantly, in each topology the node ID is unique, to allow easy identification of source and destination components (i.e., the "from" and "to" attributes) in the link definition.

Attribute name	Description	Type and range
id	Node identifier (key attribute)	*Integer* ≥ 1
name	OVS switch name	*String*
vid	VM identifier	*Integer* ≥ 1
openflowVersion	Open Flow version (V1.X)	*Integer*[1–5]
dpid	OVS switch datapath identifier	*String*
securityLevel	OVS switch security level	*Integer* ≥ 1
dependabilityLevel	OVS switch dependability level	*Integer* ≥ 1
maxFlowSize	Maximum number of flows	*Integer* >= 1
bridgeName	Name of the OVS bridge	*String*
cpu	OVS Switch CPU strength	[0–100]
deployed	Whether the switch is currently in use	*Boolean*

Table 3.5: OVS attributes

Attribute name	Description	Type and range
id	Node identifier (key attribute)	*Integer* ≥ 1
name	Container name	*String*
vid	VM identifier	*Integer* ≥ 1
deployed	Whether the container is currently in use	*Boolean*

Table 3.6: Container attributes

Attribute name	Description	Type and range
id	Link identifier (key attribute)	*Integer* ≥ 1
from	Source node identifier	*Integer* ≥ 1
to	Destination node identifier	*Integer* ≥ 1
bandwidth	Link bandwidth strength	*Integer*$[0 - 100]$
delay	Link latency (milliseconds)	*Integer* ≥ 0
lossRate	Link loss rate (percentage)	*Integer*[0–100]
securityLevel	Link security level	*Integer* ≥ 1

Table 3.7: Link attributes

3.2.1.2 Tenant attributes

Virtual networks contain less components, namely they include the definition of virtual hosts, virtual switches, and virtual links. Virtual hosts and switches are associated both to a tenant network (through a tenant ID) and to a substrate component (a container or an OVS switch). The concept of node is similar to the substrate case presented previously. A node refers to a virtual host or a virtual switch, and is unique inside a tenant network topology. The current version of the hypervisor supports a one-to-one mapping between the substrate and virtual hosts, and a one-to-many mapping between substrate and virtual switches. In other words, a container is associated exclusively to a tenant network, while an OVS switch can implement various virtual switches, each one associated to a different tenant. Tables 3.8, 3.9, 3.10 list the attributes of virtual hosts, virtual switches and virtual links.

Attribute name	Description	Type and range
id	Virtual node identifier (key attribute)	*Integer* ≥ 1
name	Virtual host's name	*String*
ip	Virtual host's IP address	*String*
mac	Virtual host's MAC address	*String*
tenant	Tenant's identifier	*Integer* ≥ 1
mapping	Mapped physical node id (container id)	*Integer* ≥ 1

Table 3.8: Virtual host attributes

Attribute name	Description	Type and range
id	Node identifier (key attribute)	*Integer* ≥ 1
name	Virtual switch's name	*String*
tenant	Tenant's identifier	*Integer* ≥ 1
mapping	Mapped physical node id (OVS switch id)	*Integer* ≥ 1

Table 3.9: Virtual switch attributes

Attribute name	Description	Type and range
id	Link identifier (key attribute)	*Integer* ≥ 1
from	Virtual source node's identifier	*Integer* ≥ 1
to	Virtual destination node's identifier	*Integer* ≥ 1
route	Physical link ids included in the route	*ArrayofInteger*

Table 3.10: Virtual link attributes

3.2.1.3 Topology reply

The topology reply is translated into a JSON object. For each key/value pair, keys corresponds to the network components while values are arrays containing the component objects. The position in the array is irrelevant as each component is identified by its *id* attribute.

```
{ "cloud": [ <cloud1> .. <cloudN> ], "vm": [ <vm1> ... <vmN> ], "
   container": [ <container1> ... <containerN> ], "switch": [ <switch1>
   ... <switchN> ], "link": [ <link1> ... <link2> ] }
```

Substrate topologies contain *clouds*, *VMs*, *hosts*, *links* and *switches* objects, while tenant topologies only contain *hosts*, *links* and *switches* objects.

3.2.1.4 Example of topology request and reply

Figure 3.4 shows an example of a substrate network that hosts two virtual tenant networks shown in Figure 3.5. In the example we consider that all VMs belong to the same cloud ("id" = 1) and that each VM runs one OVS switch and two Docker containers.
Tenant1 and Tenant2 networks make a request for, respectively, two and four hosts. When the hypervisor performs the embedding, the mapping between virtual and substrate networks is performed as follows:

- **Tenant1**: *Host1* and *Host2* are mapped to *Container8* and *Container4*, respectively.
- **Tenant2**: *Host1*, *Host2*, *Host3*, and *Host4* are mapped to *Container1*, *Container2*, *Container3*, and *Container6*, respectively.

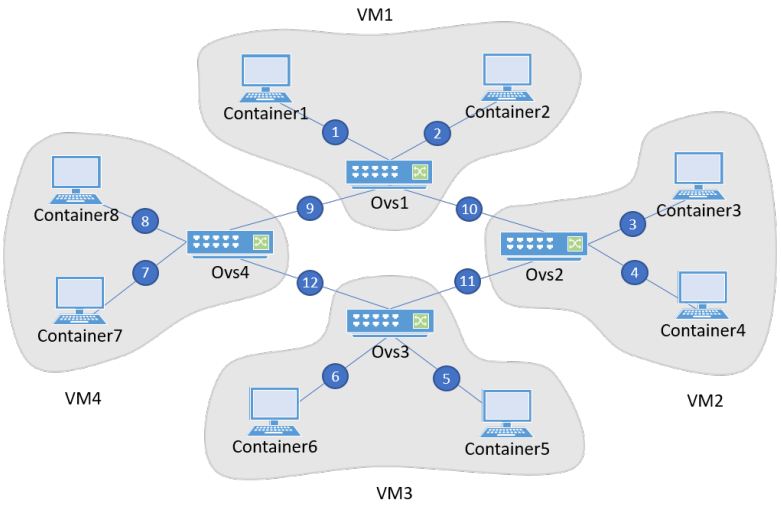

Figure 3.4: Example of substrate network

Note that the OVS2 switch is shared between both tenants, and that *Container6* and *Container7* remain unused.

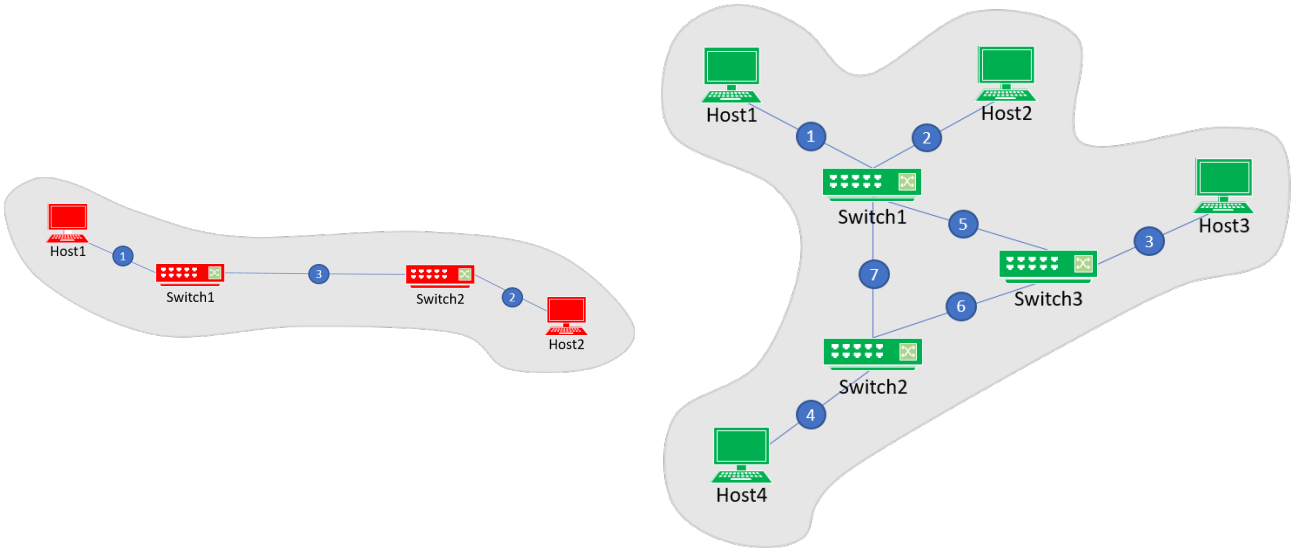

Figure 3.5: Example of virtual networks

Below is the JSON reply to the substrate topology request (**GET URL/topology?id=0**). For clarity sake, we omit some of the attributes in the reply (replaced by small dots).


```
{ "cloud": [ { "id": 1, "name": "cloud1" ... } ], "vm": [ { "id": 1, "
    name": "vm1", "cid": 1 ... }, { "id": 2, "name": "vm2", "cid": 1 ...
    }, { "id": 3, "name": "vm3" , "cid": 1 ... }, { "id": 4, "name": "vm4"
     , "cid": 1 ... } ], "container": [ { "id": 1, "name": "container1", "
    vid": 1 ... }, { "id": 2, "name": "container2", "vid": 1 ... }, { "id
    ": 3, "name": "container3", "vid": 2 ... }, { "id": 4, "name": "
    container4", "vid": 2 ... }, { "id": 5, "name": "container5", "vid": 3
     ... }, { "id": 6, "name": "container6", "vid": 3 ... }, { "id": 7, "
    name": "container7", "vid": 4 ... }, { "id": 8, "name": "container8",
    "vid": 4 ... } ], "switch": [ { "id": 9, "name": "ovs1", "vid": 1 ...
```

```
}, { "id": 10, "name": "ovs2", "vid": 2 ... }, { "id": 11, "name": "
ovs3", "vid": 3 ... }, { "id": 12, "name": "ovs4", "vid": 4 ... } ], "
link": [ { "id": 1, "from": 1, "to": 9 ... }, { "id": 2, "from": 2, "
to": 9 ... }, { "id": 3, "from": 3, "to": 10 ... }, { "id": 4, "from":
 4, "to": 10 ... }, { "id": 5, "from": 5, "to": 11 ... }, { "id": 6, "
from": 6, "to": 11 ... }, { "id": 7, "from": 7, "to": 12 ... }, { "id
": 8, "from": 8, "to": 12 ... }, { "id": 9, "from": 9, "to": 10 ... },
 { "id": 10, "from": 9, "to": 10 ... }, { "id": 11, "from": 10, "to":
11 ... }, { "id": 12, "from": 11, "to": 12 ... } ] }
```


Next, we present the JSON reply to the tenant1's topology request (**GET URL/topology?id=1**).


```
{ "host": [ { "id": 1, "name": "host1", "tenant": 1, "mapping": 8 ... },
   { "id": 2, "name": "host2", "tenant": 1, "mapping": 4 ... } ], "switch
   ": [ { "id": 3, "name": "switch1", "tenant": 1, "mapping": 12 }, { "id
   ": 4, "name": "switch2", "tenant": 1, "mapping": 10 } ], "link": [ { "
   id": 1, "from": 1, "to": 3, "route": [ 8 ] ... }, { "id": 2, "from":
   2, "to": 4, "route": [ 4 ] ... }, { "id": 3, "from": 3, "to": 4, "
   route": [ 9, 10 ] ... } ] }
```


Finally, we show the JSON reply to the tenant2's topology request (**GET URL/topology?id=2**).


```
{ "host": [ { "id": 1, "name": "host1", "tenant": 2, "mapping": 1 ... },
   { "id": 2, "name": "host2", "tenant": 2, "mapping": 2 ... }, { "id":
   3, "name": "host3", "tenant": 2, "mapping": 3 ... }, { "id": 4, "name
   ": "host4", "tenant": 2, "mapping": 6 ... } ], "switch": [ { "id": 5,
    "name": "switch1", "tenant": 2, "mapping": 9 }, { "id": 6, "name": "
   switch2", "tenant": 2, "mapping": 11 }, { "id": 7, "name": "switch3",
   "tenant": 2, "mapping": 10 } ], "link": [ { "id": 1, "from": 1, "to":
   5, "route": [ 1 ] ... }, { "id": 2, "from": 2, "to": 5, "route": [ 2 ]
    ... }, { "id": 3, "from": 3, "to": 7, "route": [ 3 ] ... }, { "id":
   4, "from": 4, "to": 6, "route": [ 6 ] ... }, { "id": 5, "from": 5, "to
   ": 7, "route": [ 10] ... }, { "id": 6, "from": 6, "to": 7, "route": [
   11 ] ... }, { "id": 7, "from": 5, "to": 6, "route": [ 9, 12 ] ... } ]
   }
```


3.2.2 Redirect all traffic request

This request is used to ask the hypervisor to redirect all flows sent to a specific virtual host – the original destination – to another one – the new destination (which could be any other virtual host). When this request is submitted to the hypervisor, the original destination virtual host will stop receiving any flow, which will be redirected to the new destination host.
The topology request syntax, arguments, and responses are given in Table 3.11.

Request	Arguments	Response
GET URL/redirect_all?	id=ID; origin_host_ip=IP_ADDRESS; new_host_ip=IP_ADDRESS	Boolean to indicate success or fail

Table 3.11: Redirect traffic request overview

The arguments refer to: the tenant identifier, which should be $id > 0$, the original destination host IP address, and the new destination IP address.

3.2.2.1 Response status codes

The response status codes are boolean and indicate if the action of redirecting all traffic from one virtual host to another one has been processed correctly. A request failure may be due to invalid syntax (malformed arguments) or because the server could not complete the requested action (non authorized action, for instance).

3.2.2.2 Reply content

The redirect_all reply is a boolean that if true indicates the action occurred properly, and false otherwise.

3.3 Code and documentation

The code for the network virtualisation core components can be accessed on the SUPERCLOUD private github repository at `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP4/sirius`. Instructions for installation are detailed in the README file.

Chapter 4 Self-management network security interfaces

This chapter describes the three components of the SUPERCLOUD network virtualization framework that form the self-management security module: a security monitor to detect security incidents, a service chaining component that allows users to compose their own security service chains, and a network security service that responds to these incidents. The chapter concludes by sketching ongoing directions to adapt this framework to the OpenDaylight ecosystem[1].

4.1 Self-management of security

Recent studies on security incidents and on abusive use of cloud services show that such incidents are very different and varied. It is therefore very difficult to anticipate certain types of incidents or attacks as threats are constantly changing. Moreover, malicious users tend to move some components of their infrastructures within the cloud itself. For instance, a research study on EC2 showed that Amazon might not be sufficiently responsive to mitigate and manage incidents in time to prevent security breaches and abuse of its users. Those observations motivate the need for an autonomous security management system enabling:

- **Detection of security incidents in a tenant network hosted over a multi-cloud**. This requires to collect and to process information from these infrastructures to have a complete view of the state of the network. The SDN model does not provide mechanisms to raise security alerts from a security equipment. Hence the interest of a new solution enabling these devices to send alerts to process them and to combine them with statistics sent from switches. Those features are mainly provided by the security monitoring component presented in Section 4.2.
- **Customizability of composition of security services and of responses to incidents**. Since the nature of attacks are different, security components for monitoring the network data plane and for defining reactions to security incidents need to be under user control. This user-centric approach enables to manage security to match users' networks and usage patterns. Hence the interest to use SDN to build a system enabling both supervision of the network data plane and supporting security policies to steer automatic responses to security incidents. Those features are mainly provided by the service chaining component presented in Section 4.3.
- **Automatic response to security incidents**. The response must correspond to coordinated and effective actions over all infrastructures, with some consistency guarantees. Those features are mainly provided by the network security component presented in Section 4.4.

All components for self-management of security are based on the Floodlight SDN controller. They benefit from the controller modularity, taking advantage of both its Java API to develop modules inside the controller (e.g., to handle security monitoring itself) and of its REST API to develop applications on top of the controller (e.g., in Python to handle service chaining). The security monitoring component notably makes extensive use of the Floodlight notification pattern based on event listeners for inter-module communications.

[1]The OpenDaylight Project is the largest open source project aiming to promote Software-Defined Networking (SDN) and create foundations for Network Functions Virtualization (NFV) [1].

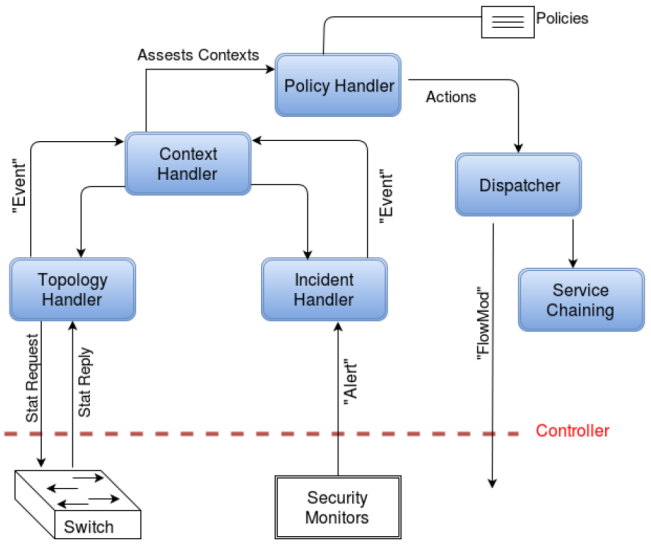

Figure 4.1: Security monitoring component: high-level architecture

4.2 Security monitoring component

The high-level architecture of the security monitoring component is shown in Figure 4.1. Next, we describe its main sub-modules and corresponding interfaces.

4.2.1 Topology handler

4.2.1.1 Overview

This module collects statistics regarding the network state by issuing requests to switches periodically. It computes the network bandwidth using Rx and Tx counters. It also provides a tunable notification service to the Context Handler for specific types of alerts (e.g., link congestion, high packet drop rate).

4.2.1.2 Interfaces

A first interface `IStatisticsService` is devoted to collecting statistics from switches, i.e., switches send statistics only upon solicitation by a request. For this purpose, two activations are performed:

- Activation of statistics collection by `IStatisticsService`. Collection is activated (resp. disabled) by invoking the `collectStatistics()` method, with a `True` (resp. `False`) argument.
- Thread activation to retrieve statistics collected by the `IStatisticsService` service. This operation is performed by executing periodically the `run()` method of this class.

A second interface `ITopologyAlertsListeners` is dedicated to allow the Context Handler to be notified when a particular event occurs, such as congestion on a link.

4.2.2 Context handler

4.2.2.1 Overview

This module defines *security contexts*. The definition of a context is the context itself and the preconditions for it to be considered valid. The preconditions concern the information collected: alerts and statistics brought up by security equipments and switches. For example, the system may be in a "malware" context if two preconditions are met: (1) a malware detection alert has occurred; and (2) a connection to a suspicious domain name has been triggered from an infected device.

When the first precondition for the context is met, a context instance is created. The context becomes active when all preconditions are fulfilled. Several types of contexts are distinguished for hosts, switches, or alerts.
When a context is active for a switch or host, the Context Handler notifies the Policy Handler by sending all necessary information: host address, switch to which it is attached, attacker address if it exists, and relevant context(s).

4.2.2.2 Interfaces

This module needs to subscribe to the notification services provided by Incident Handler and Topology Handler modules. It must therefore implement the corresponding interfaces (`ISecurityAlertService` and `ITopologyAlertsListeners`). It also provides another interface `IActiveContextAlertService` capturing the offered notification service towards other modules such as the Policy Handler when a context has become active.

4.2.3 Policy handler

4.2.3.1 Overview

This module implements the actions specified by the user-defined security policies. These policies describe reactions to trigger when a security context is active. For example, if a host is affected by the "malware" context, the reaction might be to put the machine in quarantine. Policies are stored using the JSON format following the Event-Condition-Action (ECA) model.
This module has access to user-defined security policies and receives notifications from the Context Handler. The latter notifies it when a host or switch is concerned by a context. After receiving an alert, the Policy Handler checks if an action is scheduled. If so, it notifies the Dispatcher module to execute the action. The Policy Handler implements very simple policy-based reactions. Much richer security policy management is also possible through the the network security component described in Section 4.4.

4.2.3.2 Interfaces

This module offers a REST API to allow users to define policies through POST HTTP requests. It must implement the `IActiveContextAlertService` interface to receive notifications from the Context Handler. It also has an outgoing interface towards the Dispatcher for execution of actions, either directly to install FlowMod rules in switches, or for processing by the service chaining component.

4.2.4 Incident handler

4.2.4.1 Overview

This module provides an interface with security equipments. It enhances the information collected by the controller for security monitoring of the data plane, as statistics provided by OpenFlow are insufficient. This module allows the security equipment to exchange information (alerts) with the control plane in a standard format (IDMEF or JSON). Such information will then be processed to be sent to the Context Handler module. This service can be customized to ensure that the Context handler module is notified only for specific alerts. This module does not check to see if an alert is present. It is "awakened" by the security equipment which will trigger the processing of the alert.

4.2.4.2 Interfaces

This module offers a REST API that allows network security functions to raise alerts with simple POST HTTP requests. It also offers a security alert notification service through the `ISecurityAlertService` interface, i.e., modules that register for this service will be notified each time an alert is received.

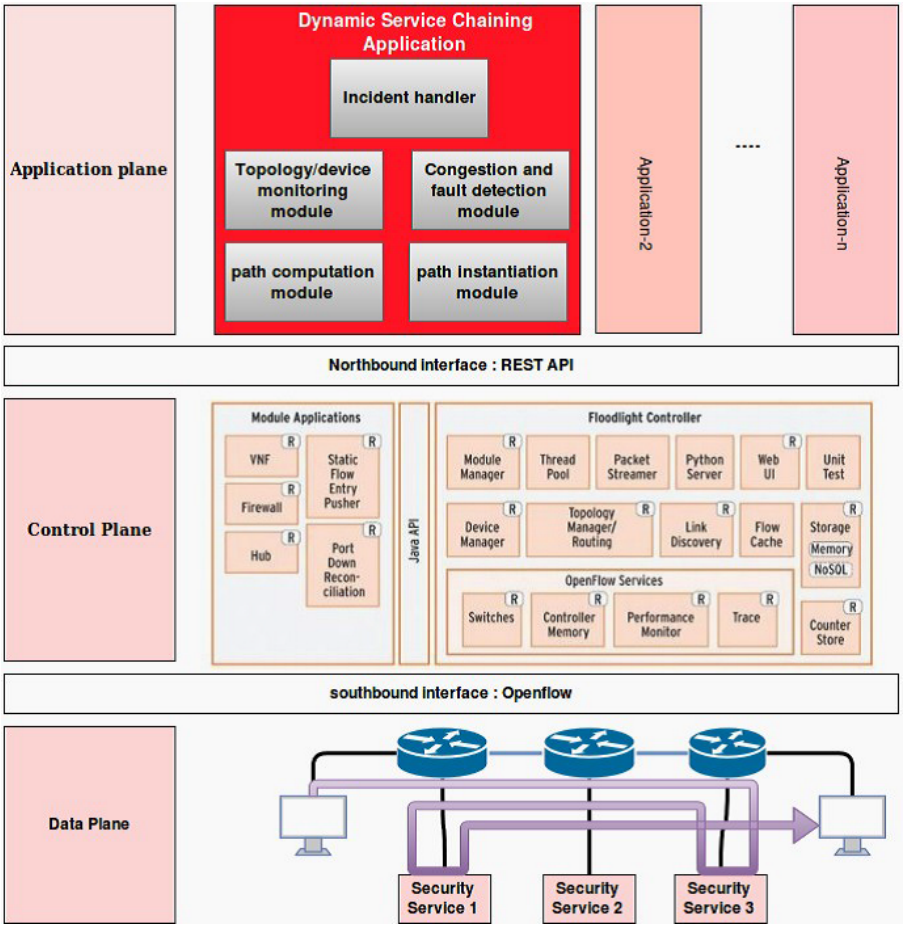

Figure 4.2: Service chaining component: high-level architecture

4.3 Service chaining component

Operators and cloud service providers are facing a real issue regarding management of middle-boxes commonly deployed in data centers, enterprise networks, etc. Such middle-boxes include security services such as firewalls or DPIs. The lack of reliable and mature routing protocol for traffic through a series of security services requires such operators and service providers to use complex, low-level and error-prone configurations. This leads to static configurations and increase service deployment time.
The service chaining component takes advantage of recent developments in SDN architecture. It takes the form of an application running over a centralized SDN controller, presenting an abstract view of the infrastructure. This allows the end-user to easily compose his own security service chains (online or off-line) in a multi-cloud environment.
The application runs on top of the Floodlight controller and uses its REST API to discover the topology, get traffic statistics or install the OVS in the switches. REST commands used are only valid for switches compatible with OpenFlow version 1.3, which must be explicitly activated in OVS.

The application architecture is shown in Figure 4.2. It includes 5 modules:

- **Topology monitoring module**: This module periodically sends requests to discover changes regarding the topology, flow and port statistics.
- **Congestion and fault detection module**: With the information returned by the Monitoring module, this module allows redirection of the flows in the event of a fault at the level of a link, a switch or a security service.
- **Path computation module**: This module enables to compute optimal paths between the source and the destination through a series of services while respecting the applicable constraints and minimizing a cost function.
- **Path instantiation module:** This module allows to translate the abstract path calculated by the Path computation module into FlowMod rules to be installed on the switches constituting the optimal path.
- **Incident handler module:** This module interfaces with an external security policy manager (e.g., the network security component, see Section 4.4). The aim is to interpret security alerts and to force some flows through specific paths to respond to detected security threats.

4.3.1 Topology monitoring module

This module gathers different information via the REST API of Floodlight in order to make it available to other modules. This data is stored in a database to facilitate its exploitation. This module then uses the data to construct a graph object that gathers all the relevant information and displays it dynamically to follow the evolution of flows as new channels are installed. Data that can be retrieved from the controller may be relevant to switches (e.g., packets received, transmitted, lost), end-device nodes, links, etc.

4.3.2 Path computation module

This module computes the complete multi-constrained path of a service chain (shortest path between successive pairs of equipments). It concatenates each partial path to obtain the total path taking into account several constraints.
Integration of the computing model is done in an iterative way: (1) defining the different flow routing constraints and an "objective to minimize" function; (2) translating these constraints into mathematical form and describing them in an object-oriented format to serve as input to a specialized solver for integer programming problems; and (3) integrating this code with the rest of the solution and validating the results of the optimization.
The constraints considered by the model are the following: list of security services; order of services; capacity of each security service; constraints of sovereignty.

4.3.3 Path instantiation module

Instantiation of a path translates a list of nodes (virtual switches, security service, and source and destination traffic servers) into OpenFlow rules that the SDN controller will install on each device in the data path. It takes as parameter the list of devices that the traffic must traverse. Routing is performed at layer 2.
This module traverses the path calculated previously, and installs the rules in the switches of the path. An example is shown in Figure 4.3[2].

[2]The module maintains a variable `current_src_mac` which represents the source MAC address of the traffic at each step of the route of the list and which is likely to change. After each traversal of a security device, the source MAC address changes, since the network packet is extracted and then re-encapsulated in a new Ethernet frame at the service level and will therefore carry the MAC address of the passing host.

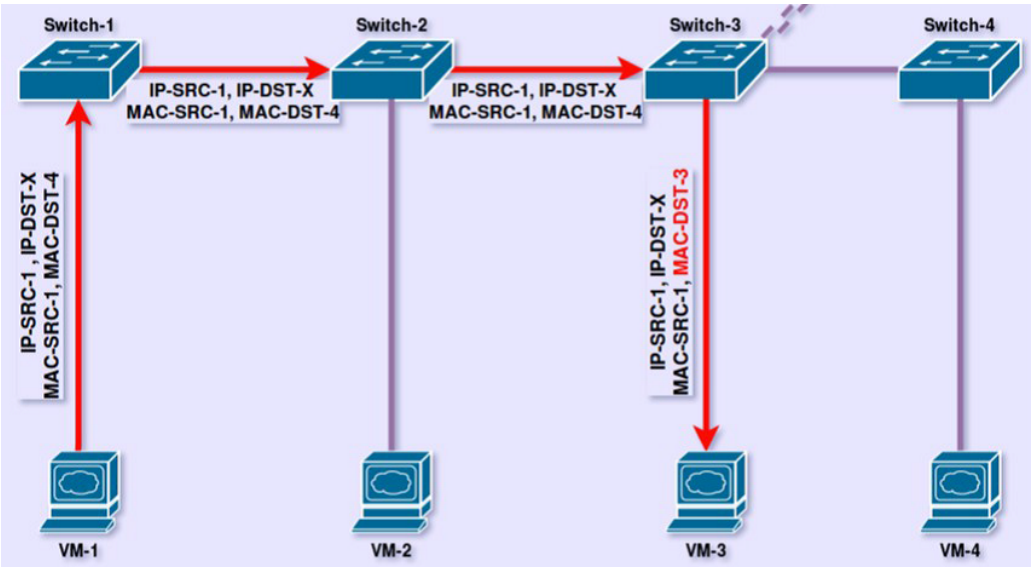

Figure 4.3: Path instantiation example

4.4 Network security module

The network security module aims at managing and deploying network security policies automatically. It interacts with the security monitoring tool which provides alerts and statistics about the SDN networks exposed by the network hypervisor.
This component is a sub-module of the network security self-management. It is deployed as an application on top of the network hypervisor. It reacts to the notifications received by the monitoring tool and instructs the network hypervisor to deploy the changes in order to dynamically adapt the network to the context of the environment. The reaction is chosen according to security policies. It can provide several reactions, such as:

- Modifying the path quality for a specific flow
- Redirecting a flow to another path
- Isolating a host
- Dropping a flow
- Instantiating new VMs or switches to change the topology and re-route flow according to the new topology

4.4.1 Components Overview

4.4.1.1 Workflow description

Our solution is based on two modules, a *Policy DataBase* (PDB) and a *Policy Decision Point* (PDP). These modules will interact with the monitoring tool described before and with the network hypervisor to achieve its goals. The operational workflow is given as follows:

1. An event is triggered at the ISP controller when a notification, which can be a security alert or a network status update, is received by the monitoring tool.
2. The monitoring module analyzes the notification and extracts the information of concern to be sent to the PDP.

3. The PDP selects the high-level action from the policy database based on the event and its corresponding conditions. The high-level action, bandwidth request, and flow information are then forwarded to the network hypervisor.

4. Based on the high-level action, the network hypervisor identifies the Policy Enforcement Points (PEPs) and computes the paths.

5. The resulting path(s) is/are then deployed via the network hypervisor by translating the high-level action into a set of OpenFlow rules that are distributed to the OpenFlow switches along the path(s).

4.4.1.2 Design Components

The functional components of the policy management system are discussed.
Policy Decision Point (PDP) is in charge of the global policy decisions. It firstly activates the context[3] based on the status of the networks and/or the received alerts. Then, based on the activated context and the alert information, the PDP requests the PDB for which actions to take (e.g, *redirect, drop, forward*). The resulting actions, together with the flow information, are finally sent to the network hypervisor to be enforced.
Policy Database (PDB) is essentially a repository containing the high-level security policies specified by the network administrator, without detailing the specific deployment strategy.

Listing 4.1: Syntax of high-level security policy

```
Event = {UDP_Flood | TCP_SYN | ICMP_Flood | DNS_Amplification | QoS_Request}
Condition = {Security_Class | Impact_Severity | ISP_Network_Status}
Security_Class = {Suspicious | Malicious | Legitimate}
Impact_Severity = {Low | Medium | High}
ISP_Network_Status = {Normal | Congested}
Action = {Redirect | Block | Forward}
```

Listing 4.2: A sample policy for suspicious traffic redirection in the face of UDP flood

```
<Policy name="Security_policy">
    <Event name="UDP_Flood">
    </Event>
    <Condition>
        <Security_class="Suspicious"/>
        <Impact_Severity="Medium"/>
        <ISP_Network_Status="Normal"/>
    </Condition>
    <Action/>
        Redirect
    </Action>
</Policy>
```

Specifically, security policies are structured using the Event-Condition-Action (ECA) model which we believe is suitable for dynamic policy management. In particular, each *Event* refers to a specific attack or incident and is associated with a set of rules. The rules are described as a set of *Conditions* that describes the context in which the attack or incident occurs, i.e. derived from the received security alert and the current network status. At last, the *Action*, is essentially a high-level action to be enforced against the identified flows.

[3]The context allows to fine-tune the policy that should be enforced, so as to minimize the side effects of policy enforcement.

Formally, we provide a syntax to deal with DDoS mitigation attacks, as shown in Listing 4.1. An *Event* may indicate one of several types of DDoS attacks or QoS requests. The *Condition* set allows to match the contexts in which the attack or incident has occurred. One context deals with the information extracted from the security alert, which includes the security class – e.g., *Malicious* labels from an attacker; *Legitimate* represents the flows that are considered benign in nature; and *Suspicious* denotes a mixture of malicious and legitimate traffic – and the impact severity of the target traffic on the customer network (*low, medium or high*). Another context captures the current status of the ISP network (i.e., either normal or congested). That last aspect enables the fine-tuning of the reaction policy, in order to reduce as much as possible the collateral damage on other flows. The list of possible actions is detailed in Sect. 4.5.2.
To illustrate the specification of the policy according to the given syntax, a sample policy addressing UDP flood attacks is given in Listing 4.2, which shows that the flows identified as UDP flood are labeled with a *suspicious* security class, and evaluated to affect the customer network with a *medium* impact. If the ISP network is in a *normal* status, the flows will be *redirected* elsewhere in the network.

4.5 Implementation Details

This section reports on the implementation details of the PDB and PDP and the alert handler.
In order to process alerts, a prototype *alert handler* has been implemented in Python as a RESTful API. It accepts requests from the customer network that contain the source and destination IP addresses, as well as the security class, impact severity and attack type fields. Table 4.1 shows the REST API exposed to the customer for issuing security alerts. The *incident handler* introduced in Sect. 4.2.4 could be alternatively used to collect alert information for the PDP.

Table 4.1: Request and Response overview

REST API	Parameters	Response
POST url/alert	source IP, destination IP, security class, impact severity	status=200
PUT url/action	source IP, destination IP, action=forward, drop, redirect	status=200

Regarding the *PDB*, the security policies are described in the XML language. Policies can be specified by the network administrator using the syntax shown in Listing 4.1.
The *PDP* is also implemented using Python. It can be deployed as a daemon to communicate with any controller, as long as the SDN controller exposes a RESTful API, such as the one displayed in Tab. 3.11, in the case of traffic redirection.

4.5.1 Interoperability

4.5.1.1 TOSCA Description

In Figure 4.4, we present a first draft of the TOSCA description of the network security module. TOSCA[4] is a standad topology description language developed by the OASIS consortium. This description will permit the deployment of the two services, the PDP and the PDB. Mainly, our component will take two inputs: a path to store the network security policies and a port number that will be used by the other components to communicate with our services based on a RESTful API.

[4]The specification of the TOSCA language version 1 can be found at: `https://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html`.

NetworkSecModule.yml

```yaml
tosca_definitions_version: tosca_simple_yaml_1_0_0_wd03
description: Network security Module.
template_name: Network-types
template_version: 1.0.0-SNAPSHOT
template_author: TSP

imports:
  - "tosca-normative-types:1.0.0-ALIEN12"
  - "alien-base-types:1.4.0-SNAPSHOT"

node_types:
  alien.nodes.NetworkSecModule:
    derived_from: tosca.nodes.SoftwareComponent
    description: >
      The TOSCA NetworkSecModule Type represents a network security  component
      that can be managed and run by a TOSCA Compute Node Type.
    tags:
      icon: /images/NetworkSecModule.png
    properties:
      component_version:
        type: version
        default: 1.0
      port:
        type: integer
        description: Port for the NetworkSecModule server
        default: 9999
        constraints:
          - greater_or_equal: 1
      DBpolicies_path:
          type: string
          default: "~/Desktop/policy_Set"
    interfaces:
      Standard:
        create:
          inputs:
            PORT: { get_property: [SELF, port] }
            DBpolicies_path: { get_property: [SELF, DBpolicies_path] }
          implementation: scripts/install_NetworkSecModule.sh
        start: scripts/start_NetworkSecModule.sh
```

Figure 4.4: Network Security Module: TOSCA Description

4.5.2 Interface

The following table details the set of possible actions designed with our solution.

Action name	Description
Forward_Middlebox *Middlebox* belongs to {firewall, NAT, IDS} **Examples:** a Forward_firewall b Forward_firewall_NAT c Forward_NAT	According to the network or system context, our solution may request from the network hypervisor to forward through a middlebox (e.g., firewall, NAT) before reaching the destination. **Example:** Forward_firewall_NAT In a network, a policy can specify that all the traffic should traverse a firewall and then a Network Address Translator (NAT) before accessing a web server. In this scenario, when a flow enters the network, forwarding rules are deployed in the downstream switches based on its destination IP address (e.g, web server) to forward the flow through the firewall and NAT devices.
Redirecting_Low_Bandwidth_Path	This action specifies to detour a flow through another path having a lower bandwidth. In this case, the controller dynamically deploys rules in the switches along the low-bandwidth path. In OpenFlow, we can modify existing rules in the switch flow table. Likewise, we can install fresh rules for the flow to process in different switches along the new path.

Action name	Description
Redirecting_Network_Service_Functions *Network_Service_Functions* belongs to {firewall, NAT, IDS}	This action means that we need to activate and use a service function that should be installed in the controller to act as a firewall, IDS or a NAT service. In this case, when a flow needs to be processed through these service functions, rules are deployed at a switch or multiple switches to redirect the flow towards them.
Drop	This action asks the network hypervisor to block a flow at the ingress switch in the network. The empty action in OpenFlow specifies the drop rule.
Redirect_all	This action means that the controller has to redirect all the flows sent to a specific server to another one (a replicate or a second server). In this case, the main server will not receive flows anymore.
Redirect_new	This action means that any flow related to a new session needs to redirected to another server (i.e., a replicate or a second server).
Replicate	This action means that the controller has to forward any flow to a backup server. This means that all packets related to a specific flow should be sent simultaneously to the main server and the backup server.
Rate_Limit	This action means that a flow has to be rate-limited to reduce the congestion in the network. We can set the maximum rate and the minimum rate in the queue of the OpenFlow switch. For instance, queue 0 can specify a maximum rate of 10 Mbps and queue 1 a maximum rate of 5 Mbps. When the flow arrives at the switch, the action can be specified to forward the flow to a specific queue based on its IP address for rate limiting.
On-demand QoS Examples: a) Gold QoS Request b) Silver QoS Request c) Bronze QoS Request	This action specifies that a client can ask its service provider to provide a high QoS or low QoS to flows depending on the network conditions (i.e., congested or normal) a) It specifies that when a client requires a gold QoS then the flow should be redirected through a path with high bandwidth. For example, we can set the bandwidth for gold class to 200 Mbps. b) It specifies to forward the traffic through a silver bandwidth path. For example, for silver bandwidth class, the bandwidth can be set to 150 Mbps. c) It specifies to forward the traffic through a bronze path. For instance, the bandwidth can be set to 100 Mbps for the bronze class.
Isolate_Host	This action means that a host should be isolated from other hosts in the network. That is, the host should not be able to send and receive packets from outside hosts.

4.6 Code and documentation

The security monitoring and appliance chaining components are being put in open source. At the time of publication of this document, final authorizations are needed before the code can be fully released. The code then can be accessed on github for Orange open source software[5]. It will also be published on the SUPERCLOUD private repository[6]. Repository access may be granted by sending an email to `marc.lacoste@orange.com`. A more complete description of the system can be found in Chapter 5 of SUPERCLOUD D4.2 [15]. Further documentation about the software is also distributed together with the code release.
The code for the network security module can also be accessed on the SUPERCLOUD repository[7]. Instructions for installation are detailed in the README file.

4.7 Ongoing work: a security agent for OpenDaylight

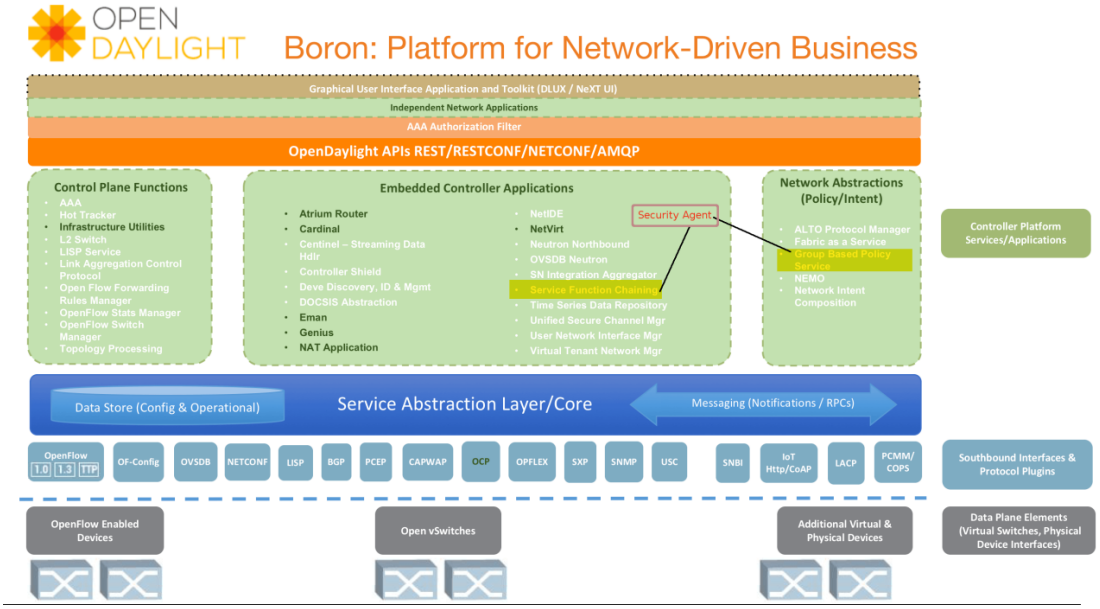

Figure 4.5: A security agent for OpenDaylight (adapted from [1])

Some of the security features of the previous components may be adapted to the OpenDaylight ecosystem where a set of Network Functions have already been developed in the SDN controller. Through its MD-SAL (Model-Driven Service Abstraction Layer), different network functions can be reused and aggregated to build new network functions.
To perform this adaptation, a security agent (northbound application) for OpenDaylight is currently developed. This agent uses existing functions like GBP (Group-Based Policy) and SFC (Service Function Chaining). Network flows are first identified by GBP. Through different policies, flows are then re-directed to different service chains. SFC enables dynamic chaining of security functions to map the identified flow through the defined policy.
In the scenario shown in Figures 4.5 and 4.6, several types of flows are identified and chained to different security functions. For instance, h35 and h36 are chained to different security functions like firewalling and IDS. A PDP (Policy Decision Point) will be added later on to the security agent to enable local security decisions for the whole network through the SDN controller.

[5] `https://github.com/Orange-OpenSource`
[6] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP4/monitoring` and `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP4/service-chaining`
[7] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP4/network-security`

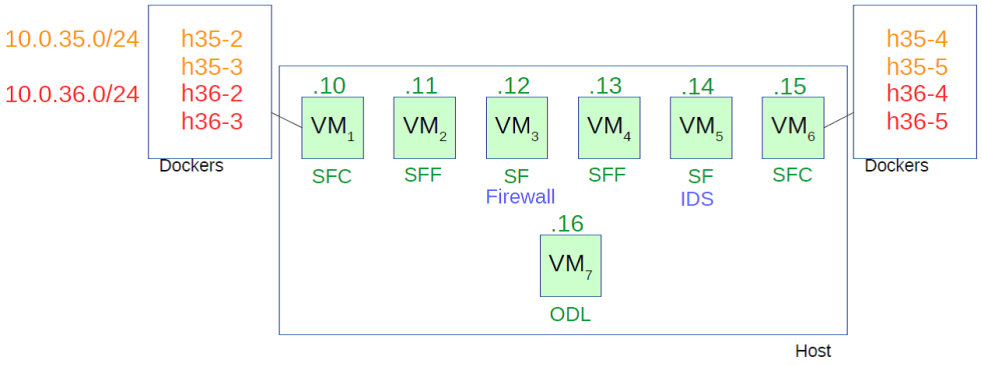

Figure 4.6: Chaining flows to different security functions

Chapter 5 Conclusions

This document forms part of the deliverable that presents the proof-of-concept prototype of the multi-cloud network virtualization infrastructure. Our purpose was to describe the architecture of the network framework, the APIs of its main components, and information on how to access and run the software developed.

- We have started by an overview of the architecture that focused on its core components: the multi-cloud orchestrator and the network hypervisor.
- We then presented the internal and external APIs for all components. Namely, the interfaces from the orchestrator and hypervisor core modules, and from the self-managed security services, including security monitoring, service chaining, and network security.

The next, final steps are dedicated to integration, an effort already in process that includes:

- Integration into the substrate infrastructure – that is currently composed of a private cloud from FFCUL and a public cloud (Amazon) – of another private cloud (the IMT testbed) and another public cloud.
- Integration of the self-management security solutions into the network virtualisation platform.
- Integration of the the storage service Janus (from Work Package 3) into the platform.
- Integration of the Maxdata and Phillips use cases defined in Work Package 5.

Bibliography

[1] OpenDaylight. `https://www.opendaylight.org/`.

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

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

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

[5] 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.

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

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

[8] 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.

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

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

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

[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] Fernando M. V. Ramos, Nuno Neves, Marc Lacoste, Nizar Kheir, Max Alaluna, André Mantas, Luis Ferrolho, José Soares, Grégory Blanc, Fabien Charmet, and Khalifa Toumi. D4.2 - Speci cation of Self-Management of Network Security and Resilience. *SUPERCLOUD*, 2016.

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

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

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