



Complete Dynamic Multi-cloud Application Management

Project no. 644925

Innovation Action

Co-funded by the Horizon 2020 Framework Programme of the European Union



Call identifier: H2020-ICT-2014-1

Topic: ICT-07-2014 – Advanced Cloud Infrastructures and Services

Start date of project: January 1st, 2015 (36 months duration)

Deliverable D5.2

Specification of network management and service abstraction

Due date: 31/03/2016

Submission date: 11/05/2016

Deliverable leader: José Aznar <I2cat Foundation>

Dissemination Level

- | | |
|-------------------------------------|---|
| <input checked="" type="checkbox"/> | PU: Public |
| <input type="checkbox"/> | PP: Restricted to other programme participants (including the Commission Services) |
| <input type="checkbox"/> | RE: Restricted to a group specified by the consortium (including the Commission Services) |
| <input type="checkbox"/> | CO: Confidential, only for members of the consortium (including the Commission Services) |
-

List of Contributors

Participant	Short Name	Contributor
Interoute S.P.A.	IRT	
Sixsq SARL	SIXSQ	Rob Branchat
QSC AG	QSC	
Technische Universitaet Berlin	TUB	
Fundacio Privada I2CAT, Internet I Innovacio Digital A Catalunya	I2CAT	José Aznar, Isart Canyameres, Eduard, Escalona, Adrián Roselló, Óscar, Moya.
Universiteit Van Amsterdam	UVA	
Centre National De La Recherche Scientifique	CNRS	

Table of Contents

List of Contributors.....	3
Figures Summary.....	6
Tables Summary.....	7
Executive Summary	8
1 Introduction.....	9
2 Update on WP5 progress. Analysis of limitations and decisions taken	10
2.1 Access to IaaS network resources.....	10
2.2 Utilization and integration with Cloud Service Provider (CSP) tools.....	11
2.3 Integration with SlipStream.....	12
3 OpenNaaS – CNSMO	14
3.1 Update on the OpenNaaS framework architecture.....	14
3.2 CNSMO Architecture	15
3.3 OpenNaaS-CNSMO service modelling	16
3.4 CNSMO interaction with SlipStream.....	18
3.5 OpenNaaS-CNSMO framework repositories and system data sheet.....	18
3.6 OpenNaaS-CNSMO main benefits and added value.....	19
4 CNSMO network services specification.	20
4.1 OpenNaaS-CNSMO service concept.....	20
4.2 Bootstrapping process: Deploying and running the network services.....	21
4.3 Multi-cloud VPN service	22
4.3.1 VPN service components	22
4.3.2 VPN implementation – Chosen technologies	23
4.3.3 VPN service workflow	23
4.3.4 VPN service PoC and Demonstrator.	25
4.3.5 OpenNaaS-CNSMO VPN network service code repository.....	25
4.4 Firewall service	25
4.4.1 Firewall service components	26
4.4.2 Firewall implementation – Chosen technologies	26
4.4.3 Firewall service workflow	26
4.4.4 Firewall service PoC and Demonstrator.....	27
4.4.5 CNSMO Firewall network service code repository	28
4.5 Load balancer service	29
4.5.1 Load Balancer service components	29
4.5.2 Load Balancer implementation – Chosen technologies	30
4.5.3 Load balancer service workflow	30
4.5.4 Load balancer service PoC and Demonstration	32
4.5.5 CNSMO Load Balancer network service code repository	32
5 Network service management and abstraction.....	33
5.1 Network management.....	33

5.2	<i>Network service abstraction</i>	33
6	Summary of network accomplished requirements	34
6.1	<i>Achieved network features and services</i>	34
7	Monitoring of other cloud networking solutions: BEACON and SWITCH projects	37
8	Next Steps	38
	Appendix A – “How to run” the VPN service demonstrator	39
	Appendix B – “How to run” the Firewall service demonstrator	42
	References	45

Figures Summary

Figure 1 – OpenNaaS deployment location to provision network services. (a) Former approach. (b) Current approach.	11
Figure 2 – OpenNaaS deployment and interaction while provisioning network services on top of the CSPs.	12
Figure 3 – OpenNaaS-CNSMO architecture and modules.	15
Figure 4 – Generic structure of an OpenNaaS-CNSMO service agent layout.	16
Figure 5 – OpenNaaS-CNSMO generic service lifecycle.	17
Figure 6 – OpenNaaS-CNSMO bootstrapping process to deploy networking services.	22
Figure 7 – VPN service modules example deployment.	23
Figure 8 – VPN service workflow.	24
Figure 9 – Firewall service workflow.	26
Figure 10 – Firewall service deployment layout.	28
Figure 11 – Load Balancer service deployment example.	30
Figure 12 – Load Balancer deployment workflow.	31
Figure 13 – VPN application deployment screenshot.	40
Figure 14 – VPN service deployment in “ready” state screenshot. Detail of the VPN server.	40
Figure 15 – Firewall service configuration deployment screenshot.	42
Figure 16 – Overview of the CYCLONE Firewall demonstrator deployment screenshot.	43
Figure 17 – Screenshot of the IPTABLES –L command outcome result.	44
Figure 18 – Screenshot of the script to launch the test	44

Tables Summary

Table 1 – OpenNaaS-CNSMO code repository location. Modules, languages and technologies.....18

Table 2 – VPN service repository and technologies25

Table 3 – Firewall service repository and technologies28

Table 4 – Load Balancer service repository and technologies32

Table 4 – Summary of network requirements imposed by the UCs, CSPs and other CYCLONE software components.36

Executive Summary

Deliverable D5.2 reports on the work and steps taken in order to satisfy the requirements retrieved in D5.1 and D3.1, by defining, implementing and demonstrating specific network micro-services integrated as part of the CYCLONE cloud federated solution. It includes the specification and implementation of the OpenNaaS-CNSMO component and integration with the SlipStream software platform, the complete definition of the network service structure for CYCLONE and the specification of the first network services that were identified as “high-priority” ones as they address some of the requirements imposed by the use cases and the other CYCLONE software tools.

1 Introduction

Previous deliverable D5.1 [D5.1] focussed on the specification of the network service requirements imposed by the use cases and the different CYCLONE components and tools brought to the project.

Starting from previous requirements, WP5 activity has taken the steps in order to satisfy those requirements by defining, implementing and demonstrating specific network micro-services integrated as part of the CYCLONE cloud federated solution and make them extensible to the CYCLONE users and tools. Deliverable D5.2 reports on the – on-going – work done to achieve this, so that the integration of networking aspects with purely federated clouds, allows users to request a set of coordinated network and IT resources in an easy and transparent way while operating dynamic deployments of distributed applications.

More specifically, the major outcomes of this deliverable are:

- Specification and implementation of the OpenNaaS Cyclone Network Services Manager and Orchestrator component (OpenNaaS-CNSMO) to provide network services in the context of CYCLONE project (and potentially in any other cloud federated environment).
- The complete definition of the network service structure for CYCLONE.
- The integration OpenNaaS-CNSMO with SlipStream application provisioning engine [SlipStream] in order to enable the provisioning of network services allowing delivering new capabilities from which stakeholders will benefit.
- The specification of the 3 first network services that were identified as “high-priority” ones in previous D3.1 [D3.1] and D5.1 [D5.1], as they already address some of the requirements imposed by the use cases and the other CYCLONE software tools.

The structure of the document is as follows: **Section 2** provides an analysis of the difficulties found while aiming to bring cloud networking services to CYCLONE. There have appeared a number of limitations that definitely impact the way network services definition and deployment were initially planned. In this section we explain most relevant ones and the decisions taken to come up with a suitable solution. In **section 3**, it is presented the OpenNaaS-CNSMO component solution as part of the CYCLONE software to implement the network services. More in detail, it is explained the need of the CNSMO module as part of the CYCLONE solution, the architecture design and implementation, the CNSMO service model structure and the integration with the SlipStream software to facilitate the services. In **section 4**, we present the previous mentioned network services: VPN, Firewall and Load Balancer services, including the workflow and code repositories. The VPN service has been already publicly demonstrated and the complete “*how to*” can be found in **Appendix A**. In **Section 5** it is provided an overview on the “*network management*” and “*network service abstraction*” concepts. In **section 6** it is summarized the status of the already accomplished, on-going and planned implementation and integration efforts to extend the offer of networking services in the CYCLONE project. Some other H2020 projects are also proposing cloud network solutions. The CYCLONE WP5 team has contacted them to find out more on their scope and approaches, so that potential collaborations may be arranged during the second half of the project for reciprocal benefit purposes. The outcomes of these first interactions are reported in **section 7**. Finally, in **section 8** we draft the next WP5 steps.

2 Update on WP5 progress. Analysis of limitations and decisions taken

During the second half of Y1 some limitations on the design, implementation and deployment of network services as part of the CYCLONE solution have arisen. Some of them were already identified in previous D5.1 report [D5.1] – see section 2.7 – and some others have appeared during WP5 work and activity with other WPs, especially WP3 (use cases), WP6 (Application Deployment and Management) and WP7 (Testbed Infrastructure). While these limitations clearly have an impact in the former WP5 plan and the initial proposed solution, WP5 target while bridging cloud networking to cloud federation environments remains intact. The most relevant limitations and decisions taken are next explained.

2.1 Access to IaaS network resources

In order for OpenNaaS to model and expose network services, the work started from the premise of having access to IaaS providers' network resources (either physical or logical network resources) in order to gain control and management over them and provide network services by exposing concrete capabilities to the CYCLONE users.

Nevertheless, the market reality is different and comes hand in hand with the different partners bringing cloud IaaS to the project: mostly due to security reasons, it's not possible to enable access to OpenNaaS to the cloud network resources (switches, Top of the Racks (ToRs), firewalls, etc.), since the provided IaaS is part of a larger infrastructure and utilizes network resources of their production environment. More specifically:

- IRT is only able to provide access to the servers and the ToR(s) connecting them.
- QSC is only able to provide access to the servers.
- CNRS is only able to provide access to the servers.

This means that OpenNaaS is consequently not able to, for example, gain access and control of the firewall of a private production cloud belonging to the testbed facilities and therefore providing to other CYCLONE components or the application layer enhanced firewalling services based in the cloud firewall management.

On one hand, this situation limits the capacity of enabling network control and management. On the other hand, this is a typical situation in today's hybrid cloud scenarios: Nor IaaS or PaaS provider may enable access to the network production resources, since this constitutes a risk that may impact the normal behaviour of the overall infrastructure that sustains the business. Thus facing such a situation has driven WP5 effort to move forward in another direction while providing network services to cloud federated environments.

The initial pretended provisioning approach to implement and provide network services (see Figure 1.a) is no longer valid, since OpenNaaS is not able to "*sit on top*" of the IaaS network resources. It is not possible to model resources as proposed in D5.1 and delegate its control to other CYCLONE modules. Thus, it has been proposed an alternative approach which aims to design and implement Virtualized Network Functions (VNFs) to shape network micro-services. These network services are deployed together with the

applications, by means of SlipStream, so that the OpenNaaS does not work on top of the network service, but it is directly integrated with SlipStream to enable the network micro-services closer to the application layer (see Figure 1.b).

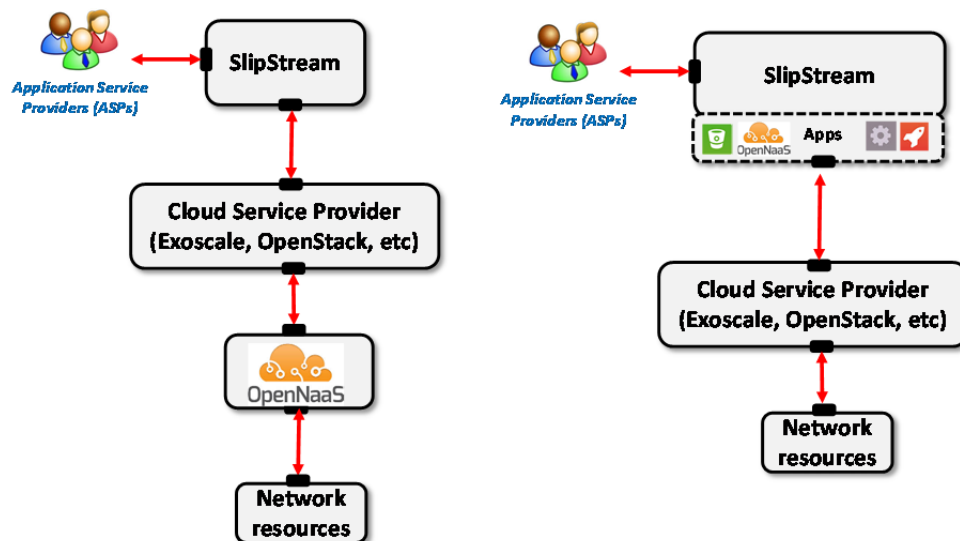


Figure 1 – OpenNaaS deployment location to provision network services. (a) Former approach. (b) Current approach.

To this end, OpenNaaS has been refactored to a new version of the software called CNSMO (Cyclone Network Services Manager/ Orchestrator). CNSMO constitutes a lightweight version of OpenNaaS specifically devoted to the adoption of cloud networking services in cloud federated environments more suitable than the former OpenNaaS platform to this end. The details of CNSMO are explained in section 3 of this deliverable.

Nevertheless, WP5 team will explore further options that may enable at least to demonstrate the support of network resources configuration offered for instance to provide L2 or L3 connectivity services with certain QoS granularity.

2.2 Utilization and integration with Cloud Service Provider (CSP) tools

Previous limitation (no access to network resources) could have driven the efforts to provide and deploy network services working on top of the CSP platforms managing each of the testbed infrastructures. The initial plan included CNRS-LAL private infrastructure (managed by StratusLab [StratusLab]), IRT and QSC private infrastructure (managed by OpenStack [OpenStack] instances) and the utilization of some public cloud facility (AWS [AWS], Exoscale [Exoscale] and others). Leaning on those (and potentially other) CSPs, it was considered the possibility to utilize OpenNaaS to work as network plugin and enable specific network options as shown in Figure 2.

Nevertheless, StratusLab currently presents a limited community effort and the outbreak of novel emerging solutions that count with more support and it is likely that CNRS-LAL moves to OpenStack. OpenNaaS would have made sense to StratusLab as Neutron does to OpenStack, so that the solution would be restricted to an ad-hoc integration with Neutron so that the set of network features and services would be constrained to OpenStack Neutron and public cloud options and roadmap, fact which may limit the scope while addressing the requirements imposed by CYCLONE use cases in terms of cloud networking.

Thus, this option has not been discarded, since still the provisioning of network services can benefit from the APIs exposed by the CSPs, but it has been considered as an additional and complementary effort to drive the provisioning of networking services in a later stage of the project.

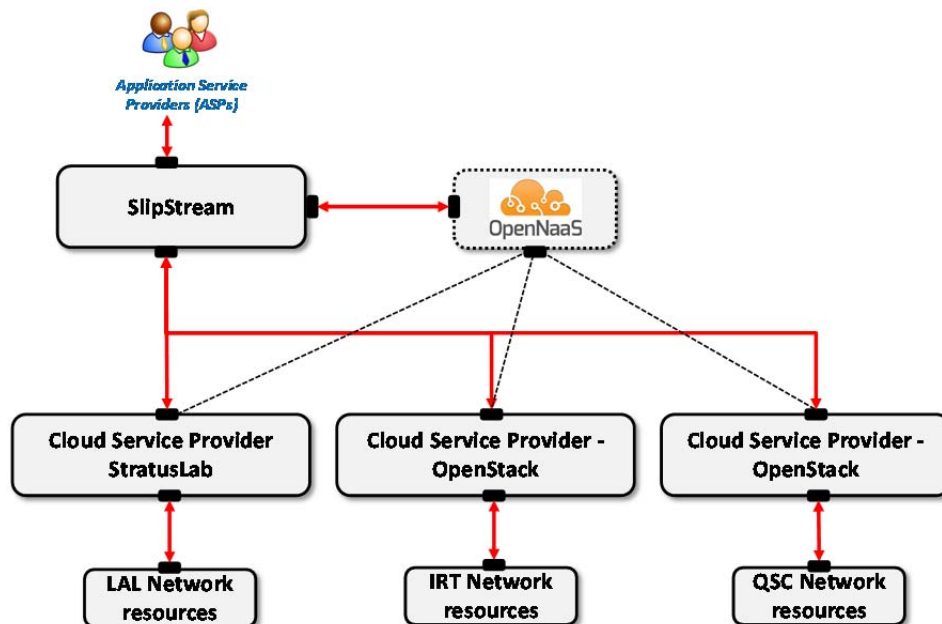


Figure 2 – OpenNaaS deployment and interaction while provisioning network services on top of the CSPs.

2.3 Integration with SlipStream

CYCLONE networking services were originally planned as an extension of the SlipStream tool. As so, it was designed that SlipStream itself would contact the networking services provider (OpenNaaS) at some point during the deployment, and receive from it a set of instructions to add the network services in the ongoing deployment. This approach would have allowed OpenNaaS to order SlipStream to modify a user deployment by adding custom application components (VMs) but also modifying user defined ones by adding additional commands in their recipes. In this approach, the user would only have needed to select the network services to be enabled as part of the deployment run form.

However, this approach would have made SlipStream dependent on a third party software (OpenNaaS) with the associated risks, but more important, it would have required a major rewrite due to the live changes made in an application while it is being deployed, moving away from the production SlipStream version and therefore, limiting the potential future exploitation of the network provisioning software as part of SlipStream offer.

With the approach which it has been finally agreed upon, CYCLONE networking services are provided by application components inside the user defined application. This puts OpenNaaS and the services offered both as application components. This approach circumvents the need of mentioned SlipStream major rewrite but, in the other hand, it has some impact in the way CYCLONE networking services may be deployed and in the user interaction with the system.

The set of features that are made available to OpenNaaS to deploy the networking services is reduced. A clear example of this is that OpenNaaS is no longer able to tell SlipStream to deploy VMs on demand. Instead, it has to use existing application components. This happens because application components are

not able to interact directly with SlipStream, but only with the orchestrator SlipStream deploys in each run. And this orchestrator does not allow changing the VM images available to the deployment and deploying new components that are not part of the original application.

To make use of CYCLONE networking services, the user defining the application has to explicitly add to it the components offering them. Thus, the user must not only know which services are available and select desired ones, but also know which components are required for each service and add them manually to the application.

Within this context, CYCLONE networking services are offered by software modules integrated in SlipStream as application components. Application components consist of a VM image with the appropriated software installation to run the software modules offering the network services, together with a SlipStream recipe (a set of scripts) which contains the deployment instructions to be run by SlipStream. These modules may take advantage of the deployment information offered by SlipStream via a client tool to coordinate the deployment workflow and ensure networking services are triggered in the appropriated phase (e.g. VPN is setup before application configuration takes place, as the application must be configured afterwards to use VPN enabled interfaces for internal traffic).

Previous limitations have directly impact on the initial proposed solution for the development and provisioning of network services. Also the interaction and integration among OpenNaaS and other CYCLONE software components differs from the initial plan. The OpenNaaS-CNSMO platform constitutes a suitable solution to deploy network services and make it available to the CYCLONE applications and use cases.

3 OpenNaaS – CNSMO

OpenNaaS-CNSMO is the platform providing the CYCLONE networking services.

3.1 Update on the OpenNaaS framework architecture

This section presents SlipStream and the other CYCLONE modules and context enforcements that have lead us to CNSMO.

The OpenNaaS framework aims to control the network elements in a vendor independent and device independent way. OpenNaaS offers high level APIs to the network operators that abstract the different configuration interfaces of the devices. In this way, the network operators focus the configuration of the network in a service driven model. This provides abstraction of the different vendor configurations and enables an automation framework easy to manage, scale and extend. The scope of the OpenNaaS framework is to potentially control any network, from a small enterprise network providing L2 connectivity to every employee to a big cloud service provider providing L2, L3 and L4 services to their customers.

Although since the last few years the Software Defined Networking and Network Function Virtualization (SDN/NFV) paradigm has gained a lot of strength and popularity and seems the way to go for new CSPs and current SPs, OpenNaaS has tried to adopt this paradigm by supporting some SDN controllers and standards. The goal of OpenNaaS is to manage and configure the network and its services and not provide and configure Cloud resources. For example, in an NFV framework, OpenNaaS can provide the connectivity between a router or a switch and the VNF input and output lying in computational resource.

In CYCLONE's context SlipStream is the front-end framework that abstracts the way that cloud applications are deployed and configured. It provides easier application deployments while choosing the optimal CSP according to the user needs. The connectivity of the Cloud resources provided by Slipstream is somehow limited. The reason for that is that Slipstream interacts with the services provided by the CSPs through their API, not realising a coordination of the network configuration or exposing network parameters to the user.

One of the goals of CYCLONE is to provide network services to the SlipStream user deployments. In this sense, OpenNaaS and SlipStream are complementary and by integrating both of them a full and end-to-end (from creating and configuring a VM to enable the connectivity to different Network services) Cloud deployment is provided. However, SlipStream only provides IP overlay networks (provided by the CSPs) with limited flexibility and configurability. Moreover, the goal of OpenNaaS is to provide and manage overlays to work on top of them. The ideal solution to fully integrate OpenNaaS and SlipStream is to let OpenNaaS control the underlay (or part of it). However since SlipStream works with well-known CSPs as Azure or Amazon Web Services, the access to configure their network devices is uncertain.

Instead, OpenNaaS has been forked in a CYCLONE adapted framework code-named CNSMO, that provides network services using the OpenNaaS concept and abstractions in overlay topologies. The reason to fork OpenNaaS into CNSMO is that the overlay has slightly different requirements. Now, CNSMO has to enable network services inside overlays, that means, user deployments that are being used and paid by users, so the CNSMO modules need to be as transparent as possible to the users not only in terms of using them but in causing minimum impact in the user's' resources usage. CNSMO has to provide also software solutions

implementing the network services, something that formerly was out of scope of the original OpenNaaS framework.

3.2 CNSMO Architecture

CNSMO is a lightweight micro-services framework developed at i2CAT. Leveraging the base concepts of Apache Mesos [Mesos], CNSMO is a distributed platform defining a basic service API and service life-cycle, together with an inter-service communication mechanism implementing the actor model and supporting different communication protocols. The system is capable of deploying and running multiple services in both local and remote environments.

CNSMO architecture is composed by the following elements (see Figure 3):

- CNSMO Core Agent
- Distributed System State
- CNSMO Service Manager

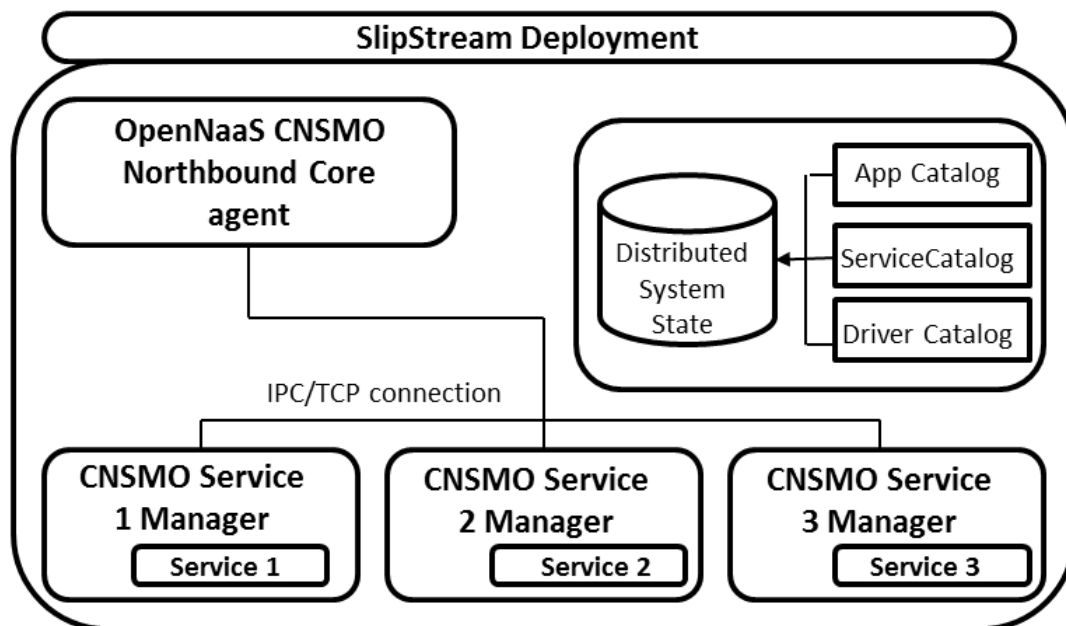


Figure 3 – OpenNaaS-CNSMO architecture and modules.

The **CNSMO core** contains a CNSMO agent running the context that enables all the modules required to deploy a service. It bootstraps all the required modules to deploy a network service.

The **Distributed System State** is used to store the application state, so it can recover from failure or system shutdown. By using it intensively, modules can become stateless which turns out to be very useful when scaling. Moreover, the System State has a message queue that is used by every component to communicate to each other's. It offers serialization mechanisms providing implementation independency between modules. It also offers load balancing features, useful to distribute the application load between multiple instances of the same CNSMO component, in this case, multiple instances of the same specific network service module (e.g. VPN service module). It also allows a framework where every CNSMO context announces itself when it is ready and allows other service to subscribe to status changes of services enabling service orchestration if needed.

CNSMO Agents are wrappers for a specific service providing CNSMO the ability to manage that service. Thanks to these agents CNSMO discovers the services, is able to spawn or delete service instances but also orchestrate service deployment between many VMs. CNSMO agents are active and report to the CNSMO core (more specifically to the distributed system state) when launched. A different agent is required for each service in each machine. A CNSMO Agent is composed by the modules shown in Figure 4:

- **Service API:** is the API that The Core Agent talks to, allowing controlling the lifecycle of the framework.
- **System state API:** Exposes a set of methods to remotely store and react to changes in the state of each system component.
- **Deployment API:** Exposes a set of methods to remotely spawn a service.
- **System.d/CLI driver:** It is the lifecycle Driver telling the framework how to launch/deploy the service. In the context of CYCLONE this can be a Docker driver, for instance.
- **Service management logic:** Internal service logic. For example, it may establish a VPN or manage firewalling rules among others, depending on the service intention. It is the implementation of the service itself.

OpenNaaS-CNSMO core service layout

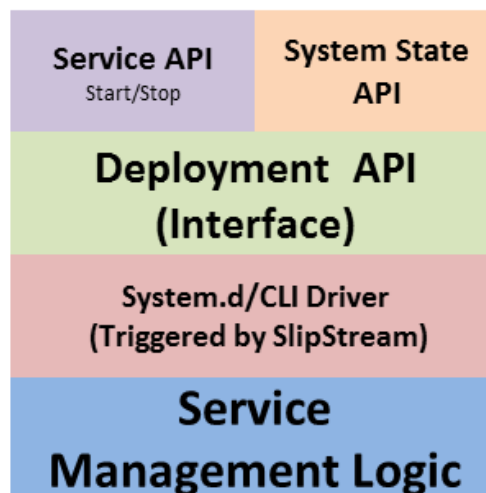


Figure 4 – Generic structure of an OpenNaaS-CNSMO service agent layout

3.3 OpenNaaS-CNSMO service modelling

In order to enable a Network service using the CNSMO framework, first, the system has to be installed and brought to “*ready state*” in order to properly manage the lifecycle of the service to be deployed. The Generic Service lifecycle of an arbitrary Network service provided by CNSMO looks as in the following sequence diagram:

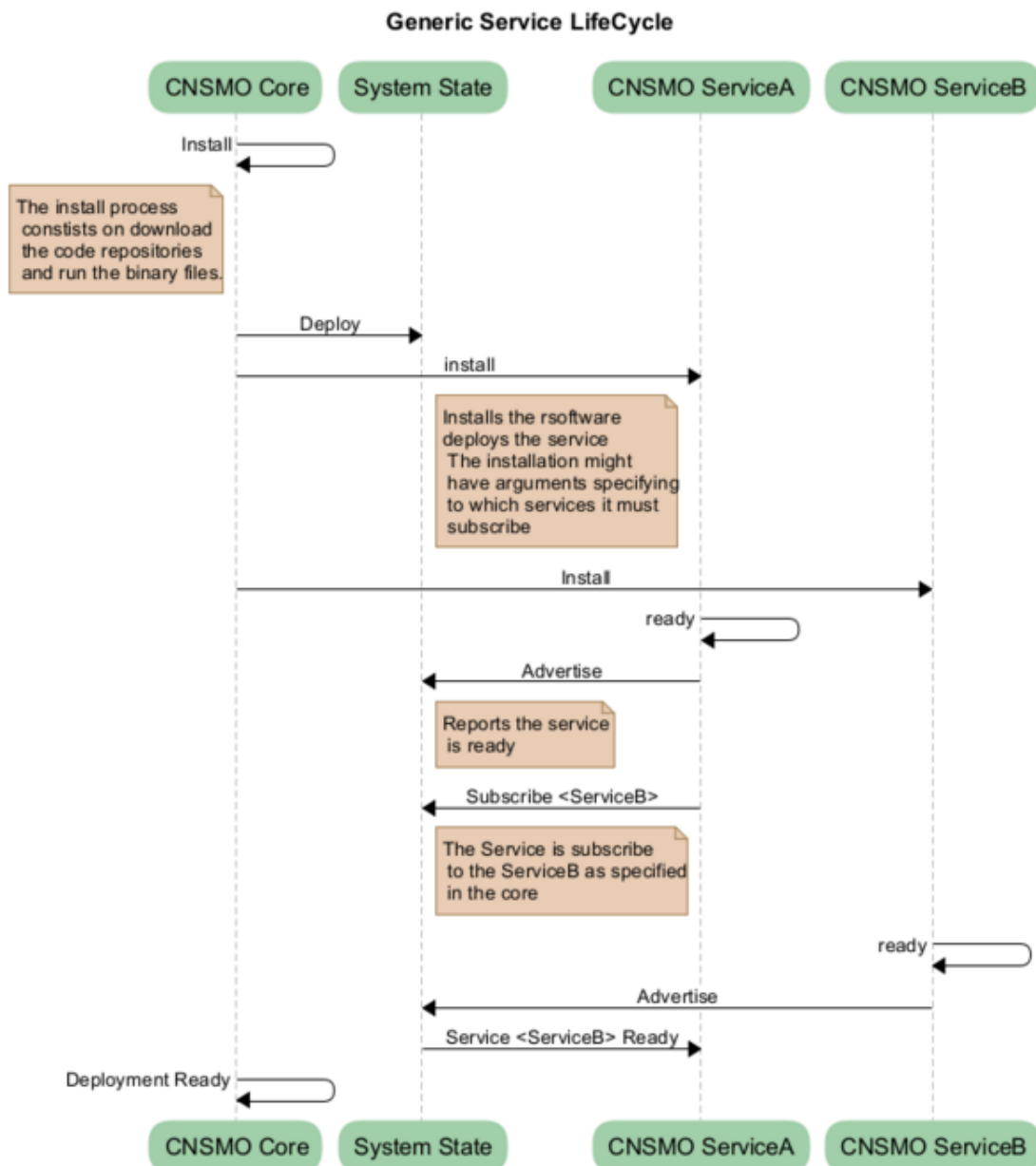


Figure 5 – OpenNaaS-CNSMO generic service lifecycle.

First, the CNSMO Core is deployed; it has an interface that enables to deploy the framework using a recipe mechanism. As a requirement of the system the first thing that the Core Agent will deploy is the System State since it is the only way to track and communicate with the other services of the system.

In the meantime, the other CNSMO services will be spawned according to what it has been specified in the recipe request. The *installation* call allows several parameters in order to specify configuration of the services, for example, to which other services it must be subscribed in order to work properly.

Once the *installation* call is triggered, the services download and configure their dependencies and announce themselves to the System State by means of an *advertise* call when they are ready. Then, the System State responds to any subscribed service if needed.

Finally when all the services have been started, the Core Service marks the process as *ready*.

3.4 CNSMO interaction with SlipStream

As it has been pointed, one of the key parts of the work consist on making the network services available to the ASPs and in CYCLONE, this is achieved by means of SlipStream platform. Therefore, it is fundamental to ensure a proper integration of CNSMO with the SlipStream logic.

The CNSMO module is deployed by means of SlipStream as part of the application deployment. The module may take advantage of the deployment information offered by SlipStream via a client tool to coordinate the deployment workflow and ensure networking services are triggered in the appropriated phase.

From the SlipStream application developer perspective, the CNSMO is composed by two types of components: The **CNSMO core** and the **CNSMO agents** running networking services.

- CNSMO core is an application component in SlipStream jargon [Jargon].The CNSMO core contains a CNSMO agent running services that are fundamental to CNSMO platform itself. It is not directly related to any networking service. Instead, it runs the distributed system state and other support services CNSMO couldn't work without.
- CNSMO agents are wrappers for a specific service. The map of CNSMO Agent and Service is *one-to-one*. In order to run a CNSMO agent, runnable python scripts are provided for convenience. Each service announces accepted parameters.

CNSMO is integrated as an application component inside SlipStream, and has direct access to the orchestrator deployed by SlipStream in any run. SlipStream is able to launch CNSMO services by means of each component recipe. A single SlipStream application component (VM image with deployment recipe) may launch none or several CNSMO agents. Hence, the map of SlipStream application component and CNSMO agent is *one-to-many*.

CNSMO core component already comes with required recipe to run the core. Although services may run each in a dedicated VM or container, due to requirements imposed by the CYCLONE project, CYCLONE networking services run directly on the user-space of user defined VMs. These services have been carefully designed to have a small footprint in the VMs they run, with the goal to minimize the impact on the user space. This is achieved by means of docker containers.

3.5 OpenNaaS-CNSMO framework repositories and system data sheet

Location	https://github.com/dana-i2cat/cnsmo	
Module	Language	Technologies
CNSMO Core	Bash, Python	Jinja2, Flask, requests
CNSMO Context	Python	Jinja2, Flask, requests
System State	Python	requests, Redis-DB

Table 1 – OpenNaaS-CNSMO code repository location. Modules, languages and technologies.

3.6 OpenNaaS-CNSMO main benefits and added value

The OpenNaaS-CNSMO component has been designed to provide with the network functionalities and services requested by the use case applications, but also to be compliant mainly with SlipStream and other CYCLONE software components to ensure a suitable integration. Thus, in addition to the provisioning of services itself, CNSMO has been designed to be:

- **Lightweight:** CNSMO is a lightweight service management and orchestrator platform. Service agents are minimal software pieces that coordinate themselves by means of a tiny distributed system state.
- **Distributed:** Using the distributed system state, agents that can easily communicate with each other's, despite the fact they are local or remote. A single network service may involve several agents distributed in different computers, while they have IP connectivity between them.
- **Modular, scalable and extensible:** Based on micro-services architecture, agents are designed as atomic single-purpose units. Since the architecture is simple and light-weight multiple services can be scaled across the deployments. Moreover, due to the easy APIs and flexible management, the functionalities of the framework can be added by implementing network services without having to integrate them into the main repository.
- **OS independent services:** The docker containers used by the network services provide isolation and OS independence by creating isolated user-spaces on top of the VM. In that way, the network services developed are ensured to be agnostic to the platform and independent from the user's framework, software and environment. Moreover, since containers are lightweight frameworks they don't interfere too much with the user data, resource usage or the VMS of the application.

4 CNSMO network services specification.

In section 2 – *Network service requirements* – of deliverable D5.1 [D5.1] it was reported the network service requirements set by:

- The CYCLONE security framework.
- The Cloud application provisioning tools, i.e. SlipStream [SlipStream].
- The IaaS provisioning platforms, with a major focus on StratusLab [StratusLab] and OpenStack.
- The CYCLONE initial use cases (i.e. Bioinformatics' and Energy use cases).

Additionally, in section 4 – *evaluation of use cases* – of deliverable D3.1 [D3.1] additional details on the network service requirements including and implementation plan were provided. These lists of requirements have driven the initial network resources implementation effort that will continue during Y2 and Y3.

The OpenNaaS-CNSMO software enables a generic service definition and architecture to accommodate any network service that needs to be implemented. Thus, despite the initial effort to extend OpenNaaS with the CNSMO module, we expect to leverage on the generic service definition approach to speed up the development of later network services during the project time being. In this section it is explained the detail of the CNSMO generic service architecture definition, the service bootstrapping process and the first implemented services that are already available.

4.1 OpenNaaS-CNSMO service concept

A network service in the context of CYCLONE is a set of configurations in a user deployment (carried out by SlipStream) that provides an added value to the default functionalities of the different CSPs. For example, provide connectivity between two or more CSPs, add new firewall functionalities to the user VMs, configure load balancer, etc.

OpenNaaS-CNSMO is the software component responsible of deploying, configuring and running the CYCLONE network services in both local and remote environments.

The CNSMO services concept has been designed so that they are recursive, stateless, independent, scalable and developed under the premise that a service can potentially launch any other service developed in CNSMO.

- *Recursive* means that any CNSMO context can manage another CNSMO context as many times as needed, providing a way to chain or construct complex services. For example the VPN service has one degree of recursiveness since the VPN Orchestrator manages the CNSMO frameworks in charge of the configuration, the VPN Server and the clients.
- *Stateless* makes reference to lack of any external state other than running or not running. Simplicity is one of the CNSMO keys. A CNSMO framework does not have a complex state machine that needs to be integrated to the applications that are being managed by CNSMO. Instead it only has one

state, running. This provides the possibility to migrate the CNSMO contexts to other namespaces without having to deal with a complex recovery/migration procedure.

- *Independent* applies for the ability to be able to run as standalone module without having any external dependencies other than the python binaries. With independence, distributing the contexts becomes easier. It also leverages the ability to migrate and recover from failures in a CNSMO context, as a single context may fail but not the modules that might be on the system.
- *Scalable* means that any deployed network service scales as much as the CYCLONE application does: In case a CYCLONE application needs to scale out an application deployment by adding a number of IT resources (e.g. VMs), those additional IT resources will be configured with the networking requested network services too, avoiding the need to redeploy the network services stack again.

4.2 Bootstrapping process: Deploying and running the network services

Starting from the premise that CNSMO relies on SlipStream to deploy the network services CYCLONE cloud federation, the general bootstrapping process to run a network service is shown in Figure 6.

Step 1: In the first step, the application developer needs to specify the application profile and requirements. To this end, the user must fill in a recipe in which the detail of the required deployment (including network services) is specified.

Step 2: SlipStream deploys the requested VMs to run user's application based on the requested recipe and bootstraps CNSMO. Each time an application is deployed by SlipStream, it builds component images, runs those images in VMs and, afterwards, runs deployment recipes of each VM. The CNSMO image contains a *system.d* lifecycle service that runs CNSMO when SlipStream launches the network module.

Step 3: Once CNSMO has been launched, it offers an API so that others can interact with it. The CNSMO SlipStream application contains a deployment recipe with appropriate instructions for CNSMO to deploy the chosen networking services. The recipe is run by SlipStream inside the CNSMO VM. This recipe gathers information of the deployment from SlipStream by means of SlipStream command line [SlipStream-CLI] client, calls CNSMO API to deploy desired network services and uses the SlipStream command line client to announce to the rest of component that the networking services are set up, so SlipStream can resume their deployment (running their deployment script).

Step 4: The deployment information is used by CNSMO to determine which services must be deployed in each component. CNSMO locates the rest of the components in the deployment, finds their roles, determines which service must be deployed in each component and deploys them.

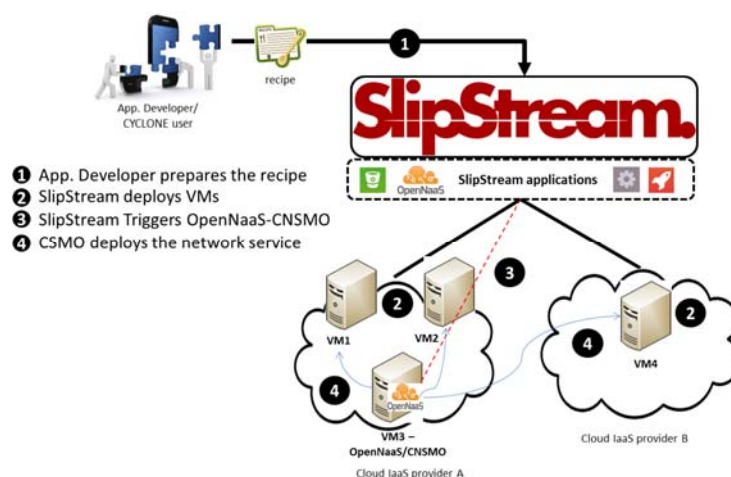


Figure 6 – OpenNaaS-CNSMO bootstrapping process to deploy networking services.

4.3 Multi-cloud VPN service

The VPN service is in charge of providing secure connectivity between all client VMs even if they are provisioned using different cloud service providers.

4.3.1 VPN service components

In order to fully deploy the VPN Service, the following components are required:

- **System State:** The system state is the distributed database that acts as a service discovery manager for all the CNSMO contexts spawned across the different elements of the system.
- **VPN Orchestrator:** Service spawned in the CNSMO Context. It interacts with the rest of the services in order to configure and run the VPN as whole. In order to be able to deploy the service, the orchestrator is subscribed to the other services (through the System State) and it will not start to deploy anything before checking that the status of the rest of the services is ready.
- **VPN Configurator:** Service spawned in the CNSMO context. This service is in charge to provide the configuration files required for the Server and the Clients and generate the required certificates and keys. As input, this service expects the generic configuration of the VPN, (VPN Server Port, IP ranges used by the Clients, etc.).
- **VPN Server:** Service spawned in the CNSMO context. This service expects to receive the certificates and configurations of the VPN Server, and provides API to launch the docker container containing the OpenVPN Server software.
- **VPN Clients:** Services Spawned in the CNSMO context. This Service expects the certificates and configurations of the VPN Clients from the Orchestrator and provides API to manage the Docker APP containing the OpenVPN client software.

Each of previous components are complete services whose internal structure is the one described in Figure 4, except for the System State which does not require from the "System State API". In the context of CYCLONE, previous components are deployed by SlipStream as explained in section 4.2 in a set up consisting of the following VMs:

- **Server VM:** A VM that contains the VPN server CNSMO context, the VPN configurator CNSMO context and the VPN Orchestrator CNSMO context. The role of this VM is to establish the link between the entire client VMs of the target VPN.
- **Client VMs:** The client VMs the VMs that are going to be part of the VPN, the role of these VMs is just to run the user applications and have a VPN Client CNSMO context. The client VMs are the VMs deployed to run CYCLONE use cases' applications.

Therefore, for the concrete VPN service case, the deployment of a CYCLONE application which demands " n " VMs, will need of an additional VM to be able to include this network service, thus, the total number of VMs will be " $n+1$ ". Figure 7 shows an example deployment for an application which demands 4 VMs. The 4 client VMs are the native CYCLONE application VMs and the 5th one is devoted to run the VPN server context.

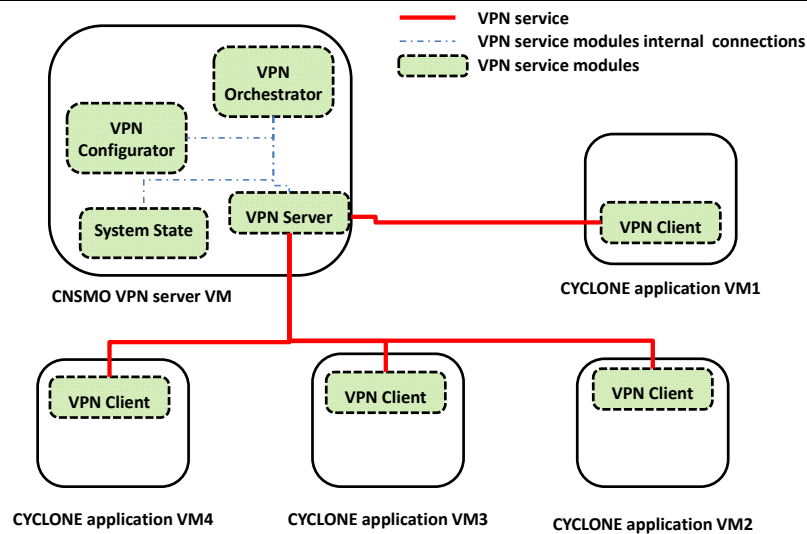


Figure 7 – VPN service modules example deployment.

4.3.2 VPN implementation – Chosen technologies

While considering the implementation of the VPN service, the following technologies are involved:

- System State: Redis DB with the PUBSUB [PUBSUB] mechanism.
- VPN: OpenVPN [OpenVPN]
- Server and Clients: *Dockerised* Apps running the VPN Clients/Server
- CNSMO context: CNSMO i2CAT repository (python mainly).
- CNSMO APPtoAPP Communication: Exposed REST services as first prototype, moving to TCP streams serialized with high performance schemas (Protobuf, or others).

4.3.3 VPN service workflow

The following Sequence Diagram summarizes the VPN Deployment Workflow. For the sequence diagram a two client and one server VPN is chosen for simplicity, although the VPN could be comprised by any number of clients. Every Service Context described is started at an arbitrary time, and there is not specific order on how the interactions run. The Sequence diagram is only presented in a “sequential” way for the sake of the simplicity and easy understanding of how the system works. Figure 8 represents the VPN service workflow.

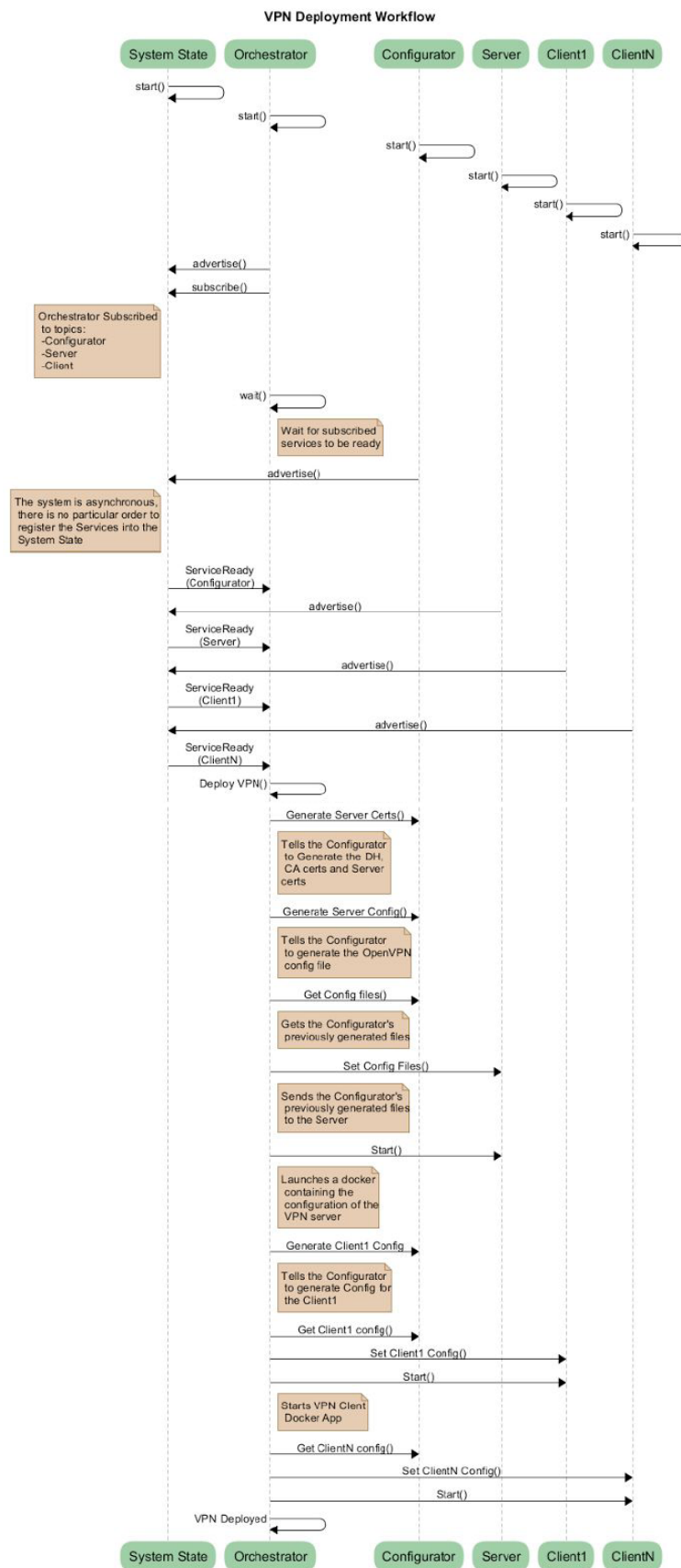


Figure 8 – VPN service workflow.

4.3.4 VPN service PoC and Demonstrator.

4.3.4.1 VPN PoC specification

4.3.4.1.1 Scenario

The scenario in this proof of concept needs to prove *dockerizing* a VPN server and every single client of the VPN. Therefore, the docker container must share the network with the Host machine to allow the VPN routing to work properly.

4.3.4.1.2 Solution

First, we had to create the CA and Diffie-Hellman files [Diffie-Hellman] using OpenVPN. Afterwards, using OpenVPN too, we generated the certificates & keys for the server and the client. These files will be copied into the docker containers to launch and connect to the VPN. Anyway, this is not enough. We also had to create a bash script to share the networking Linux device files with the containers.

4.3.4.1.3 Procedure

Once we had all the files ready, we launched the VPN server container into a remote VM which will be listening for new clients and then launched the VPN client container. The server automatically added the client to the VPN. Finally, we were able to ping the VPN client's private IP address from the remote VM and vice versa.

4.3.4.2 VPN Demonstrator

The VPN service demonstration consists of a deployment recipe supported by SlipStream and uploaded in the SlipStream instance provided by SixSq, namely Nuvla [Nuvla].

When launched, this recipe deploys a set of VMs, each in a cloud of user choice, and configures the VPN to enable isolated IP traffic between them. SlipStream is able to deploy in multiple commercial and private clouds. Each user configures a set of accounts and SlipStream will prompt the user which one to use when deploying an application. In Appendix A, the complete “*how to run*” workflow of the VPN demonstrator has been detailed.

4.3.5 OpenNaaS-CNSMO VPN network service code repository

Repository		Description
https://github.com/dana-i2cat/cnsmo-net-services		Docker files that execute the service
https://github.com/dana-i2cat/cnsmo/tree/master/src/main/python/net/i2cat/cnsmoservices		CNSMO executables to manage the docker containers.
Module	Language	Technologies
VPN	Bash	Docker, OpenVPN

Table 2 – VPN service repository and technologies

4.4 Firewall service

The Firewall service is in charge of establishing a single FW instance per VM without using the security groups (e.g. a kind of top level firewall provided by the CSP and which cannot be bypassed). The motivation of having this type of service is to reduce the number of security groups used by the different SlipStream

applications. On some clouds, the number of security groups available per VPC is lower than in others and what it is more relevant, very often the number of security groups that a user can create by CSP is limited by a small number. This difference makes the “Abstraction” of the security groups difficult for SlipStream since there is no generic way to specify the different security needs of the users in a flexible way.

4.4.1 Firewall service components

- Firewall agent (FW agent): CNSMO context running the firewall on every application VM that requires this service.

4.4.2 Firewall implementation – Chosen technologies

While considering the implementation of the Firewall service, the focus is on the technologies to build the FW agent:

- CNSMO context: CNSMO i2CAT repository (python mainly).
- Docker using IP tables
- API to configure the FW rule at deployment time

4.4.3 Firewall service workflow

Figure 9 shows the Firewall service deployment workflow.

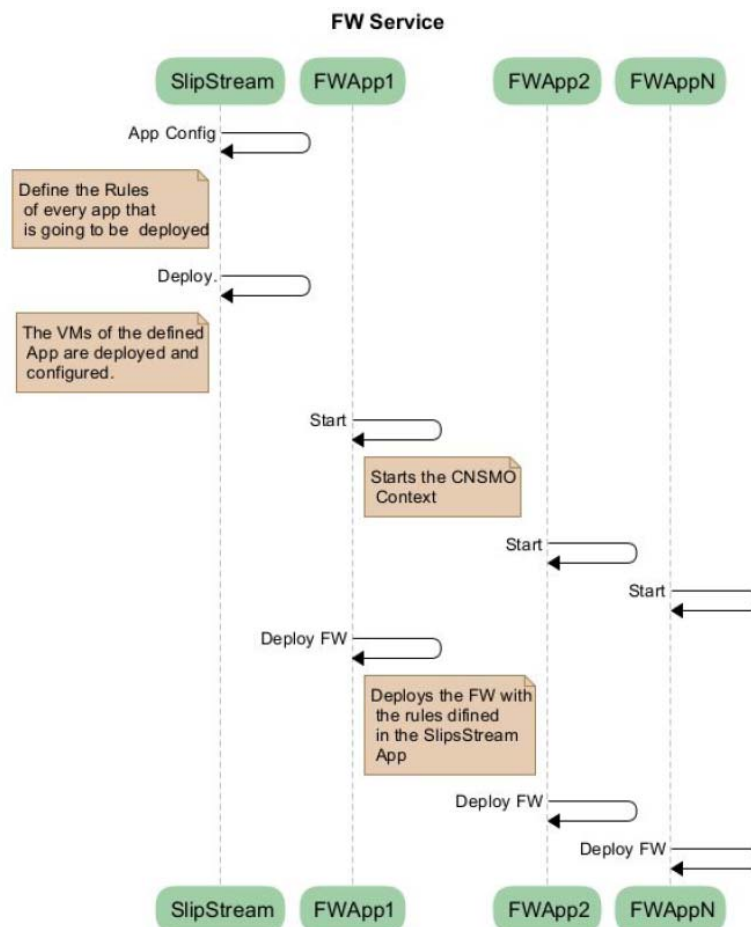


Figure 9 – Firewall service workflow.

4.4.4 Firewall service PoC and Demonstrator

4.4.4.1 Firewall PoC specification

4.4.4.1.1 Scenario

In this scenario the host needs a manageable firewalling system according to the Security Groups rules of the Cloud Management Platform. This system must be able to add and delete the firewalling rules within the host on demand.

4.4.4.1.2 Solution

Aiming for the most isolated, light and clean solution we decided to create a custom Docker container to manage the firewall through IPTables. This custom container sets the Linux Alpine image as base for subsequent instructions which prepare the docker to run as an executable script. Once the container has been built, it's ready to run an inner python script as an IPTable wrapper to manage the host's firewall rules.

4.4.4.1.3 Procedure

In order to prove our solution we first need to build the docker image. Once the build has been successfully done we can run the docker with parameters `add`, `del`, `in`, `out` `protocol`, `destination-port` `ip address` `drop` and `acpt`. When the container finishes running, it is automatically cleaned up and removed and we can see it modified the IPTables rules of the host. See example below:

- `docker build -t fw-docker`
- `docker run -t --rm --net=host --privileged fw-docker add in tcp 8080 1.1.1.1/27 drop`
- `docker run -t --rm --net=host --privileged fw-docker del in tcp 8080 1.1.1.1/27 drop`

Running the previous commands adds an IPTable rule into the host and deletes it just after.

4.4.4.2 Firewall Demonstrator

As in previous VPN service case, the demonstrator consists in an application deployment recipe supported by the SlipStream Nuvla instance. In this scenario the host needs a manageable firewalling system according to the Security Groups rules of the Cloud Management Platform. This system must be able to add and delete the Security Groups rules within the host on demand. In Appendix B, the complete scenario and "how to run" workflow of the Firewall demonstrator has been detailed.

The FW demonstrator consists of deploying CNSMO and 3 VMs, one client, and two servers. The client will be able to generate traffic on the ports 8080 and 8081. The servers will have a listener on both ports.

In order to prove the functionality of the FW, both servers will have installed the FW service. On the Server1, the firewall will be configured to drop all traffic from TCP port 8081; on the other hand, the Server2 will have configured the firewall to drop all traffic from port 8080.

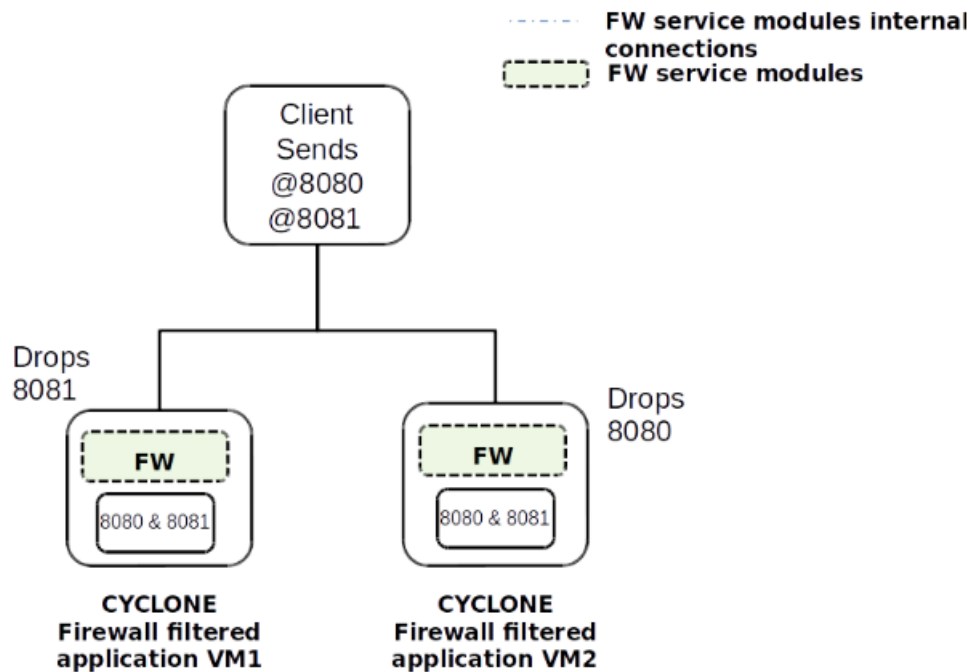


Figure 10 – Firewall service deployment layout

In this application, the application developer (Clara in [SlipPersona]) has created 4 different components, CNSMO server and the 3 mentioned above, and has already defined a default set of firewalling rules in each one.

Client does not filter traffic; hence, its firewall contains no rule. However, one server filters traffic on port 8080, while the other one filters traffic on port 8081. This is done by adding the following rules in the value of the input parameter called `net.i2cat.cnsmo.service.fw.rules` that has been defined for such purpose:

- To filter 8081: `{"direction": "in", "protocol": "tcp", "dst_port": "8081", "ip_range": "0.0.0.0/0", "action": "drop"}`
- To filter 8080: `{"direction": "in", "protocol": "tcp", "dst_port": "8080", "ip_range": "0.0.0.0/0", "action": "drop"}`.

4.4.5 CNSMO Firewall network service code repository

Repository		Description
https://github.com/dana-i2cat/cnsmo-net-services		Docker files that execute the service
https://github.com/dana-i2cat/cnsmo/tree/master/src/main/python/net/i2cat/cnsmoservices		CNSMO executables to manage the docker containers.
Module	Language	Technologies
Firewall	Bash, python	Docker, IPTables, Python Interface for IPTables

Table 3 – Firewall service repository and technologies

4.5 Load balancer service

4.5.1 Load Balancer service components

In order to provide this service, the following components are needed:

- **System State:** The system state is the distributed database that acts as a service discovery manager for all the CNSMO contexts spawned across the different elements of the system. As mentioned on Section , this service is required by the CNSMO framework
- **LB Configurator:** Component spawned as CNSMO context that in charge of making the LB configuration files required by the HA proxy. In order to make this service generic and extensible even when the system is running, the LB configurator is already deployed.
- **LB Orchestrator:** Component spawned as CNSMO service that is in charge of register all the services to be load-balanced and notify the configurator to update the HA Proxy configuration. It also tracks the state of the agents and the Masters
- **LB Agent:** Service spawned as CNSMO context that is in charge to inform the orchestrator that is a service to be used as backed of the load balancer. The service lies on the user VMs that are going to be used as scaled or as HA service.
- **LB Master:** Service spawned as CNSMO context that is in charge of managing the LB software contained in a docker container. It receives the configuration via the LB Orchestrator (and previously generated by the LB Configurator). This service will be the entry point of the “scaled” services.

Each of previous components are complete services whose internal structure is the one described in Figure 4, except for the System State which does not require from the “System State API”.

The way these services are used in CYCLONE as part of SlipStream integration is the following:

- **LB VM:** This VM will contain the Base CNSMO services: The System State, the Orchestrator and the LB Master. This VM will be the entry point of the application that is going to be scaled or used with HA.
- **LB Agent:** For every Application that the user wants to load-balance or scale, there will be an LB Agent, these VMs will be responsible of advertise themselves to the LB orchestrator in order to be able to the Master redirect and distribute the traffic.

Therefore, for the concrete LB service case, the deployment of a CYCLONE application which demands “n” VMs, will need of an additional VM to be able to include this network service, thus, the total number of VMs will be “n+1”. Figure 11 shows an example deployment for an application which demands 3 VMs. The 3 client VMs are the native CYCLONE application VMs and the 4th one is devoted to run the LB server context. There is an extra VM that is the one requesting or generating data for the LB. The last application might be part of the application or outside depending of the application.

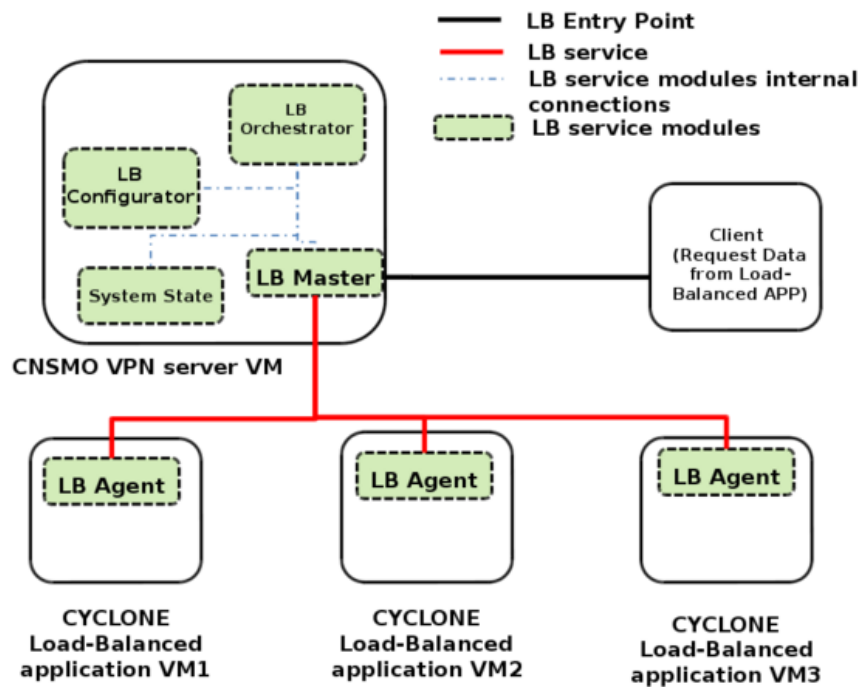


Figure 11 – Load Balancer service deployment example.

4.5.2 Load Balancer implementation – Chosen technologies

For the implementation of the Load Balancer service, the following technologies have been utilized:

- Load balancer: HAproxy using TCP mode
- Server: Dockerized HAproxy running the Load Balancer server
- Load Balancer scheduling: Configurable (Round Robing, FIFO, etc).

4.5.3 Load balancer service workflow

The following Sequence Diagram summarizes the LB Deployment Workflow. For the sequence diagram a two agents and one LB is chosen for simplicity, although the LB could be comprised by any number of agents. Every Service Context described is started at an arbitrary time, and there is not specific order on how the interactions run. The Sequence diagram is only presented in a “sequential” way for the sake of the simplicity and easy understanding of how the system works. Figure 12 represents the LB service workflow.

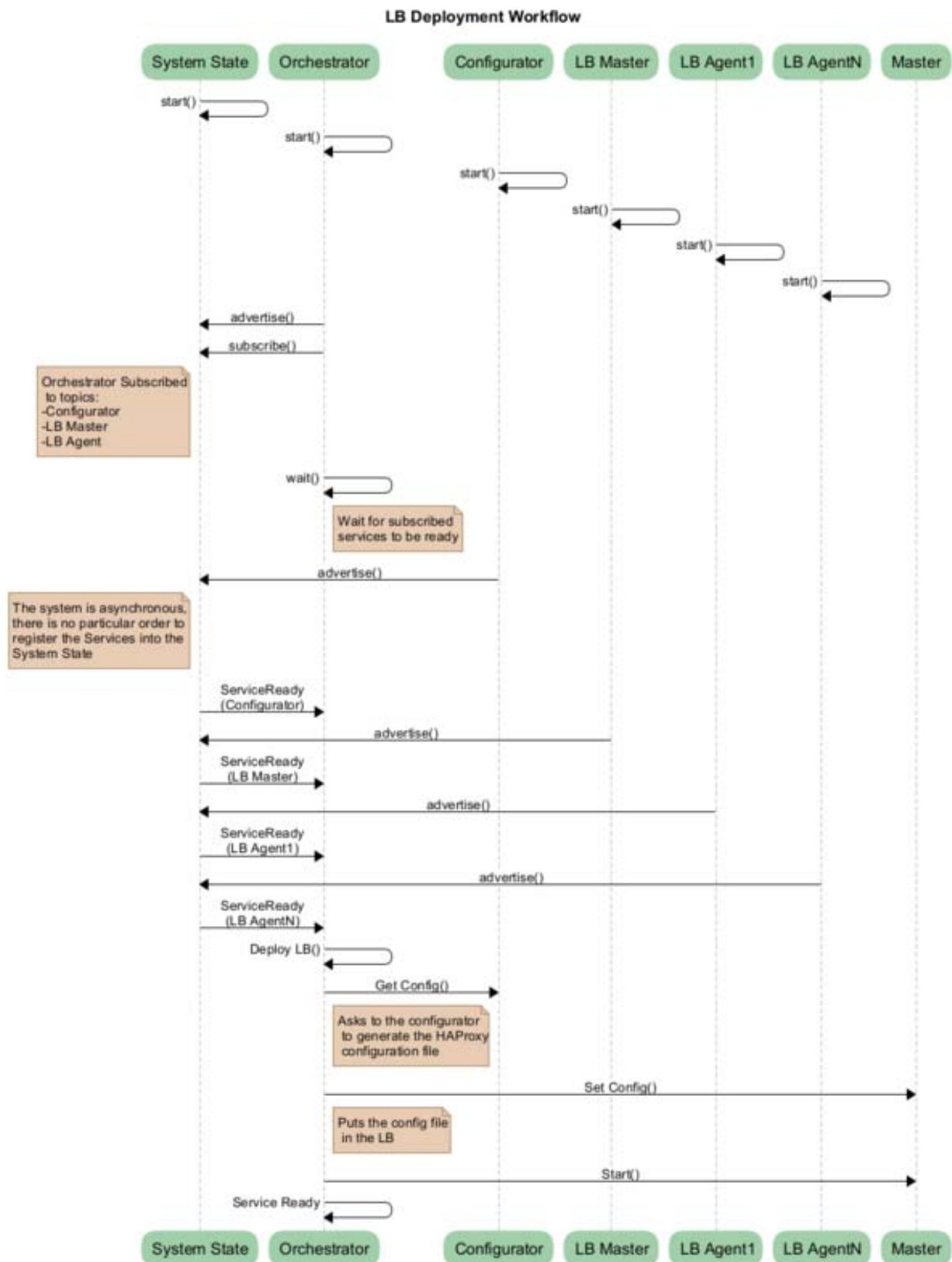


Figure 12 – Load Balancer deployment workflow.

4.5.4 Load balancer service PoC and Demonstration

4.5.4.1 Load Balancer PoC specification

4.5.4.1.1 Scenario

We need to prove TCP Load Balancing between different backend servers through a Docker container. The container must be aware of the backend servers when it's run and balance the connections with the Round Robin scheduling mode. We need to prove docker can easily accomplish this job.

4.5.4.1.2 Solution

HAProxy provides a reliable and high performance solution and it's well known it works but we need it to be encapsulated in a container. There are already some *HAProxy* docker images but they can be improved by removing some junk in the *Dockerfile* and that's what we did.

For different configurations, the *Dockerfile* and the *HAProxy* configuration file need to be modified. The configuration file tells *HAProxy* where and how to load balance the incoming TCP connections.

4.5.4.1.3 Procedure

To prove the solution, we deployed three virtual machines in Slipstream: two serving as wordpress servers with some differences and one to balance the network traffic between them. When everything was ready, we simply opened an Internet browser and connect to the Load Balancer's public IP multiple times and notice each consecutive connection is redirected to a different server.

4.5.4.2 Load Balancer Demonstrator

By the time the document has been released, the WP5 team is working on the demonstrator in a similar way as the one explained for previous VPN and Firewall services. The demonstration results will be reported in later WP5 reports and publications.

4.5.5 CNSMO Load Balancer network service code repository

Repository		Description
https://github.com/dana-i2cat/cnsmo-net-services		Docker files that execute the service
https://github.com/dana-i2cat/cnsmo/tree/master/src/main/python/net/i2cat/cnsmoservices		CNSMO executables to manage the docker containers.
Module	Language	Technologies
Load Balancer	Bash, python	Docker, LB scheduling algorithms.

Table 4 – Load Balancer service repository and technologies

5 Network service management and abstraction

Network management and service abstraction are two important features while introducing network services into cloud federations. Therefore it has been considered suitable to dedicate a section to these two items and clarify their scope according to the network services design and implementation in the context of CYCLONE.

5.1 Network management

The initial idea behind “network management” is aligned with the possibility to control by means of OpenNaaS, specific IaaS providers’ network resources (firewalls, switches, ToRs, Routers, etc.) and be able to delegate certain management functions of such resources to CYCLONE users (i.e. application developers). As it has been explained in section 2.1, to the date, OpenNaaS is not able to gain access and control over such network resources and the fact that this is a common situation in today’s hybrid cloud scenarios.

Therefore, instead of enabling CYCLONE users with management capabilities over the network itself, specific management options over the network services have been implemented. Currently, it is possible for CYCLONE users to select which of the available network services want to be included while deploying the application(s) and define concrete specific fields relevant to each particular service, e.g. select which ports should remain opened while deploying a firewall service, or select the preferred IaaS to allocate the network service components in the case of the VPN service. In the future, it is foreseen to also include some additional management options as the possibility to concatenate network services, or to select several service flavours while modifying the service logic that will be reported in later deliverables. In the case one or more IaaS resources could be made available, the WP5 team will also demonstrate the possibility to delegate concrete control and management functionalities of such network resources.

5.2 Network service abstraction

As in previous section 0, the initial idea behind provisioning abstracted network services was mostly related to the possibility for OpenNaaS-CNSMO to sit on top of the IaaS network resources. The abstraction model would rely on virtualization to decouple physical topology and vendor-specific details from their control and management features to be offered to the CYCLONE users, hiding the complexity of network resources configuration. This model pretends to be flexible enough to procure different designs and orientations, but fixed enough so common tools can be built and reused across plugins.

With the new approach the network abstraction service concept has been also re-defined: OpenNaaS-CNSMO is integrated with SlipStream. SlipStream, simplifies and abstracts the underlying IaaS infrastructure, in such a way that CYCLONE users do not need to deal with the complexity of requesting and managing resources of one or several IaaS providers, but already provides the VMs in a transparent way. OpenNaaS-CNSMO relies on this abstraction of resources already provided by SlipStream to deploy the network services also ensuring the abstraction of the network services’ elements that such services may comprise.

6 Summary of network accomplished requirements

So far, with the implementation of the explained network services, a number of requirements are covered and the progress of the WP implementation and integration efforts is moving forward to ensure a proper adoption of these services in CYCLONE federation as well as to introduce new services and network features suitable to the different CYCLONE stakeholders.

6.1 Achieved network features and services

Table 5 – taken from D5.1 section 2.6 – shows the complete list of foreseen network requirements imposed by the use cases, the CSPs and the other CYCLONE software components:

- In green colour, it can be identified the network requirements which have been already addressed.
- In yellow, it can be identified the ones which are “*in-progress*”.
- The items highlighted in blue will be closer analysed. It will be evaluated the possibility to develop an OpenNaaS-CNSMO connector to integrate with CYCLONE CSP platforms (OpenStack, AWS, etc.) in order to provision these network features too.
- Finally, the ones in white have been scheduled for a later stage.

Origin of the requirement	Item	Requirement	Priority level	Due (if planned)
CYCLONE framework component (SlipStream)	SlipStream Cloud Application management description model	Integrate with SlipStream application description model (networking features)	High	M14-M18
CYCLONE framework component (SlipStream)	Standard application description model adoption: CIMI	Integrate with CIMI model for networking purposes.	High	
CYCLONE framework component (SlipStream)	Firewalling mechanism	Enable a large quota of per-User security groups	High	M14-M18
		Enable a large quota of per-application security groups	High-Medium	M14-M18
		Enable automatic firewalling capability	High	M14-M18
CYCLONE framework component (SlipStream)	VPN configuration and management	Enable automatic VPN configuration options	High-Medium	M14-M18

CYCLONE framework component (SlipStream)	Load balancing configuration and management	Enable load balancing configuration options	Medium	M14-M18
CYCLONE framework component (SlipStream)	Guarantee network QoS level	Guarantee QOS SLAs in multi-cloud application deployments	High	Depends on the possibility to access network resources
CYCLONE framework component (SlipStream)	DHCP configuration and management	Enable automatic DHCP configuration options	Medium-Low	
CYCLONE framework component (SlipStream)	DNS configuration and management	Enable automatic DNS configuration options	Medium-Low	
CYCLONE framework component (TRESOR)	Network logging	OpenNaaS network service management tool should be able to log its messages to the ELK logging stack	High	M18-M24
CYCLONE framework component (CSPs)	Network virtualization	Virtualize network resources participating of the CYCLONE federation	High	
CYCLONE framework component (CSPs)	Network abstraction	Abstract network resources participating of the CYCLONE federation to reduce network complexity	High	
CYCLONE framework component (CSPs)	Network isolation	Ensure applications' traffic isolation while sharing same networking resources	High	M14-M18
CYCLONE framework component (CSPs)	Multi-tenancy support	Enable several applications to share the same networking resources	High	M14-M18
CYCLONE framework component (CSPs)	Dynamic network discovery	Auto-discovery of networking resources entering/leaving the CYCLONE network infrastructure	Medium	Depends on the possibility to access network resources
CYCLONE framework component (CSPs)	Management delegation	Network management and configuration options should be enabled to different CYCLONE stakeholders (mainly CSPs and cloud application managers)	High	Depends on the possibility to access network resources
CYCLONE framework component (CSPs)	SDN/OF support	Support of OF based network resources	High	Depends on the possibility

				to access network resources
CYCLONE framework component (CSPs)	Elasticity	Enable network elasticity	Medium	M14-M18
CYCLONE framework component (CSPs)	Inter DC connectivity	Provide with network inter-DC service management	Medium	
CYCLONE Use cases (Bioinformatics and Energy)	End-to-end secure data management	The connection between the cloud infrastructure and the user computer need to be secured	High	M14-M18
CYCLONE Use cases (Bioinformatics)	VM isolated network	The VMs of a user must be deployed in a dedicated and isolated network	High	M14-M18
CYCLONE Use cases (Bioinformatics)	VPN connectivity services	Provide access to all VMs of an user in a simple manner	High-Medium	M14-M18
CYCLONE Use cases (Bioinformatics)	Multi-clouds distribution of community reference datasets	Provide replication mechanisms for public reference datasets	High-Medium	M14-M18
CYCLONE Use cases (Bioinformatics and Energy)	Dynamic network resource allocation	Provide Dynamic network resource allocation according to the steps of the application workflow	High	M14-M18
CYCLONE Use cases (Bioinformatics and Energy)	Multi-clouds distribution of user data	Distribute the user data in several cloud infrastructures	High-Medium	M14-M18
CYCLONE Use cases (Bioinformatics and Energy)	WAN high bandwidth links	Provide dynamic allocation of high-bandwidth links between the data producers and the cloud infrastructures	High	M14-M18
CYCLONE Use cases (Bioinformatics and Energy)	Guaranteed network performances (QoS)	Guarantee the network performances for some applications features (remote display, real-time)	High-Medium	M14-M18

Table 5 – Summary of network requirements imposed by the UCs, CSPs and other CYCLONE software components.

7 Monitoring of other cloud networking solutions: BEACON and SWITCH projects.

CYCLONE project is part of the “*Inter-cloud challenges, expectations and issues*” cluster together with other FP7 and H2020 projects. The BEACON [BEACON] and SWITCH [SWITCH] projects are also taking part and some of the activities that are being driven there are of the interest of CYCLONE WP5, since they are directly related to cloud networking.

During Y2, CYCLONE has been monitoring the activity carried out by these 2 projects and the approaches taken to provide with network solutions to cloud federation environments. Additionally, during the last NetFutures edition [NetFutures] CYCLONE and BEACON had a F2F meeting to explain the scope of the cloud networking developments and identify potential liaisons. The most relevant outcomes from that session were:

- BEACON approach consists on an OVN-based Overlay controller [OVN] in order to support virtual network abstractions and it is being integrated with OpenStack and OpenNebula [OpenNebula] CSPs in order to provide cloud federated network services.
- BEACON federates the network and the cloud all at once.
- Geolocation constrains are taken into account. It is possible to specify in which specific location the resources are needed.

In CYCLONE we are using a different approach utilizing OpenNaaS-CNSMO as an application in SlipStream to bring the network services in cloud federations in a transparent manner and independently on the underlying CSP managing the IaaS providers’ facilities. Nevertheless, the OVN approach to provision virtual networks inside the clouds looks promising, so that we will follow up in contact to gain a better view on BEACON results. The organization of specific cloud networking workshops is also a possibility.

8 Next Steps

This deliverable reports on the updates of WP5 progresses, the detail of the justification and implementation of the OpenNaaS-CNSMO component and the need to integrate with SlipStream to provide with network services in CYCLONE cloud federation. It has been also presented the service model definition and it has been presented some network services that are already implemented, integrated with SlipStream and demonstrated.

The next steps planned in the context of WP5 include:

For the short term: (M16-M20)

- Adoption of the network services by the bioinformatics and Energy use cases by including networking options in the CYCLONE users' recipes.
- Implementation of the planned requirements as explained in Table 5.
- Cross activities with WP4 to integrate the network service logs as part of the overall CYCLONE logging system.
- Dissemination of one or more of the implemented network services in events and conferences.

For the mid and long term (M20-M36) the following activities have been already foreseen:

- Implementation of the remaining specified requirements. The concrete requirements that strictly depend on the possibility to control and manage network resources will be reconsidered and addressed if possible.
- Implementation of additional requirements that may impose the new CYCLONE use cases that are currently being identified under WP3 umbrella.
- Implementation of connectors and mechanisms to integrate or monitor specific network features exposed by CSPs' APIs, focussing on OpenStack and other public (Exoscale, AWS, etc.) solutions.
- Integrate the whole cloud networking solution as part of the CYCLONE cloud federation while deploying it over the testbed set up by WP7.

Appendix A – “How to run” the VPN service demonstrator

How to run

In this appendix it is detailed the complete sequence of steps to un the VPN service from the Nuvla instance:

1. Log-in to Nuvla site [Nuvla Site].
2. In your profile page, check there is at least one active CSP account, at least 2 for a multi-cloud demonstration.
3. In your profile page, check you have an SSH key configured, it will be used afterwards to access deployed VMs.
4. Select the demonstrator application which can be found [here](#) (it only works if previous steps have been done).
5. Press “Deploy” button. A deployment form appears.
6. In “Cloud” input dropdown, select “Specify for each node”
7. Specify a different cloud for each node. The form should look similar to the one below.

The screenshot shows the Nuvla web interface with the 'Deploy Application' modal open for the 'VPN-app' (Version: 3391). The modal is divided into 'Global parameters' and 'Component parameters'.

Global parameters:

- Scalable deployment (beta): ☐
- Tolerate deployment failures: ☐
- SSH access: ☒
- Cloud: Specify for each node (dropdown)
- Keep running after deployment: Always * (dropdown)
- Tags: (empty text field)

Component parameters:

- VPN_client:** Multiplicity: 2, Cloud: cyclone-tr1
- VPN_server:** Multiplicity: 1, Cloud: cyclone-tr2 *

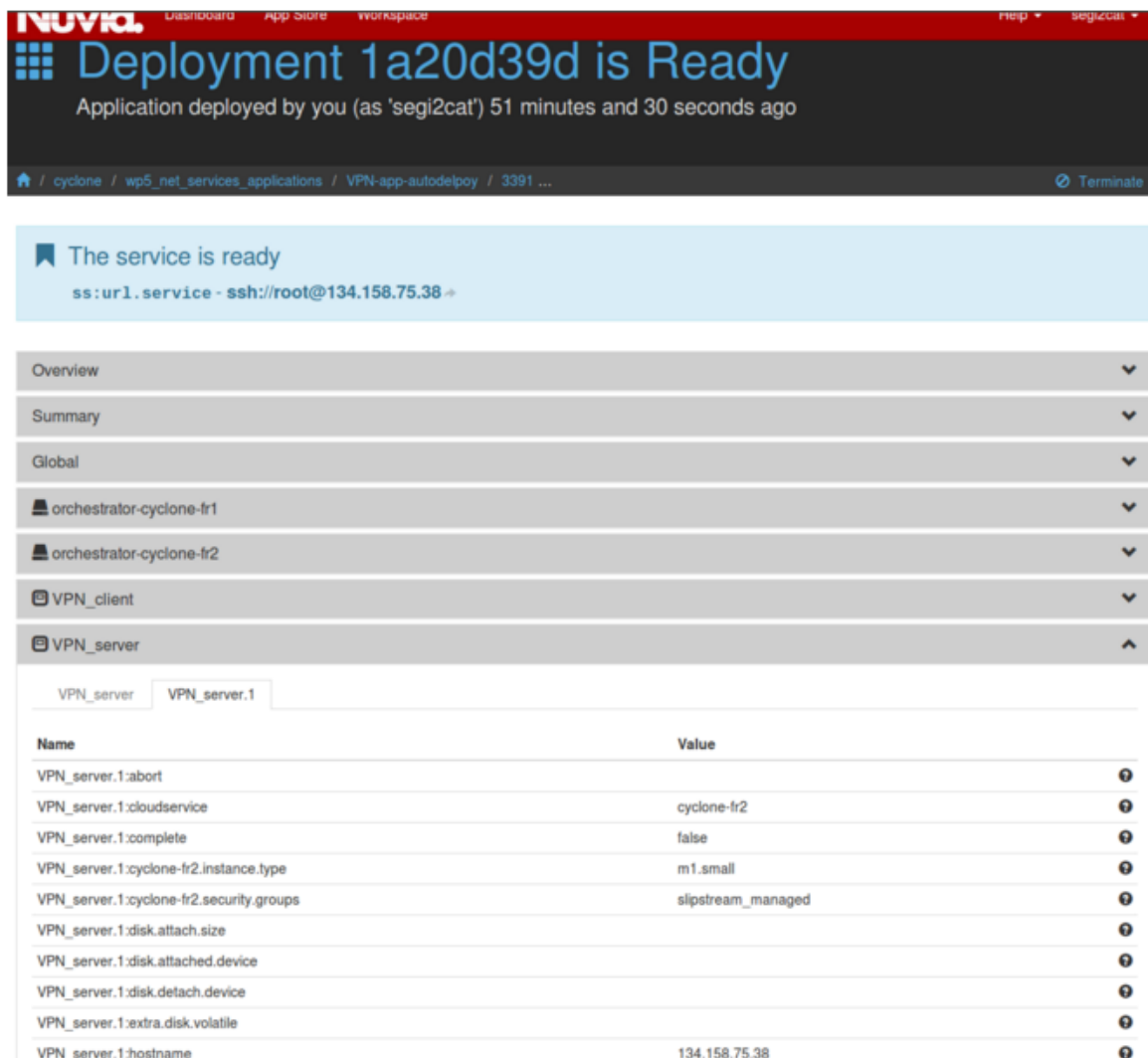
Choose the cloud where each application component will be deployed, how many of each (parameter *Multiplicity*) and parameters for each node.

Buttons: Cancel, Deploy Application

Footer: Copyright © 2016 SixSq, All rights reserved. | cyclone made software | Terms and Conditions | Powered by SlipStream @ v3.1 | Open source under Apache 2.0 License.

Figure 13 – VPN application deployment screenshot.

8. Press the “*Deploy Application*” button. This starts the deployment process which takes up to 15 minutes.
9. At this point SlipStream orchestrates the application deployment. It uses each configured Cloud Service Provider Platform to load the recipe VM images, boot them and launch the recipe scripts. These scripts install required software and launch the VPN deployment, which follows the workflow described above in this document.
10. Wait for the deployment process (“*run*” in SlipStream terminology) to finish. SlipStream announces with the “*Ready*” state.
11. Visit the run page to inspect what VMs compose them, and what is their IP address. The run page can be accessed from the dashboard, by clicking in the ID of a specific run. Figure 14 shows a deployment ready screenshot. More specifically, it shows the details of the VPN server component of the service (infrastructure where is has been deployed, IP address, etc).



Name	Value
VPN_server.1:abort	
VPN_server.1:cloudservice	cyclone-fr2
VPN_server.1:complete	false
VPN_server.1:cyclone-fr2.instance.type	m1.small
VPN_server.1:cyclone-fr2.security.groups	slipstream_managed
VPN_server.1:disk.attach.size	
VPN_server.1:disk.attached.device	
VPN_server.1:disk.detach.device	
VPN_server.1:extra.disk.volatility	
VPN_server.1:hostname	134.158.75.38

Figure 14 – VPN service deployment in “ready” state screenshot. Detail of the VPN server.

Check the VPN service establishment

In order to demonstrate that the VPN is working, several checks can be done:

- Connect through SSH to each client VM. Launching *ifconfig* command shows a tap0 interface with the addressing of the VPN.
- Ping can be used to demonstrate connectivity between VMs.
- Tcpdump can be used to demonstrate the traffic is passing through the VPN (`tcpdump -ne -i tap0`) and over the public net but ciphered (`tcpdump -ne -i eth0 'port 1194'`).

Expected Result

The VPN demonstrator sets up a VPN environment on Slipstream consisting on the following VMs:

- Server VM: A VM that contains the VPN server CNSMO context, the VPN configurator CNSMO context and the VPN Orchestrator CNSMO context. The role of this VM is to establish the link between all the client VMs of the target VPN.
- Client VMs: The client VMs the VMs that are going to be part of the VPN, the role of these VMs is just to run the user applications and have a VPN Client CNSMO context.

Appendix B – “How to run” the Firewall service demonstrator

How to run

1. Log-in to Nuvla site [Nuvla Site]
2. In your profile page, check there is at least one active CSP account, at least 2 for a multcloud demonstration.
3. In your profile page, check you have an SSH key configured, it will be used afterwards to access deployed VMs.
4. Select the firewalling demonstrator application which can be found here [Firewall-demo].
5. Press “Deploy” button. A deployment form appears. Similar to the one shown in Figure 15:

Nuvla Dashboard Application

FW-demo

Application
Version: 3489

Summary

Application Components

Name	Default config
CNSMO_server	Component Default multiplicity Default cloud
Filtered_8080	Component Default multiplicity Default cloud Parameter map
Filtered_8081	Component

Deploy Application

Global parameters

- Scalable deployment (beta) ☐
- Tolerate deployment failures ☐
- SSH access ☒
- Cloud: cyclone-ir2 *
- Keep running after deployment: Always *
- Tags:

Component parameters

- CNSMO_server**
 - Multiplicity: 1
- Filtered_8080**
 - Multiplicity: 1
 - Input: net.12cat.cnsmo.service.fw.rules [{"direction": "in", "port": 8080, "protocol": "tcp", "dst_port": "8080"}]
- Filtered_8081**
 - Multiplicity: 1
 - Input: net.12cat.cnsmo.service.fw.rules [{"direction": "in", "port": 8080, "protocol": "tcp", "dst_port": "8080"}]
- Not_filtered**
 - Multiplicity: 1
 - Input: net.12cat.cnsmo.service.fw.rules []

Figure 15 – Firewall service configuration deployment screenshot.

6. Notice the different values for `net.i2cat.cnsmo.service.fw.rules` input parameter in each component. Although the person deploying the application is allowed to change the rules (when knowing what to do), the application developer has already configured the appropriate rules for the application to work as expected. Hence, there is no need to change the firewalling rules when deploying the demonstrator.
7. Press the “Deploy Application” button. This starts the deployment process which takes up to 15 minutes.

At this point SlipStream orchestrates the application deployment. It uses each configured Cloud Service Provider Platform to load the recipe VM images, boot them and launch the recipe scripts. These scripts install required software and launch the firewalling configuration, which follows the workflow described above in this document.

8. Wait for the deployment process (“run” in SlipStream terminology) to finish. SlipStream announces with the “Ready” state.
9. Visit the run page to inspect what VMs compose them, and what is their IP address. The run page can be accessed from the dashboard, by clicking in the ID of a specific run.

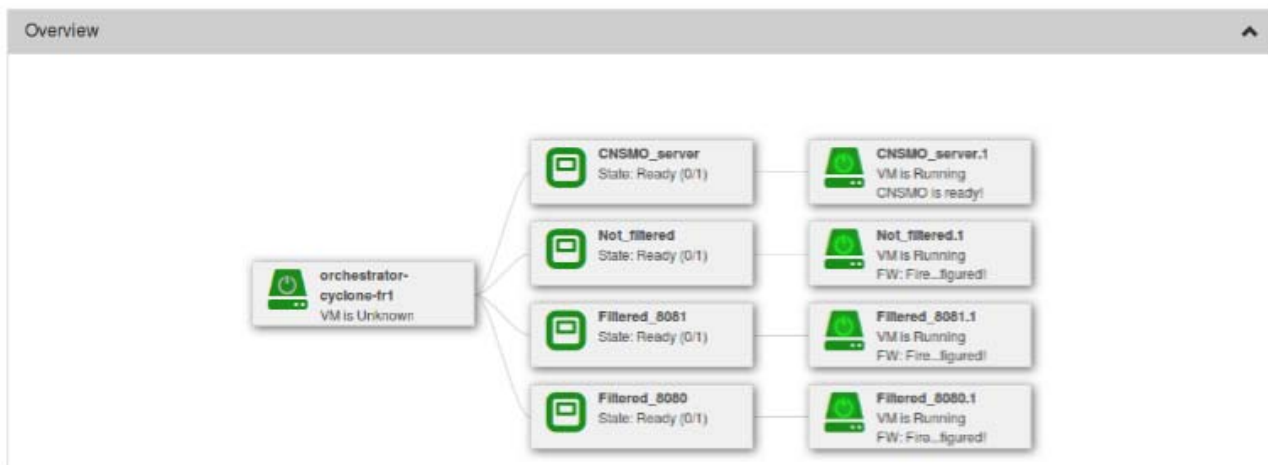


Figure 16 – Overview of the CYCLONE Firewall demonstrator deployment screenshot.

10. In order to demonstrate that the FW is configured as expected, connect through SSH to the client VM and to each server VM. Launching `iptables -L` command shows the firewalling rules applied to the local machine. Both server machines should have entries dropping traffic to port 8080 or 8081. A sample `iptables -L` output dropping 8080 (`http-alt`) follows:

```
root@onevm-179:~# iptables -L
Chain INPUT (policy ACCEPT)
target prot opt source destination tcp dpt:http-alt
DROP tcp -- anywhere anywhere tcp dpt:9091
ACCEPT tcp -- anywhere anywhere tcp dpt:9092
ACCEPT tcp -- anywhere anywhere tcp dpt:9093
ACCEPT tcp -- anywhere anywhere tcp dpt:9094
ACCEPT tcp -- anywhere anywhere tcp dpt:9095
ACCEPT tcp -- anywhere anywhere tcp dpt:ssh

Chain FORWARD (policy ACCEPT)
...
```

Figure 17 – Screenshot of the IPTABLES –L command outcome result.References

11. In order to demonstrate the FW is working as expected, that is, each server FW is dropping traffic in either 8080 or 8081 ports, one can launch the following test. In each server, run a couple of web servers one at port 8080 and the other at port 8081. You can use the following script to do so:

```
echo 'from flask import Flask
import sys
app = Flask(__name__)
' > demo_app.py
echo "@app.route('/') >> demo_app.py
echo '
def hello_world():
    return app.config.get("data")
' >> demo_app.py

echo "if __name__ == '__main__':" >> demo_app.py
echo '    port = int(sys.argv[1])
        data = str(sys.argv[2:])
        app.config["data"] = data
        app.run(host="0.0.0.0", port=port)
' >> demo_app.py

python demo_app.py 8080 "Hello from web server running at port 8080" &
disown !
python demo_app.py 8081 "Hello from web server running at port 8081" &
disown !
```

Figure 18 – Screenshot of the script to launch the test

Then, from the client VM, use curl to send requests to both servers. Each server should be filtering traffic either to port 8080 or 8081, so try both ports in each server. You can use the following command if you used the script above to launch the web servers:

```
curl http://server_ip:port/
```

In one port you'll see *'Hello from web server running at port 8081'* while in the other curl will hold waiting for a response it never gets.

Expected Result

The result of this PoC is that the servers even if they exposed two ports, they will only be able to receive data on one port. Server1 will only receive data directed to port 8080 and Server2 will only receive data directed to port 8081. This is achieved by configuring the IPTables of each VM according to the firewalling rules specified by the application developer when defining the application.

References

- [AWS] <https://aws.amazon.com/>
- [BEACON] <http://www.beacon-project.eu/>
- [D3.1] CYCLONE Deliverable D3.1: “*Evaluation of Use Cases*”. (Pending of approval). Available at: <http://www.cyclone-project.eu/assets/images/deliverables/Evaluation%20of%20Use%20Cases.pdf>
- [D5.1] CYCLONE Deliverable D5.1: “*Functional specification of the E2E Network service model*”. (Pending of approval). Available at: <http://www.cyclone-project.eu/assets/images/deliverables/Functional%20specification%20of%20the%20E2E%20Network%20service%20model.pdf>
- [Diffie-Hellman] https://wiki.openssl.org/index.php/Diffie_Hellman
- [Exoscale] <https://www.exoscale.ch/>
- [Firewall-Demo] https://nuv.la/module/cyclone/wp5_net_services_applications/FW/FW-demo
- [Jargon] SlipStream key concepts:
http://ssdocs.sixsq.com/en/latest/advanced_tutorial/key-concepts.html
- [Mesos] <http://mesos.apache.org/>
- [NetFutures] <http://netfutures2016.eu/>
- [Nuvla] <http://nuv.la/>
- [Nuvla Site] <https://nuv.la/login>
- [OpenNebula] <http://opennebula.org/>
- [OpenStack] <http://www.openstack.org/>
- [OpenVPN] <https://openvpn.net/>
- [OVN] <http://openvswitch.org/support/slides/OVN-Vancouver.pdf>
- [PUBSUB] <http://redis.io/topics/pubsub>
- [StratusLab] <http://www.stratuslab.eu/>
- [SlipStream] <http://sixsq.com/products/slipstream/>
- [SlipStream-CLI] http://ssdocs.sixsq.com/en/v2.23/advanced_tutorial/automating-slipstream.html#command-line-client
- [SlipPersona] <http://sixsq.com/personae/>
- [SWITCH] www.switchproject.eu/

<END OF DOCUMENT>