Objective FP7-ICT-2009-5-257448/D-5.3

Future Networks

Project 257448

"SAIL – Scalable and Adaptable Internet Solutions"

D-5.3
(D.D.2) Description of Implemented Prototype

Date of preparation: **2012-08-31**
Start date of Project: **2010-08-01**
Project Coordinator: **Thomas Edwall**
Ericsson AB

Revision: **1.1**
Duration: **2013-01-31**

Document Properties

Document Number:	D-5.3
Document Title:	**(D.D.2) Description of Implemented Prototype**
Document Responsible:	Hareesh Puthalath (EAB)
Document Editor:	Hareesh Puthalath (EAB)
Authors:	Azimeh Sefidcon(EAB), Bob Melander (EAB), Enrique Fernandez (EAB), Hareesh Puthalath (EAB), Victor Souza (EAB), Ayush Sharma (FHG), Sathyanarayanan Rangarajan (FHG), Dev Audsin (HP), Paul Murray (HP), Suksant Sae Lor (HP), Houssem Medhioub (IT), Marouen Mechtri (IT), Daniel Turull (KTH), Fetahi Wuhib (KTH), Paul Périé (Lyatiss), Romaric Guillier (Lyatiss), João Soares (PTIN), Jorge Carapinha (PTIN), Márcio Melo (PTIN), Pedro A. Aranda (TID), Matthias Keller (UPB)
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	1.1

Production Properties:

Reviewers:	Hannu Flinck (NSN), Bengt Ahlgren (SICS), Benoit Tremblay (EAB)

Document History:

Revision	Date	Issued by	Description
1.0	2012-08-31	Hareesh Puthalath	Final version
1.1	2012-10-04	Hareesh Puthalath	Fixed typos, set dissemination level to Public

Disclaimer:

Abstract:

This document describes the implementation of the Cloud Network (CloNe) prototyping. Cloud networking in the context of SAIL focuses on the integrated provisioning of network resources connected to data centre resources on-demand. As such, existing data centre-based Infrastructure as a Service (IaaS) can be interconnected through the wide area network using a network service model. At the core of the solution resides a distributed control plane which binds resources together and thus creates full virtual infrastructures. A new network service interface that fits the needs of cloud computing has been designed and implemented. This document presents the tools utilized in prototyping, extensions performed to existing tools, components that were implemented from scratch, as well as how those components were integrated into a system.

Keywords:

IaaS, Cloud Networking, Prototyping, MPLS, OpenFlow, VPN, OpenStack, OpenNebula, Infrastructure Service Interface, Distributed Control Plane, Link Negotiation Protocol

Executive Summary

This document is a public deliverable of the Scalable Adaptive Internet soLutions (SAIL) EU-FP7 project and describes the implementation of the Cloud Networking prototype. The document describes an integrated prototype developed by multiple partners of the project. That prototype was tested on a test bed encompassing four different European sites hosting data centers and emulating operator networks over the Internet.

Cloud networking in the context of SAIL is about integrating the on-demand provisioning of network resources to data centre resources. Cloud computing infrastructures emcompass computing nodes, storage elements, and internal data centre networking components. All these components have to be allocated and configured to create an infrastructure in the cloud. While the configuration of computing and storage nodes is already supported by a large number of tools, the support for flexible networking within a data center is still a challenge. Let alone the allocation of network resources in operator networks, a key element when connecting users to the data centres.

In SAIL Cloud Networking, existing data centre-based IaaS resources can be interconnected through the wide area network using a network service model. To implement that a distributed control plane binding resources together was needed and is at the heart of our solution. A link negotiation protocol has been implemented to connect operator owned resources (e.g., VPNs) to data centre ones. Moreover, a new network service interface that provides the on-demand allocation of wide-area network resources was designed and implemented.

To simplify prototyping and ensure an easier migration path, CloNe has built on and extended existing accepted cloud management tools, interfaces, networking protocols and libraries. This document presents those tools, the extensions performed, specification of components implemented from the ground up, as well as how those components were integrated and how an application was deployed on the system.

Contents

1 Introduction

The CloNe work package set about to provide integrated on-demand allocation of computing, storage and networking across data centres and operator networks in a seamless way. More specifically, the prototyping activity has focused on the integration of wide area network services with infrastructure as a service providing customers with the ability to create distributed infrastructures at once. The allocation and configuration of resources across different providers is automatically performed by the system without manual intervention from the customer or provider. The customer is left with the responsibility to specify the particular needs of its application. This document describes the implementation of that prototype.

The prototype is comprised of a set of components, which were developed by different partners and incrementally integrated. The foundation was provided by the architecture task where concepts, actors, interfaces, protocols and functions were defined. Those components were either implemented from scratch or by extending existing software when suitable. A large-scale test bed was put in place to show how the implemented components would interact in a realistic scenario. The integrated prototype was deployed on that test bed and at the moment of writing a public demonstration of the system has already been performed [1].

This chapter continues by describing the role of prototyping in general and objectives of prototyping in WPD. That is followed by describing the prototyping approach and methodology in WPD and an overview of the contributions produced by this activity along with a short story of how these contributions are orchestrated. This chapter will also describe the structure and organization of this document.

1.1 Role of Prototyping

In many contexts the purpose of prototyping is to build an early model for evaluating the new concepts or enhancing the precision of details. The result of prototyping can serve as a base for specification of a real system to be built. The prototyping can focus on various aspects such as functionalities, or proof of principles. Research prototyping will always differ from software production as in latter case a more custom design may be intended. There are pragmatic and practical limitations to prototyping experimentation (for example real-time security-dependent cash transaction applications or networking across actual operator's wide area network) that imply a risk that an actual design may not perform as well as what is show cased in prototyping.

Considering the above, and the context of the research vs. production implementation when an innovative solution is being introduced, one of the main purposes of prototyping is of proof of concept nature. Prototyping also provides an early view on what a final system could look like. It assists the identification of any problems with the defined framework of concepts. Prototyping in nature can be *horizontal* showing the end-to-end solution while stubbing the details of the components involved or *vertical* focusing on specific functions and show casing their specific optimization and added-value.

Besides, the methodology followed by prototyping could be *evolutionary* by building a solid structure and continue to refine it by adding extra features and fine tuning the details or *incremental* where different components are built separately and are integrated at the end.

1.2 Objective of Prototyping

The main ambition of the prototyping activity in CloNe is to demonstrate the capabilities of an evolved Cloud Networking model. This model aims for a flexible architecture, which allows the deployment of complex applications over heterogeneous networks spread over multiple domains. One of the key features of the cloud network model is the concept of a flash network slice. Along with augmenting the networking capabilities in the data centre, CloNe also tries to add and make use of distributed processing and dynamic network provisioning capabilities into an operator controlled network environment like a WAN. These capabilities are intended to provide an enhanced platform for applications with specific demands like distributed processing.

During prototyping of CloNe, the architecture was put to test and capabilities of the cloud networking model were demonstrated. This demonstration not only shows the networking capabilities in the data centre, but also will show dynamic network provisioning in Wide Area Network (WAN) by connecting data centres. This included the development of distributed control functions and protocols, interfaces, as well as management functionalities and basic security solutions. The goal of prototyping is not only to prove some of the contributions of analytical research but also to extend the research to the details only experimentation can achieve.

1.3 Prototyping Approach and Methodology

The development of the CloNe solution was conducted in an iterative approach for realizing a novel cloud networking solution. Based on the initial CloNe architecture which defined the main concepts, interfaces and information flow, prototyping activity focused on verification of the solutions in a large scale prototype encompassing data centers connected through CloNe enabled operator networks. The prototyping task followed an iterative process where components were refined along the way (e.g., a simple version of the Link Negotiation Protocol (LNP) covering only VLANs negotiation was first implemented and evolved in to a general version, presented in this document).

Experimental research in WPD under the umbrella of prototyping played a major role in fine-tuning the details of the CloNe solution and making the abstract concepts developed in the first version of the architecture a palpable reality. The strategy was to show in a real test bed how the proposed concepts could be implemented using existing technology when available, and creating new when needed.

The nature of prototyping in CloNe was both horizontal showing the feasibility of main use cases and vertical when dealing with evaluation and demonstration of the added value and optimizations made by some of the components such as the network control layer and optimized resource management.

The methodology followed in CloNe prototyping was evolutionary when creating the real test bed that span over four different countries connecting multiple data centers together and fine tuning the detail of this test bed and communication protocols along the way. On the other hand developing the different components was performed in an incremental approach: first they were developed separately and the integration was performed after each component could be successfully demonstrated individually.

1.4 Scope and Overview of Contributions

In the lifetime of SAIL WPD, CloNe committed to the mission of providing a unified system for creation, control, management and optimization of virtual resources. Virtual resources are distributed compute, storage and network resources and the provisioning is on-demand and in an

elastic nature. This solution also supports heterogeneous technologies, domains, and providers. The main CloNe contributions that will be further described in this document are:

- Distributed Control Plane (DCP): it binds data centres and wide area networks together allowing for true automated cross-domain deployments of infrastructure. It is a set of protocols, interfaces and control operations that enable infrastructure service providers to interact and exchange cross-administrative domain information. DCP is used for reference resolution, notification of events, and link negotiation. In that context, a Link Negotiation Protocol was designed and implemented.
- Open Cloud Networking Interface (OCNI): this is an interface that was defined in CloNe for the control of networking resources both in data centres and in operator networks. The OCNI interface allows customers to specify the needed network service through a high level interface where connectivity details can be later agreed upon between involved domains. pyOCNI is the implementation of that interface.
- libNetVirt: it is an API that creates a network abstraction layer to allow for the management of different network technologies in a programable way. libNetVirt can control both OpenFlow and MPLS based networks.
- VXDL: a descriptive language used to define full virtual infrastructure that can span multiple providers, allowing for the specification of application requirements in an elastic fashion.
- Generic Resource Management Protocol (GRMP): a protocol that was developed for performing VM placement optimization in a datacenter at runtime. Two different management objectives have been developed for managing computing resources in an OpenStack based data centre.

1.5 Document Outline

This document is organized in a way to provide coherent and complete view of CloNe experimental research while providing enough details that can serve the interested reader as basis for further expansion and developments. An overview of the use-cases implemented along with an architectural context is presented in Chapter 2. Chapter 3 presents existing state of art cloud management tools and network protocols that were used to build CloNe's test bed. The individual components developed in the context of CloNe are described in Chapter 4. Chapter 5 presents a use-case centric description of how the components of Chapter 4 are placed together to form a system. Potential extensions which have been identified are described in Chapter 6. Finally, Chapter 7 presents an analysis of the main challenges faced when implementing the CloNe prototype and Chapter 8 the conclusions.

2 Use Cases and Architectural Context

The implementation of the CloNe prototyping focuses on the key novel concepts that were proposed in the first version of the architecture [2]. The purpose was not only to put to test the feasibility of implementing such an architecture, but to get a better understanding of the implications and details involved when implementing such a solution. The main concepts that has been implemented are the infrastructure service interfaces, distributed control plane, resource scheduling and optimization, network abstractions layers for data centres and wide area networks, along with an authentication solution. In a nutshell, the main feature that CloNe prototype should demonstrate is the dynamic allocation of integrated computing and network resources across multiple heterogeneous administrative domains, including data centres and networks.

2.1 Prototyping Test bed

In the CloNe context, the test bed is the hardware and software that is used as a platform for our experimentation. The software part of the test bed is composed of existing state of the art tools, systems and interfaces that have either been used as they are or added with new functionalities. For example, OpenStack is a cloud management platform that was used as part of the test bed to deploy a data centre. CloNe has also developed and implemented an advanced Virtual Machine (VM) scheduling algorithm on OpenStack. Being able to use existing pieces of software was a sensible approach that has allowed us to concentrate on specific challenges of relevance to CloNe as well as learning from existing solutions.

2.2 Use Cases Workflow

In [3] two main scenarios for Cloud Networking were defined, the Dynamic Enterprise and the Elastic Video Distribution. Within each scenario, four use cases were extensively described. The purpose of those scenarios and use cases was initially to guide the development of the architecture and later to demonstrate the novel capabilities of the underlying infrastructure developed by CloNe. During the prototype specification phase, in order to better understand the use case requirements, a comprehensive analysis of the operational steps for execution of the use cases was made. That analysis was made from a system's perspective, which means that CloNe is taken as a whole and no specific software component was identified at this moment. Despite, this work shed an initial light onto the core functionalities that needed to be developed to successfully demonstrate the CloNe capabilities.

Regardless of their different nature, the analysis performed showed that all of the use cases bear similar workflow. That meant that it was possible to generalize those execution steps with the same sequence. The operational steps involved in the execution of the use cases are pictured in Figure 2.1.

The first step, so called step zero (since it actually occurs before any [1]tenant interaction), happens when the different administrative domains discover each other. During this step a data centre domain could discover the network domain it is connected to, using a predefined (or standardized)

[1]Tenants are customers of Cloud Providers. A tenant interacts with a Cloud Provider by requesting a set of storage, network and compute resources that the latter realizes by allocating them in its physical infrastructure.

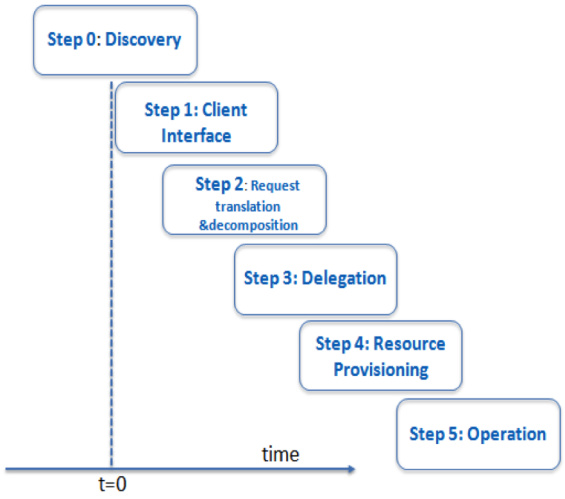


Figure 2.1: Steps of operation

discovery mechanism. General properties associated to the domain can be exchanged, e.g., jurisdiction information, provided security level, amongst others. Administrative domains should inform their peers of available resources as well. No resources are actually allocated now; this step simply makes the two domains aware of each other and their capabilities. While an automated protocol for discovery would be advisable, the small number of connecting administrative domains should justify manual configuration as well.

Step 1 is the moment when a tenant first interacts with the system. The tenant does so by sending a request to an Infrastructure Service Provider of choice. The tenant requests a full virtual infrastructure, composed of virtual compute, storage and networking connected in a particular way. The preferred way to do so is to use a declarative description language, which will allow for the specification of the full infrastructure at once, compared to individual requests for single resources.

When the request is received by the Infrastructure Service Provider, it is its responsibility to break it down into smaller infrastructure pieces that may be implemented by other providers. That process is called Decomposition. This should be done using knowledge about the properties and resources owned by other domains. High-level goals can be translated onto specific arrangements of the virtual infrastructure.

In the next step, those pieces of infrastructure are delegated for implementation to other Infrastructure Service Providers. The Delegation process (Step 3) happens through an Infrastructure Service Interface, where resources are requested. The receiving domain should be able to accept or deny the request. Step 4, is where those resources are actually provisioned in each domain (which happens immediately after the request has been accepted in step 3). Here, negotiation of links amongst the involved domains will happen through the Distributed Control Plane.

After Step 4 is completed, the pieces of the infrastructure are available and connected to each other. The full infrastructure can then be used by the tenant in the Step 5 - Operation phase.

2.3 Implemented Use Cases

When it comes to the application that should run on the developed prototype, an existing e-commerce application has been chosen. The reason for doing so is that focus should be put on the new infrastructure and networking features of cloud networking and not on the application running on top of it, which does not necessarily need to be modified. Indeed, most applications running in

data centres today are not written for a particular cloud platform deployment. For that reason, the application utilized was the open-source web shop OpenCart.

OpenCart [4] is a complete web shop solution that features not only essential functions (product categorization, multi-language multi-current website) but some advanced as well (product reviews and ratings, downloadable products, related products). It offers a simple administration page and it does not require any technical knowledge from the web shop owner once it is operational, making it very user friendly.

In order to demonstrate the deployment of the application across different data centres, the database and front-end parts has been decoupled from each other. To connect them, a flash network slice based on a L3 MPLS VPN is established. Details about the test bed can be found in 3.7.

The demonstration of the prototype happens as follows. First the application owner specifies, utilizing a GUI, the need for two VMs in different locations connected by a flash network slice, one is the database and another is the front-end server. The application owner specifies as well the need for a customer facing network interface with a public IP address. A Decomposer resolves those requirements onto the existing resources. The interfaces of the data centres and network operators involved are then invoked, OCCI and OCNI respectively. Through DCP the different domains can agree upon the details of their connections (e.g., which VLAN tags to use, routing protocols to be used for exchanging routes etc.).

2.4 Architectural context

2.4.1 CloNe Domains

As already defined in the CloNe architecture document [2], an administrative domain (hereinafter referred to only as domain) is a collection of either physical or virtual resources that are under the management of an administrative authority. Whilst CloNe does not have any limitation in terms of the number of concurrent domains present in the system at a given point in time, it defines two kinds of domains: WAN and Data Centre (DC); all participating domains belong to either of these two categories and not both at the same time. Data centre domains are pools of compute, storage and network resources that a given end-user can use to deploy his virtual infrastructure; while wide area network domains provide connectivity services, especially across geographically distributed domains.

2.4.2 Reference Domain

Figure 2.2 shows the *reference domain* and the upcoming subsections describe how the three layer domain model (resource, single-domain infrastructure and cross-domain infrastructure) defined in [2] has been implemented.

2.4.2.1 Resource Layer

A virtual resource is defined as an abstract representation of a component in a virtual infrastructure (e.g. virtual machine, volume on a block device, virtual network), and it resides within the boundaries of a single domain. A virtual resource can be classified as a compute, storage or network resource.

The resource layer encompasses the management interfaces responsible for creating and destroying virtual resources. VNET [5] and LibNetVirt [6] are the CloNe contributions to this layer and are described in Sections 4.4 and 4.5.1, respectively.

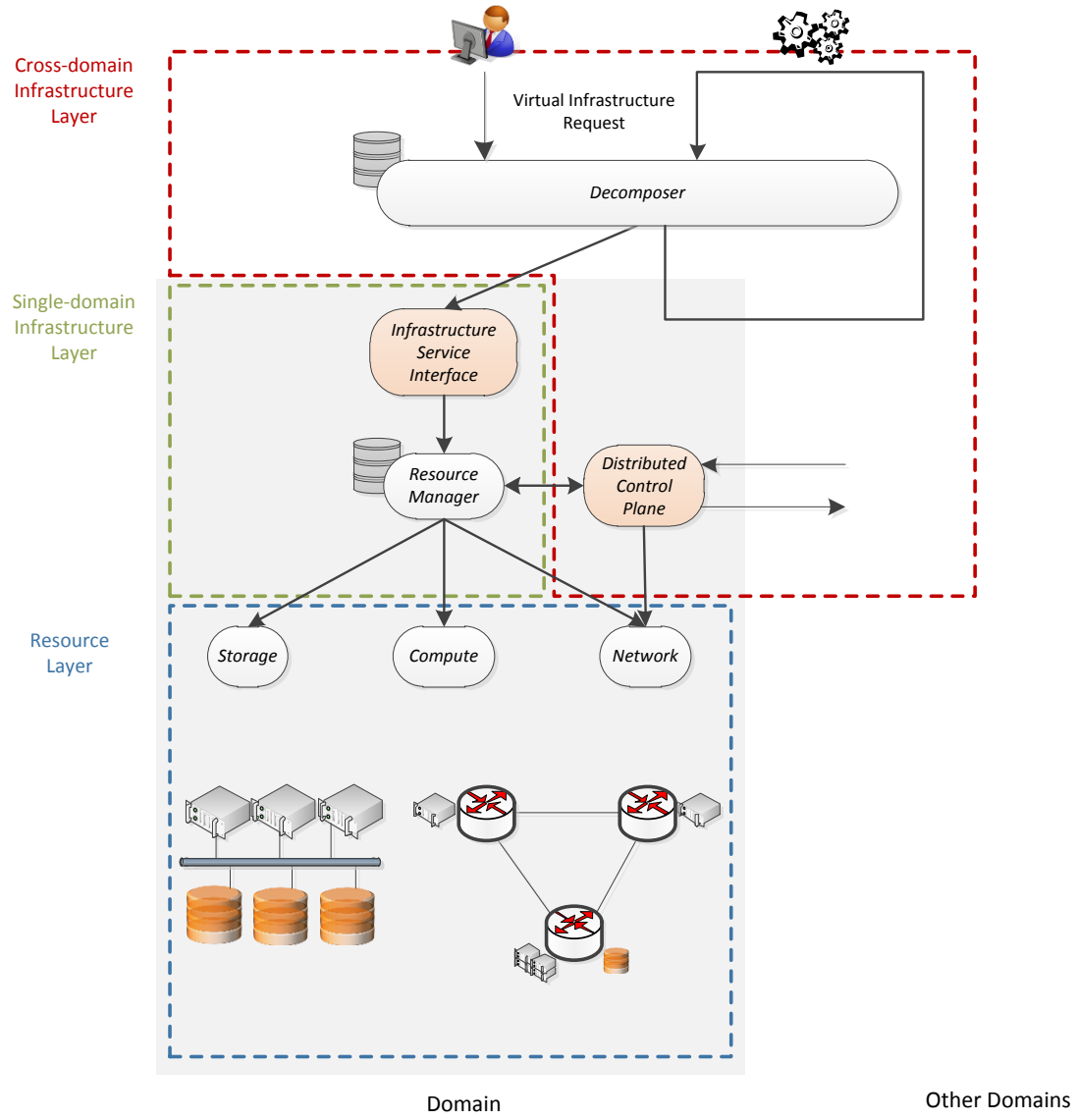


Figure 2.2: Reference Domain and its components

2.4.2.2 Single-domain Infrastructure Layer

A single-domain infrastructure is considered to be a set of virtual resources managed collectively within a single domain. This infrastructure is managed by a single administrative authority that has management rights over the underlying equipment and virtualisation technology, as well as full knowledge about the available resources and virtualisation capabilities at any time.

Infrastructure Service Interface

The *Infrastructure Service Interface* is the entry point to each one of the CloNe domains. This module is the one receiving all the incoming virtual infrastructure requests and, once parsed, responsible for delegating them to the *Resource Manager*, which will then try to find an allocation solution for the requested resources.

The use of a common interface by all the domains has always been a requirement due to the heterogeneous nature of the CloNe test bed. Rather than developing a new one for this project, the decision was taken to use Open Cloud Compute Interface (OCCI) [7] as the common *Infrastructure Service Interface* (and to extend it to make it suitable for WAN domains, as Section 4.2 describes). Thus, the different domains participating in the CloNe test bed had to implement their own back-end mapping the OCCI and Open Cloud Network Interface (OCNI) specifications to the particularities of the cloud management platforms used by Data Centres or Network Management System (NMS) used by WAN operators.

Resource Manager

The *Resource Manager* is a component that has full knowledge (allocation and physical status) about its corresponding domain's resources. This component is responsible for providing a solution to the incoming virtual infrastructure requests originated by a tenant. Once the request has been processed and a solution has been found, this component delegates to *Resource Level* (e.g. storage, compute and network managers) and/or to *Cross-domain Infrastructure Level* (e.g. Distributed Control Plane (DCP)) modules, if needed, the allocation of the requested resources.

Regarding DC domains, modern cloud management platforms include resource management functionality as one of their instrumental components. Section 3.1 lists the cloud management platforms that has been used to build the CloNe test bed and describe their main components and features.

Finally, when it comes to enhancements made to existing cloud management platforms, Section 4.5.2 describe the CloNe specific improvements in terms of management of compute resources inside the DC.

2.4.2.3 Cross-domain Infrastructure Layer

A cross-domain infrastructure is considered to be a set of linked virtual resources managed collectively across multiple domains. This infrastructure is managed by multiple administrative authorities. In contrast to a single-domain infrastructure, the state of the underlying equipment and virtualisation capabilities are likely not fully shared beyond the boundaries of the domain owning each particular resource. Hence, a specific interface via which resource virtualisation can be negotiated is required.

Section 3.3 introduces the domains participating in the CloNe test bed and briefly summarizes how they are connected to each other.

Virtual Infrastructure: Description and Decomposition

In the CloNe test bed tenants have to describe the virtual infrastructure they want to get allocated. This may not be just individual virtual resources as supported by OCCI/OCNI, but more complex infrastructure expressed in an abstract description/modelling language. Section 4.1.1 describes how Virtual eXecution Description Language (VXDL) [8] has been used for this purpose.

This abstract definition of the virtual infrastructure is later translated into concrete actions to be performed by the registered domains. The different registered domains specify some of their physical properties (e.g. geographical location), which are then compared to the requirements specified in the VXDL file. This process is covered in more detail in Section 4.1.2.

Distributed Control Plane

The DCP describes a category of protocols, interfaces and control operations that enable two or more domains to interact and exchange cross-administrative domain information.

Section 4.3 covers how DCP has been implemented, as well as provides detailed information of how the participating domains interact with each other in order to create a virtual infrastructure that spans multiple heterogeneous domains.

3 Integrated Testbed

This chapter specifically looks into the utilized cloud management platforms and network technologies. It concludes with a detailed picture about how the participating domains have been connected to each other to create what we call the unified CloNe test bed. The deployed test bed has been later used for experimentation, leading us to the development of the components further described in Chapter 4.

3.1 Cloud Management Platforms

Cloud platforms are software toolkits for managing data centre infrastructures to create private, public or hybrid IaaS clouds. They usually orchestrate storage, network, virtualization, monitoring, and security technologies to deploy multi-tier services (e.g. compute clusters) as virtual machines on the physical infrastructure. The toolkit usually includes features for integration, management, scalability, security and accounting. Another aspect is the ability to provide cloud users and administrators with a choice of several cloud interfaces (e.g. EC2 Query [9], OGF OCCI [7] and vCloud [10]) and hypervisors (e.g. Xen [11], KVM [12], to cite some) and a flexible architecture that can accommodate multiple hardware and software combinations in a data centre. Management concerns like standardization, portability and interoperability are important in this context. In a public/hybrid cloud scenario, these service interfaces are typically accessible over a communication network like Internet via a predominant Web service design model called Representational State Transfer (REST).

Three different cloud platforms have been used by the data centres part of our test bed. They include OpenStack and OpenNebula, both open source as well as Cells-as-a-Service, a proprietary platform.

3.1.1 OpenStack

OpenStack [13] is a collection of open source projects (such as Nova, Glance, Keystone, Quantum and Horizon, to cite some) to build public and private clouds. The project has been designed to deliver a massively scalable Cloud operating system. To achieve this, each of the constituent services has been designed to work together. The instrumental points are the Application Programming Interface (API)s that each service offers; while allowing each of the services to use another service (e.g. Nova uses Keystone API to authenticate all incoming requests), these APIs also allow an implementer to switch out any service as long as the new one is compliant with both inbound and outbound OpenStack interfaces.

1. **Nova** (a.k.a. OpenStack Compute) is a cloud fabric controller. It is used to create virtual machines on dedicated hosts. Among its components are:
   - nova-api: This component accepts and responds to end user compute calls. It triggers most of the orchestration activities as a response to incoming compute requests.
   - nova-scheduler: This component takes the request for a VM and then determines which compute host it should run the VM on. In order to make this decision, this component could use simple or more complex algorithms (e.g. randomly select a compute host,

chose the one that is less loaded, etc.). CloNE specific improvements of this component are described in Section 4.5.2

- nova-network: This component is in charge of controlling and manipulating the networking aspects of the cloud fabric whenever a networking request is received. This involves for example setting up virtual interface and bridging these interfaces, updating iptables [14]/ebtables [15] rules, etc. This is a legacy component and will eventually be replaced by Quantum (see below).
- nova-compute: This component is responsible for VM management tasks. These include creating and terminating VMs. nova-compute does not directly do these operations. Instead it delegates these activities to the hypervisor via APIs provided by the hypervisor or through libvirt [16].

From an architectural point of view it is worth mentioning that all the communication between the above mentioned Nova components relies on an Advanced Message Queuing Protocol (AMQP) message queue, which at the time of this writing is implemented via RabbitMQ.

2. **Glance** (a.k.a. OpenStack Image Service) provides discovery, registration, and delivery services for virtual disk images. The Glance API server provides a standard REST interface for querying information about virtual disk images stored in a variety of back-end stores, including OpenStack Object Storage and Amazon Simple Storage Service (S3). Clients can register/upload new virtual disk images with Glance, query for information on publicly available disk images, and use the client library for downloading them.

3. **Quantum** is a component aimed to provide "network connectivity as a service" between interface devices (e.g., network interfaces) managed by other OpenStack services (e.g., Nova). The Quantum service exposes an API that presents a logical abstraction for describing network connectivity. This service delegates to a plugin that implements the service to manage virtual and physical interfaces and switches within the data centre in order to make sure those devices forward packets according to the behaviour defined in the logical model specified in the API. CloNe specific improvements of this component are described in Section 4.3.2

4. **Keystone** is a component that provides Identity, Token, Catalog and Policy services and it is commonly referred as OpenStack's Identity Service. Keystone is responsible for confirming the identity of a user or the truth of a claim. Keystone will confirm that incoming requests are being made by the user who claims to be sending the request. These claims are initially in the form of a set of credentials (username and password, or username and API key). After initial confirmation, Keystone will issue the user a token that the user can then provide to demonstrate that their identity has been authenticated when making subsequent requests.

Figure 3.1 illustrates the flow between the above mentioned OpenStack services, and it can be summarized as follows:

- Nova stores and retrieves virtual disks (a.k.a. images) and associated metadata in Glance.
- Nova relies on Quantum to create logical networks and to plug Nova VMs in these networks to enable connectivity between them.
- Eventually all these services authenticate with Keystone.

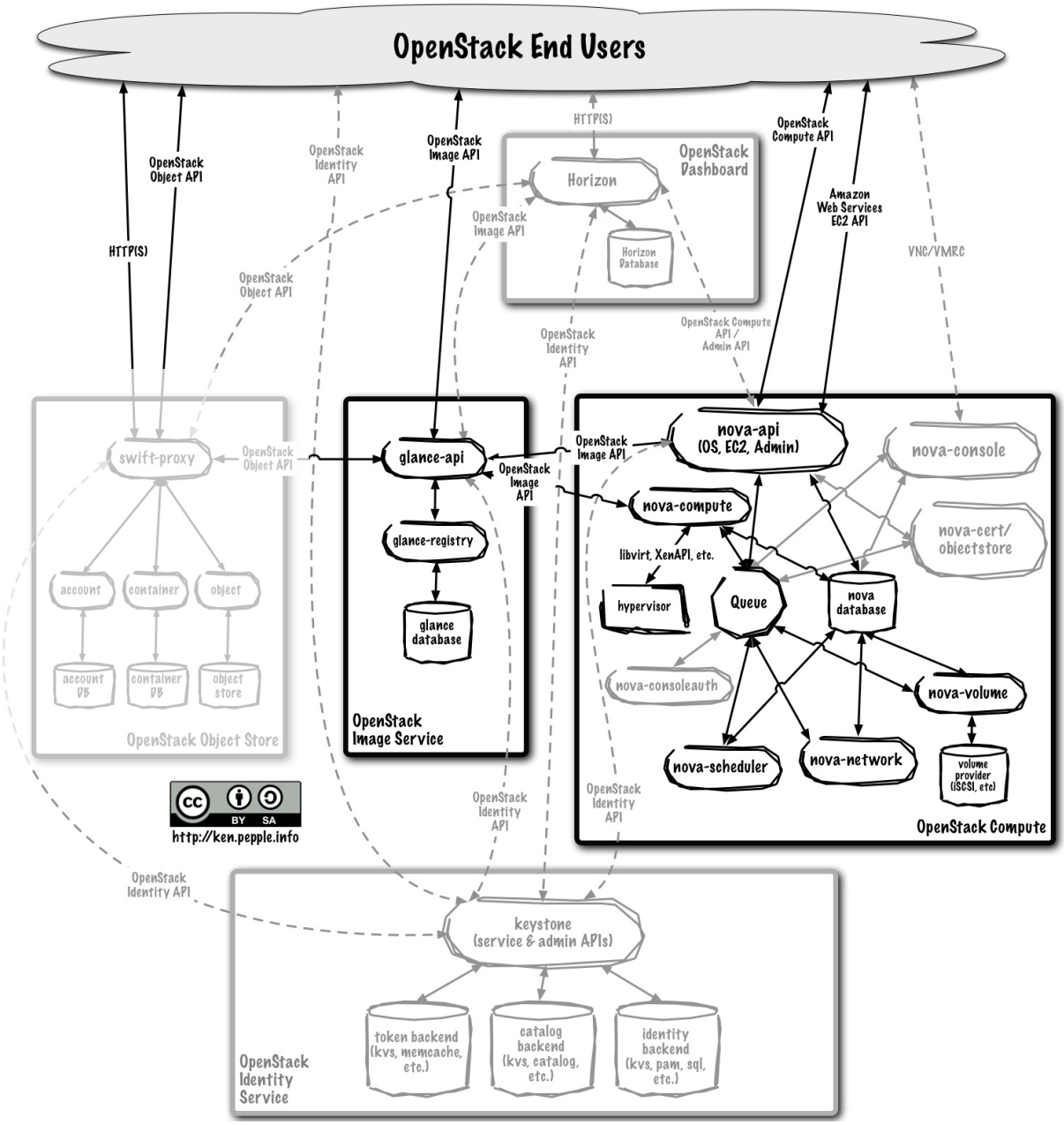


Figure 3.1: Flow between OpenStack Services

3.1.2 OpenNebula

OpenNebula [17] is another popular open source toolkit for building and managing virtualized enterprise data centers and cloud infrastructures. OpenNebula aims to provide an open, flexible, extensible, and comprehensive management layer to automate and orchestrate the operation of virtualized data centers by leveraging and integrating existing deployed solutions for networking, storage, virtualization, monitoring or user management.

Unlike OpenStack, OpenNebula is a single project that possesses an adaptable and extensible architecture with tools, interfaces and components that can be easily modified for a customized cloud service or product.

The interfaces of OpenNebula (See Figure.3.2) can be classified in two categories:

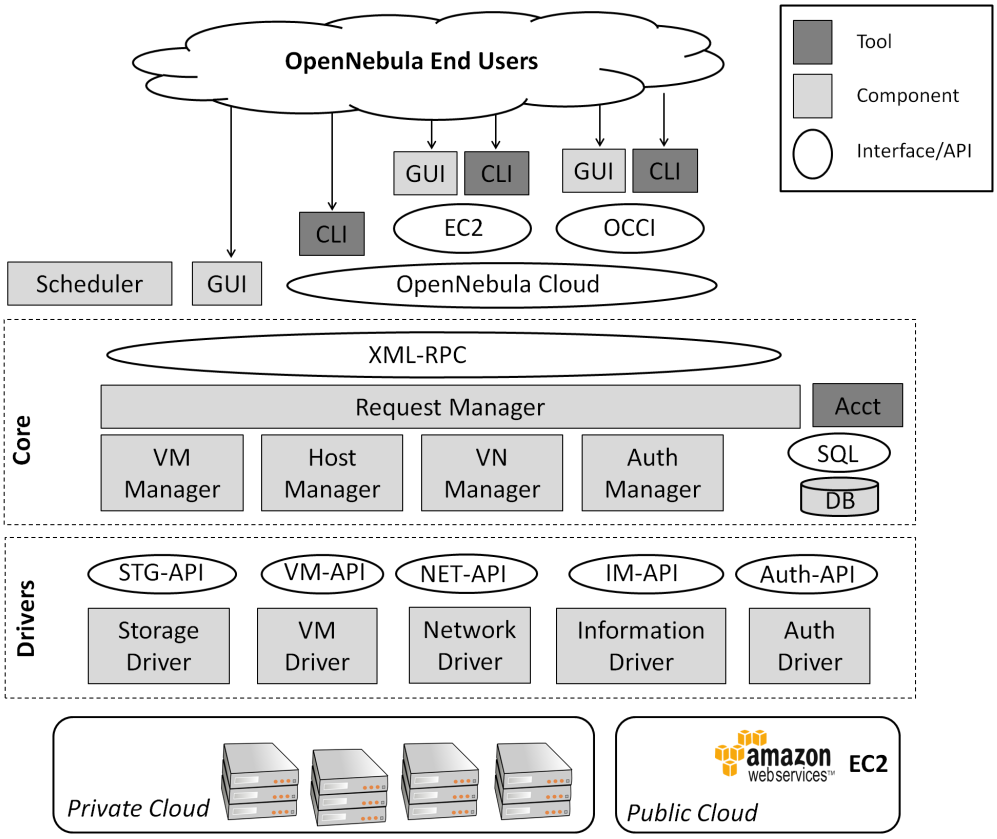


Figure 3.2: OpenNebula Architecture

1. **Tenant Interfaces**: These are public facing interfaces accessed by the tenant to manage virtual machines, networks and images through REST APIs. These interfaces hide most of the complexity of the Cloud and are specially suited for end-users. OpenNebula implements two different tenant interfaces, namely:
   - EC2-Query API: OpenNebula implements the functionality offered by the Amazon's EC2 API, mainly those related to virtual machine management. In this way, one can use any EC2 Query tool to access the OpenNebula Cloud.
   - OCCI: This service enables one to launch and manage virtual machines using the latest draft of the OGF OCCI API specification.
2. **System Interfaces**: While the tenant interfaces are primiarily suited for end users and provide only a high level functionality of the cloud, system interfaces expose the full functionality of OpenNebula and are mainly used to adapt and tune the behavior of OpenNebula to the target infrastructure.
   - OpenNebula XML-RPC Interface: This is the primary interface for OpenNebula, and it exposes all the functionality to interface the OpenNebula daemon. Through the XML-RPC interface you can control and manage any OpenNebula resource, including virtual machines, networks, images, users, hosts and clusters.
   - Cloud API (OpenNebula Cloud API (OCA)): This interface provides a simplified and convenient way to interface the OpenNebula core. The OCA interfaces exposes the same functionality as that of the XML-RPC interface. OpenNebula includes two language bindings for OCA: Ruby and Java.

- Drivers Interfaces: The interactions between OpenNebula and the Cloud infrastructure are performed by specific drivers each one addressing a particular area (storage, virtualization, monitoring and authorization).

OpenNebula provides a set of tools and components for the end user to easily interact with the framework, namely: CLIs for the Amazon's Elastic Compute Cloud (EC2), OCCI and OCA interfaces, as well as two graphical user interfaces (GUIs), one works on top of the XML-RPC (a.k.a. Sunstone) and the other on top of OCCI (a.k.a. Self-Service).

Among the various components that integrate the OpenNebula project it is important to highlight the core ones: Request Manager, VM Manager, Host Manager, VN Manager and Authentication Manager.

- Request Manager: component that decouples most of the functionality in the OpenNebula core, from external components. It exposes a eXentisible Markup Language - Remote Procedure Call (XML-RPC) Interface, and depending on the invoked method a given component is called internally, i.e., VM Manager, Host Manager, VN Manager and Authentication Manager.
- Virtual Machine Manager: component responsible for the management and monitoring of VMs.
- Virtual Network Manager: component responsible for the management of virtual networks. It handles Media Access Control (MAC) and Internet Protocol (IP) addresses and their association with VMs and the physical bridges of the VMs.
- Host Manager: component responsible for managing and monitoring the physical resources.
- Auth Manager: component responsible for authorizing and authenticating user requests.

All the these components rely on a set of pluggable modules, i.e. drivers, to interact with specific middleware (e.g. virtualization hypervisor, cloud services, file transfer mechanisms or information services).

3.1.3 Cells-as-a-Service

Cells-as-a-Service (CaaS) is the HP Labs implementation of an infrastructure service that provides the properties that address the issue regarding the privacy of data, security and the reliable delivery of critical components of the enterprise's IT. It focuses on the requirements for enterprise-grade virtualised cloud computing infrastructure with a guarantee over the security in term of isolation.

Cells are isolated combinations of virtual resources, connected to create the desired service infrastructure, and form the basis of the way in which service providers manage these resources, control access to the services that run upon them, and allow the services to be self-adaptive. Generally, cells are created on demand for a service provider via a portal with each cell created initially containing only a Cell Controller. This controller is responsible for securely interacting with the service provider acting as the only access point through which to request and manage virtual infrastructure. It responds to the service provider with the status of the virtual infrastructure indicated.

Each Cell contains a collection of virtual components including virtual machines (VMs), storage volumes and subnets (network components) declared as elements in a Cell model. The specification also describes how these components are connected to create the requested infrastructure. Moreover, each component or connection may be defined with a set of relevant attributes. For example, VM elements include specifications for memory requirements, bus addresses for volume attachments, connection to one of a number of Cell subnets and behaviour in the event of failures.

Volumes may be specified with a given size or initialised using an existing volume as an image. Note that, the Cell model is typically an XML document (see 3.3). Changes to the model may be handled by submitting an updated model document or through an API that supports incremental changes to the model, and this may be invoked by the service provider externally to the cell, or from within the Cell by the service itself.

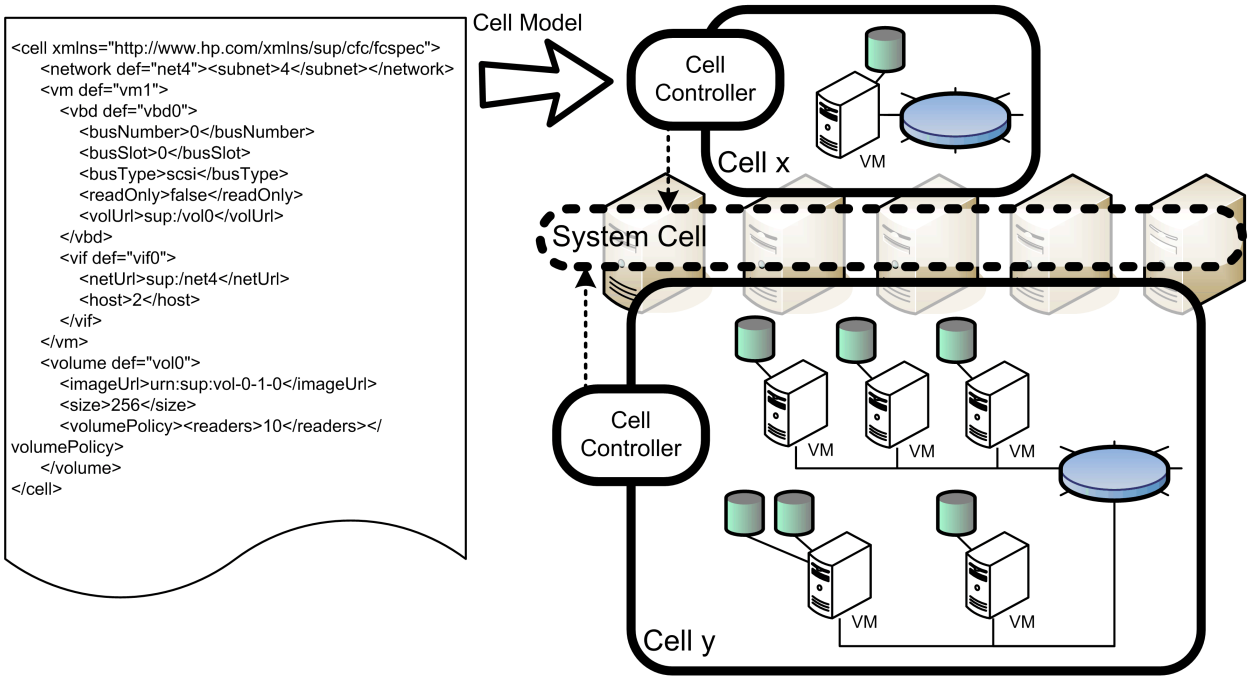


Figure 3.3: Cells-as-a-Service Prototype

The boundary of a Cell and any separation of components within the Cell is enforced by the underlying system. This boundary consists of both network connectivity between hosts and subnets of the Cell and between Cells, plus the ability to mount volumes owned by other service providers. By default, the traffic is only permitted between VMs on the same subnet and volumes are only visible for connections and imaging within the same Cell. However, additional rules can be enforced through the model to allow shared volumes between two or more Cells as well as permit communications between different subnets either in the same Cell or across Cells. For connectivity between different Cells, the rules added need to be reciprocal.

This management is entirely handled by a special privileged Cell known as the System Cell. No part of any other Cell may communicate directly with the System Cell apart from a locked-down and secure bastion component of each Cell Controller. The System Cell is responsible for creating and deleting all the virtual components and managing their connectivity, for enforcing all the connectivity policies defined within each Cell and for Cell interaction, and for enforcing any policy regarding recovery or scalability limits associated with each Cell.

The System Cell runs across all the physical hosts within the infrastructure, each of which must be running a hypervisor (in this case, KVM). The System Cell contains two types of component: Host Managers and core system services such as Resource Management and Storage Management. A Host Manager runs on each physical host within the privileged VM (often known as the host OS) and is responsible for managing and validating every action that occurs on that physical host such as the creation of new virtual machines or the communication between two virtual entities.

Each Host Manager enforces the isolation of Cells from each other and from the System Cell by mediating access to the physical hosts compute, network and storage capabilities. The Host Manager transforms abstract VM model elements into configuration data appropriate to the hypervisor

and uses its command interface to manage VM lifecycles according to desired state expressed in the Cell model.

The Host Manager interacts with a Storage Manager to create and remove virtual block devices as required by hosted VMs. When required, the Storage Manager initialises a new volume from an image volume using copy on write techniques, thereby achieving near instantaneous initialisation with the added advantage of reducing storage requirements by making the copy on write volume smaller than the image. This is implemented in the Host Manager as directed by the Storage Manager. It is implemented as a kernel module which also has the capability to cache blocks on local host disks in order to offload SAN traffic and so considerably improve performance in certain situations.

Cell subnets are implemented as virtual overlay networks on a single shared physical network. Virtual networking typically relies on hardware routers to convey encapsulated packets between virtual subnets. This has been avoided in the Cells implementation because there is no standard interface for managing routers. Instead this has been implemented using a novel fully-distributed virtualised router that facilitates single network hop communication between endpoints; unlike traditional software routers, this solution operates at the OSI network layer so that packets can be forwarded directly to their destination. Every physical host implements a part of the distributed virtual router by filtering and modifying all packets from or to its local VMs according to the network rules expressed in the Cell models.

To support the requirement to manage overall performance qualities of service, the networking layer provides networking resource control to limit and prioritize VM's bandwidth consumption. This controls both the contention on the host for the VMs wishing to use the network and potential congestion within the network switches. This resource control also applies to networked storage I/O to manage overall loadings on the storage system. The resultant prototype system has proven to be highly scalable and resilient.

3.2 Networking Technologies

A plethora of different networking technologies are used for network virtualization and tenant isolation in the DC and Network Operator (NO) domains that together constitute the overall test bed. This has allowed experimentation with Flash Network Slice (FNS) management across heterogeneous infrastructures, which is the reality of operational networks today and likely also in the future. The DC domains either use legacy network technology, such as Virtual Local Area Networks (VLANs) or more recent approaches. These include L2 frames tunnelled on top of IP, VNET (Subsection 4.4.1), or OpenFlow-based solutions.

The NO domains (i.e., the WAN providers) use commonly deployed network technologies, in the form of IP/MPLS networks, but also more forward-looking but less operationally proven solutions based on OpenFlow. In both cases, the infrastructure is able to provision layer 2 (Virtual Private Wire Service (VPWS) and Virtual Private Line Service (VPLS)) and layer 3 Virtual Private Network (VPN)s.

3.2.1 MPLS Networks

One important objective of CloNe is to bring solutions applicable not only to future networks but also currently deployed network technologies. Multi-Protocol Label Switching (MPLS) has for many years been a popular transport technology for IP and other services, especially among large telecommunications carriers. Thus, it has been desirable to prototype and validate the CloNe solutions against an MPLS infrastructure. To that end the test bed includes two interconnected IP/MPLS networks; one at Ericsson premises and another at PTIN premises. The former makes

use of virtual routers running a commercial IP/MPLS control and data plane network stack and the later makes use of commercial physical routers.

Both the Ericsson and PTIN networks have a topology consisting of four core (P) routers interconnected in a full-mesh. These P routers then each connect to one of four Provider Edge (PE) routers. The two WANs are interconnected via a Provider Edge (PE) router in each network. Data centres and customer sites (i.e., partners in the project) are also attached to either of the WANs via a PE.

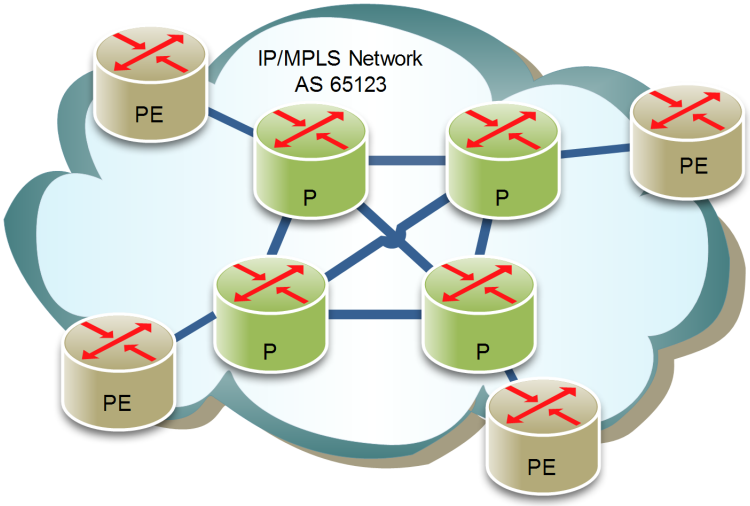


Figure 3.4: PTIN WAN Test bed - IP/MPLS Network

The PTIN WAN test bed, which is located at Aveiro (Portugal), is depicted in Figure 3.4. The four core routers are Cisco 3700 routers, whereas the four edge routers are a mix of Cisco 2800 and 7200 routers. All the core (P) and edge (PE) routers run Open Shortest Path First (OSPF) to learn topology and configure routes, and Label Distribution Protocol (LDP) as a way to dynamically exchange the MPLS labels. The edge routers also run Border Gateway Protocol (BGP), configured to announce the Autonomous Systems (AS) 65123, and used to exchange the VPNs routes among all PE peers. The BGP can be also used to exchange routes with other NOs, i.e., the Ericsson WAN. The PTIN WAN test bed either support the setup of L3-VPNs or point-to-point L2-VPNs (VPWS). For the case of L3 VPNs, customer routes can be exchanged using BGP, OSPF, or Routing Information Protocol (RIP).

Ericsson site also has a WAN utilizing IP/MPLS. This network is fully virtualized. The routers run as VMs using Kernel-based Virtual Machine (KVM) as the hypervisor. It uses a network stack similar to the commercial hardware based router. The full details of the Ericsson WAN domain is shown in Figure 3.5. The nine router VMs are distributed across four physical servers (*'sailcompute1'* through *'sailcompute4'*) and interconnected using virtual switches (named *'br#'* in the figure). Each of the provider (P) routers have two PE routers connected to it.AS number is 65183. OSPF and LDP are used for routing table maintenance and label distribution, respectively. The Ericsson WAN supports L3-VPNs and L2-VPNs (VPLS) and customer routes can be exchanged using BGP or OSPF.

3.2.2 Openflow

OpenFlow[18] is a standard that enables researchers to run experimental protocols in existing production networks. It decouples the control and the forwarding plane of a network element

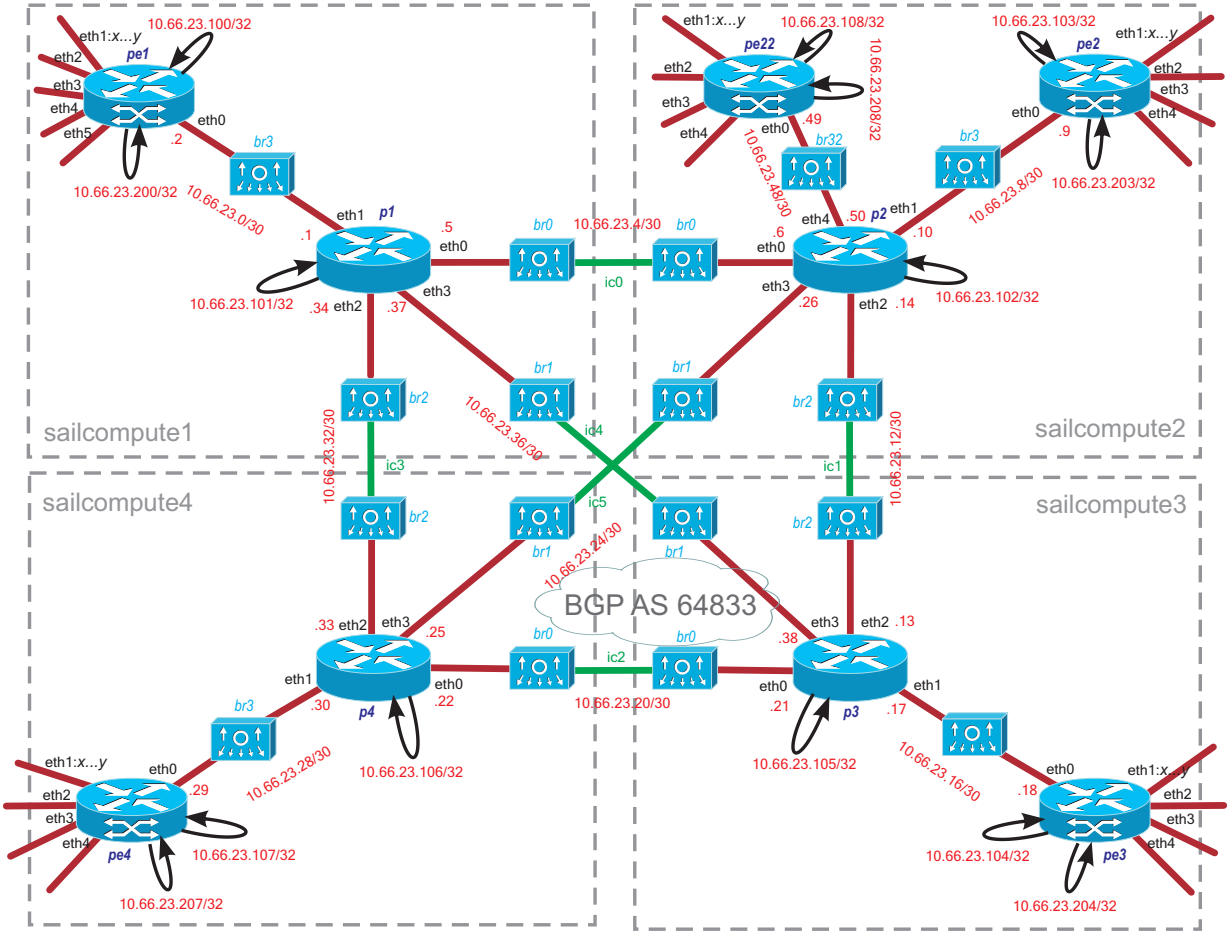


Figure 3.5: Ericsson WAN Testbed - IP/MPLS Network

(switch or router) and specifies a protocol (OpenFlow protocol) for interaction between them.

This enables the creation of a centralized controller (for example NOX [19]), which can communicate with the (OpenFlow enabled) switches in order to control their flow tables. The remote access simplifies the management of the network, because it provides a global view of the network domain, which enables the automated creation of isolated networks of the different network elements. Using a controller with a global view, it is possible to establish FNS in a complete network domain (Data Centre or Network Operator). Once the FNS is set up, the central controller does not need to be involved in the traffic forwarding, which is installed in the affected switches.

OpenFlow was used both inside the Data Centre to manage the network virtualization and to provide independent networks to the different tenants and inside the NO domain (Section 3.3).

OpenFlow has been used in the data centre in order to control, manage, virtualize and isolate traffic between virtual machines. It is running on servers that are controlled by a cloud platform (e.g., OpenNebula). One of the major advantages of using OpenFlow in data centres is the flexibility to detect and manage the mobility of VMs.

The OpenFlow network domain at IT is depicted in Fig. 3.6. This network is composed by 5 OpenFlow switches running on different VMs and by a NOX controller running on another VM. This OpenFlow network domain is connected to the MPLS network (Section 3.2.1) via a GRE tunnel.

In addition to its use inside the data centre we have also used OpenFlow in the wide-area network (Section 4.3.3.2) as well as supported it as a network technology in the network virtualization library,

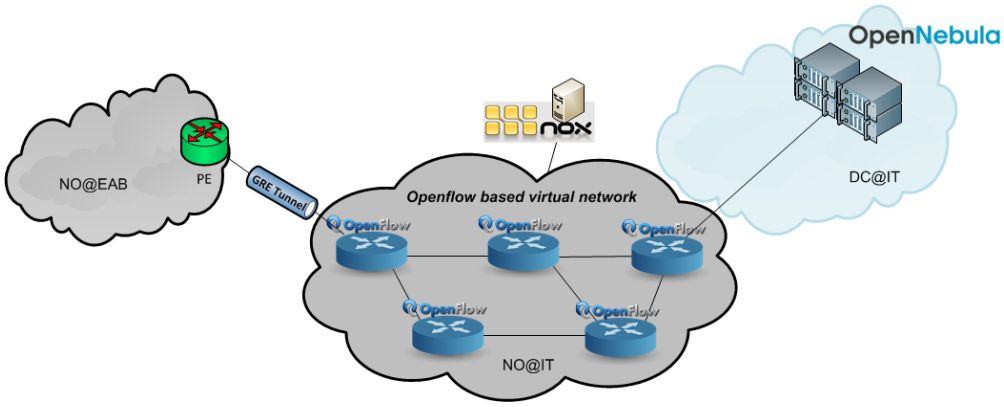


Figure 3.6: IT - OpenFlow network test bed

LibNetVirt (Section 4.5.1) we have developed. OpenFlow in libNetVirt permits a flexible and fast way to verify that a particular resource belongs to a specific FNS and to establish end to end path between two resources in a single domain. If multiple domains are present, OpenFlow can also be used to interconnect them.

The use of OpenFlow in some parts of the SAIL test bed provided us with the following experiences:

- The use of a global view permits the use logically centralized algorithms to enforce isolation in the network.
- Using OpenFlow for fault management permits quick reaction if a disruption in the network is created. The global view permits receiving notifications when a network element is down and reroute the affected traffic to another path.
- However, if the use of OpenFlow is reactive (the path is computed and installed when a flow arrives), a small delayed in introduced in the first packet of the flow.
- OpenFlow controller can be tested in a simulated environment, which can be equal to the real environment. This permits the experimentation with the network for unexpected events without affecting the production network.

3.3 Interconnecting Domains

The CloNe integrated test bed is a heterogeneous one composed by 7 different domains, 4 Data Centre domains and 3 Network Operator domains which are physically located at 4 different european countries, and it is depicted in Figure 3.7. The DC1 domain is located at HP premises (Bristol, UK), the DC2 and NO1 domains are located at Ericsson premises (Stockholm, Sweden), the DC3 and NO3 domains are located at Institute Telecom premises (Paris, France) and the DC4 and NO2 domains are located at PTIN premises (Aveiro, Portugal).

The overall test bed has been designed with flexibility in mind, allowing it to provide different testing options. Each DC domain is interconnected with the remaining DCs via, at least, two different NO and also the NO are interconnected. This allows the DC to connect to multiple NOs, as well as avoiding the dependency of all traffic being routed through one site. Also, the interconnection between NOs has been created with an ambition to look further into inter-network provider aspects and associated prototyping challenges. The interconnection details are provided further below.

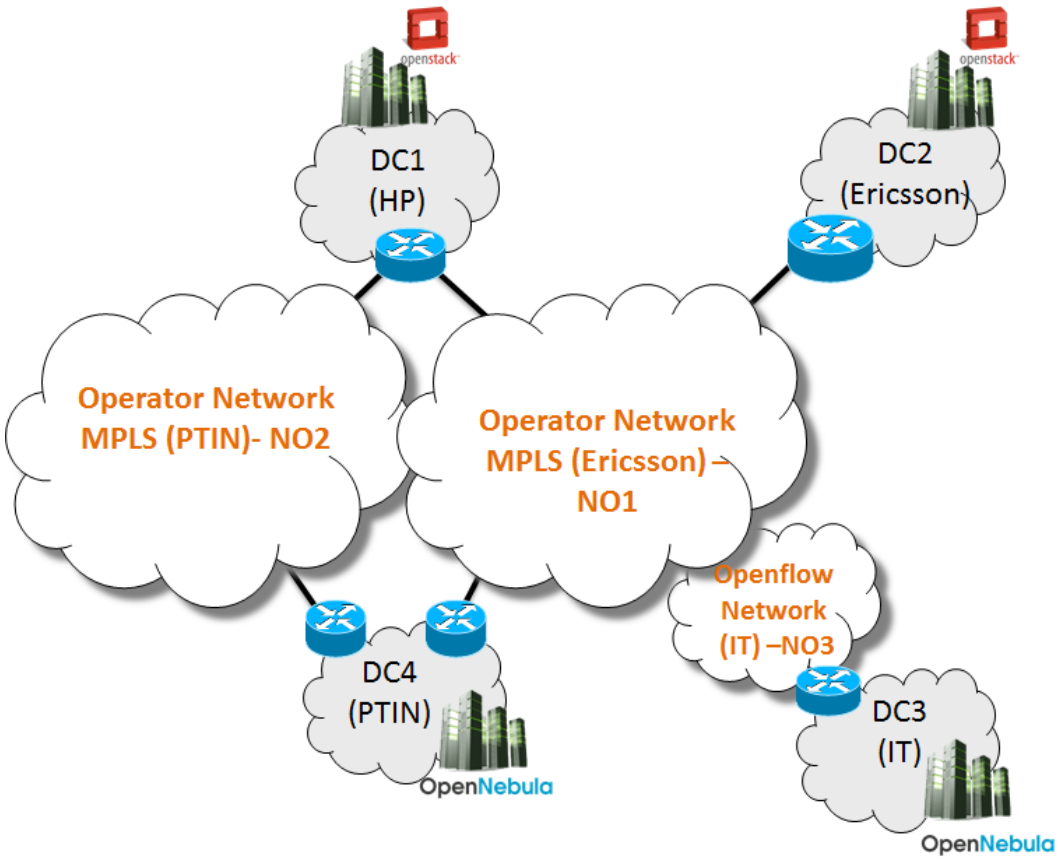


Figure 3.7: Test bed Overview

For cost and time reasons, the project decided to rely on Internet for connectivity between all sites. The different domains in the testbed, i.e. DCs or pair of DC plus NO, are interconnected using IPSec tunnels, in order to create an isolated but integrated test bed. To emulate a logical link layer 2 connection between the sites, Generic Routing Encapsulation (GRE) tunnels are used on top of the pre-existing Internet Protocol Security (IPSec) tunnels. This approach, not only enables the emulation of a direct layer 2 connection, but also allows the setup of new logical layer 2 connections, e.g. VLANs, in a dynamic way and without needing to configure additional equipment. Figure 3.8 provide more details on the setup used to interconnect the domains.

Open Virtual Switch (OVS) [20] is an open source virtual switch that is now part of the stock Linux kernel distribution. One of the features of OVS is the capability to use GRE tunnels as virtual links between different running instances of OVS. Such a tunnel is treated by OVS as any other link and can thus be configured with for example VLAN trunks. This is a good match to the requirements of the CloNe test bed.

Hence, each site in the test bed use one or several OVS instances to interconnect to the other sites. As shown in the figure, OVS runs in some physical machine on site and a GRE tunnel port is created on the OVS instance. Through configurations that are partially external and site-dependent, this GRE tunnel is routed to the IPSec tunnel that reaches the targeted site (i.e., the other domain). A similar setup exists on the other site.

Figure 3.8 illustrates the case where a CE router is connected to a PE router and VLAN 5 is used to create the tenant logical link. In the figure the PE is dynamically configured with a logical NIC for VLAN 5 (shown as the red VIF rectangle). The OVS instance needs to trunk VLAN 5 on the two involved ports. On the CE side, the GRE port trunks VLAN 5, whereas the other

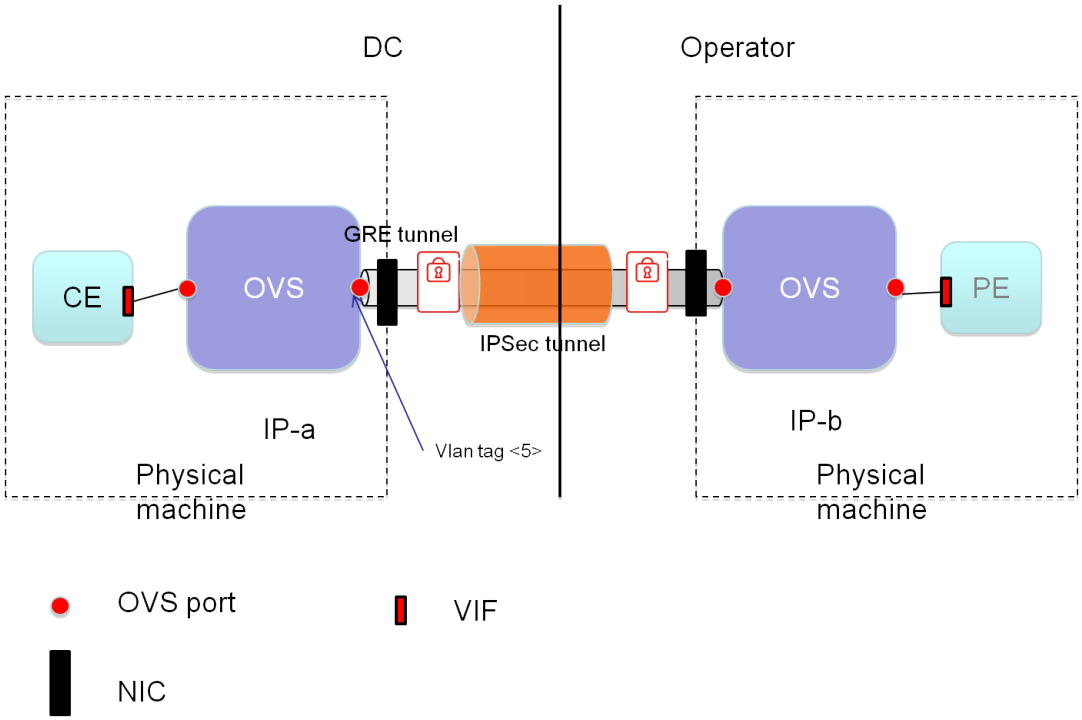


Figure 3.8: Setup used to interconnect the domains over the Internet

involved port facing the CE is configured as an access port for VLAN 5. This is a simplified example just to illustrate the basics of site interconnection and relation to tenant isolation. In practical deployments, the DC side is more complex as the tenant logical link VLAN 5 needs to be stitched with the isolation mechanism used by the DC virtual network solution (e.g., Quantum in OpenStack or VNET in Cells-as-a-Service).

Leveraging the Internet and IPSec/GRE tunnels for underlying inter-site connectivity means that bandwidth guaranteed links are not achievable and that MTU is reduced due to IPSec and GRE header overhead. However, these limitations are not crucial, nor do they undermine the usefulness of the test bed as the focus of CloNe is primarily on control plane aspects, rather than to maximize application throughput in the prototyped use cases.

4 Components

This chapter describes the components developed within the scope of the CloNe prototyping activities. In Section 4.1 contributions on how to represent a Virtual Infrastructure and how to handle Virtual Infrastructure requests are presented. Section 4.2 describes CloNe's significant addition in terms of interfacing with clouds, particularly when it comes to interact with WAN domains offering inter-datacentre services. Implementation details regarding the required inter-domain interaction enabling cross-domain virtual infrastructure deployments is described in Section 4.3. CloNe contribution on data centre networking in the form of adaptation of the early VNET[5] work to OpenStack is detailed in Section 4.4. In Section 4.5 the management concepts and functions that have been developed in prototype are described. Finally, in Section 4.6 we explain how identity management in a distributed cloud scenario can be addressed.

4.1 Virtual Infrastructure

A Virtual Infrastructure (VI) is a structured aggregation of heterogeneous computing and communication resources. A VI is time limited. It consists of sets of resources in charge of processing, storing and communicating data plus the underlying interconnection topology including configurable communication equipments (i.e. associated virtual network).

As part of the CloNe prototyping, the end-user needs to be able to describe its virtual infrastructure in an abstract description/modelling language. This abstract definition of the virtual infrastructure will be translated into concrete constraints by the goal translation function. These constraints will then be used by the Decomposer functionality to compute an allocation of the requested resources on the physical resources at different providers.

4.1.1 Describing Virtual Infrastructure

The users need a model to define their VMs, their network topology and other high level requirements. For this purpose a descriptive language called VXDL has been used. The VXDL [8] [21] is a language that allows the modeling and formal description of a Virtual Infrastructure.

The VI concept and its associated VXDL description language present four key points:

1. Simplicity and abstraction of complex virtual infrastructures for high-level manipulation. A VI description must give the functional, temporal and capacity of the aggregate as well as of each of its components. A VI description can be completely agnostic of hardware, protocol as well as geographical level dependencies. Physical details can be integrated if necessary.

2. The VI description natively integrates the fact that elements are not independent and that they are interconnected. The joined specification of networking elements and computing elements is made possible but is not mandatory. VXDL can be used to model computing resources only or to model a virtual network only.

3. The timeline specification is used to describe the dynamic aspect of virtual infrastructures. Start time and the duration must be specified. Intervals can be specified within the VI lifetime to decompose its phases, which is where resources must be really reserved;

4. The specification of attributes representing the expected Quality-of-Experience enables Service Level Agreement (SLA) definition.

The VXDL language mainly aims at enabling the description of pool of resources and networks that are virtualized but are also, to some extent, adaptable to the modeling of physical pools or networks.

Document Structure

A VXDL document is described using XML and adheres to the normative XML Schema. A VXDL files includes four areas as shown in Figure 4.1

1. The general description corresponds to the identifier of the VI object.
2. The resource description part, where individual or group of resources is specified and constraints are defined in terms of performance, security, reliability, configuration and visibility.
3. The topology section, which models the interconnection of the individual entities. All the links may be controlled and weighted.
4. The timeline, which enables to program the VI.

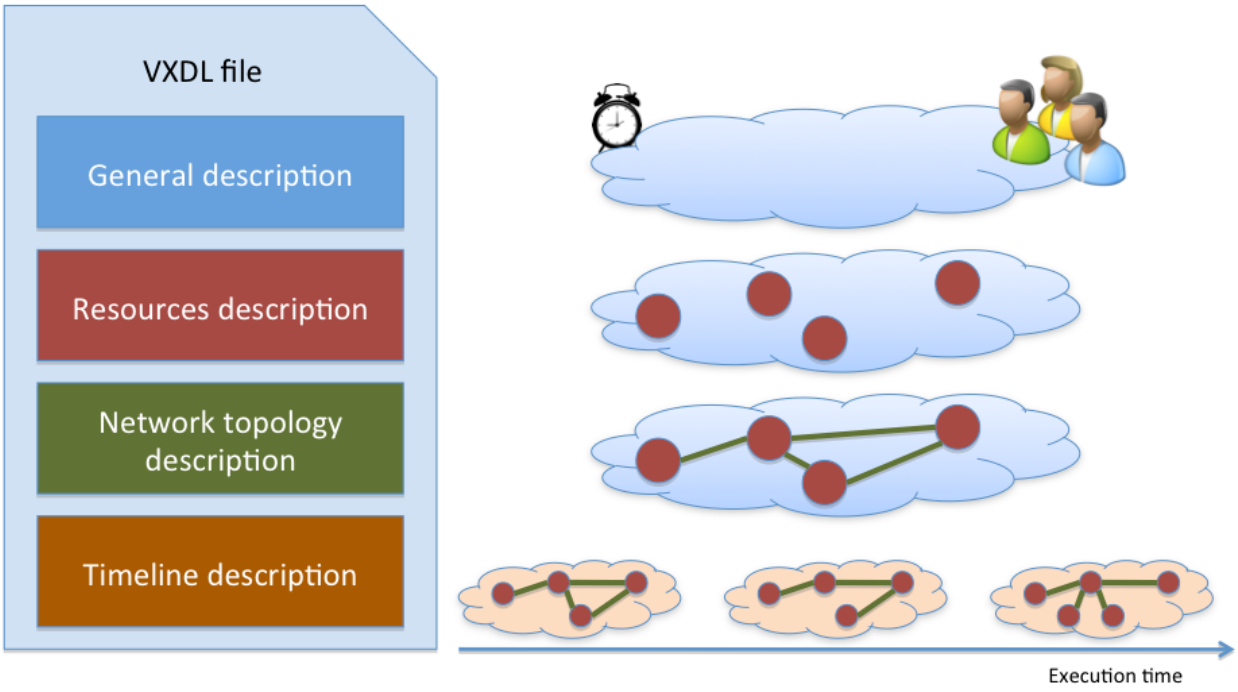


Figure 4.1: Structure of VXDL document

4.1.2 Decomposer

The goal of the Decomposer module is to convert abstract or high level requirements into concrete actions to be performed on physical resources. Using a VXDL file as an input (either generated from the GUI or provided as a file), this module makes the translation from the VXDL request to a set of node and link requirements and allocates them to the available physical resources.

The different providers (data centres or NO) are registered on the decomposer and specify some of their physical properties (e.g., their geographic location). The VXDL request is split according

to the requirements on the physical resources and Infrastructure Service Request messages are sent to the providers matching the requirements.

The allocation process, inside the decomposer, maps a graph of virtual resources on a graph of declared network and data centre resources.

The graph-embedding problem is well-know to be NP-hard [22]. There are numerous works on solving the problem with heuristics based on path-splitting methods [23], multi-commodity flow modeling [24], and substrate characteristics [25]. Our algorithm [26] solves this process as a subgraph isomorphism detection [27], [28] (which is solvable in polynomial time) to incorporate the allocation constraints and to examine the metrics proposed.

It is based on the optimization of the allocation quality constraint. Allocation quality of VI is the average of all distances (calculated in hops) between the virtual resources and one specific geographical landmark (the reference point). More specifically, the reference point can be

- The location of the user.
- The location of a certain virtual resource, as specified by the user.

The virtual and physical components being compared are called candidates. Usually, in a subgraph-isomorphism detection, a virtual candidate is tested with all candidates. This approach requires a large number of comparisons between requested and available resource capacities (for example, to identify the shortest physical path that can host a virtual link), and consequently results in a elevated computational cost to find an allocation solution.

To accelerate the processing and make the algorithm scalable on multi-domains and complex Vitual Infrastructures, we have produced an allocation heuristic to allocate virtual infrastructures. The algorithm uses an extension of this method, able to exploit the location of virtual resources. The goal is to reduce drastically the set of solutions in a first time. Initially, the location-aware algorithm identifies the set of physical landmarks specified by the user, as well as the set of virtual resources without location constraint. An iteration is performed on those sets. Each time, one physical location specified by the user is defined as the required location constraint for components that do not have this information. At this moment, a subgraph-isomorphism detection is performed to find a map solution. If no allocation solution is found for this configuration, the location constraint is relaxed for those resources, i.e., the locations precision is decreased.

We adopted a multi-thread implementation of the proposed solution. The execution is finished when all threads return an empty answer, or when a map solution is found. Our implementation makes use of future objects and synchronization mechanisms available in the Java language.

As a single provider might not have enough resources to implement the user request or the user request might specify the use of different providers, it is necessary to be able to delegate the deployement to multiple providers.

To implement this feature, the decomposer can automatically generate the appropriate (intermediate) requests to Network Operators to instantiate one (or several) virtual links.

4.2 Infrastructure Service Interfaces: OCCI & OCNI

Each domain participating in the CloNe test bed needs an interface to expose and access their offered resources. This is done by means of the *Infrastructure Service Interface*, which needs to be implemented both by DC and NO domains. Due to the heterogeneous nature of the CloNe domains, a common and consistent interface facilitates the orchestration of resources across multiple domains running different *Cloud Management Platforms*. The interface of choice is OCCI [7]. In contrast to Amazon EC2 [9], Apache Deltacloud [29], OpenStack and OpenNebula interfaces, OCCI [7] is an open standard developed by Open Grid Forum (OGF) providing both a rich data model and a

concise set of operations facilitating the capture of the state and the management of the network, compute and storage resources offered by any *cloud provider*, respectively. Its generic data model makes it fairly easy to extend OCCI with new and more specific features; as is the case of OCNI, which extends the basic OCCI model not only to enhance the current intra-datacentre network offering, but also to include inter-datacentre features.

Current generation of Infrastructure as a Service (IaaS) platforms and their corresponding interfaces focuses mainly on compute and storage resources inside data centres with networking as an after-thought to fulfill mostly data centre specific networking requirements. If intra-datacentre networking is well supported by existing IaaS interfaces, inter-datacentre networking has not been fully addressed. Inspired by the basic model of OCCI, we have developed a network specific service interface to fill this need. It is called OCNI, which stands for Open Cloud Networking Interface. In other words, OCNI is a network specialization of OCCI. OCNI can be used both in a data centre area and the network operator area.

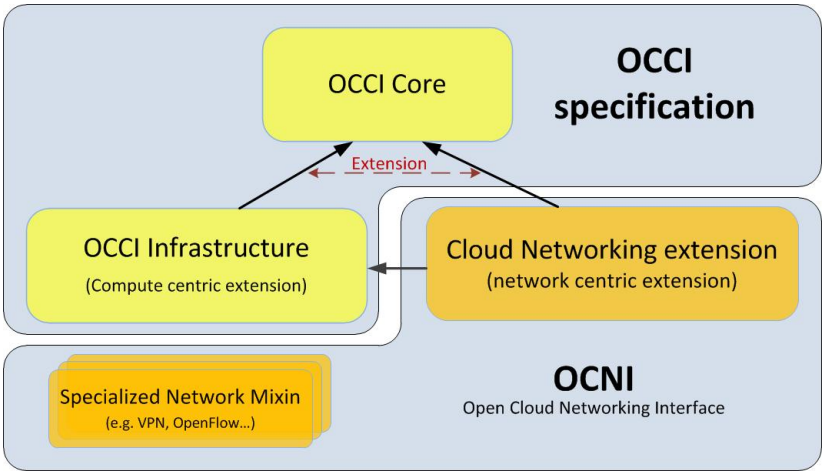


Figure 4.2: OCNI and OCCI

4.2.1 OCNI

OCNI is related to OCCI as shown in Figure 4.2. There are two categories of additions. The first is cloud networking extension to the OCCI Core model to add support for networking technologies other than VLANs alone. The second element consists of a number of specialized network *mixins*. A *mixin* allows the specialization of generic network elements with capabilities or properties. As a an example a generic network element such as link can obtain L3-VPN or OpenFlow characteristics by the use of a *mixin*.

pyOCNI: a Python Implementation of OCNI

pyOCNI [30] is the actual implementation of OCNI. pyOCNI has been developed using the Python language and uses Eventlet [31] as a concurrent networking library and ZODB3 [32] as database. As shown in Figure 4.3, pyOCNI is composed of six main blocks:

- Server: the component that can receive a request.
- Client: the component that can send a request.
- Serialization: the component that can do the information serialization.

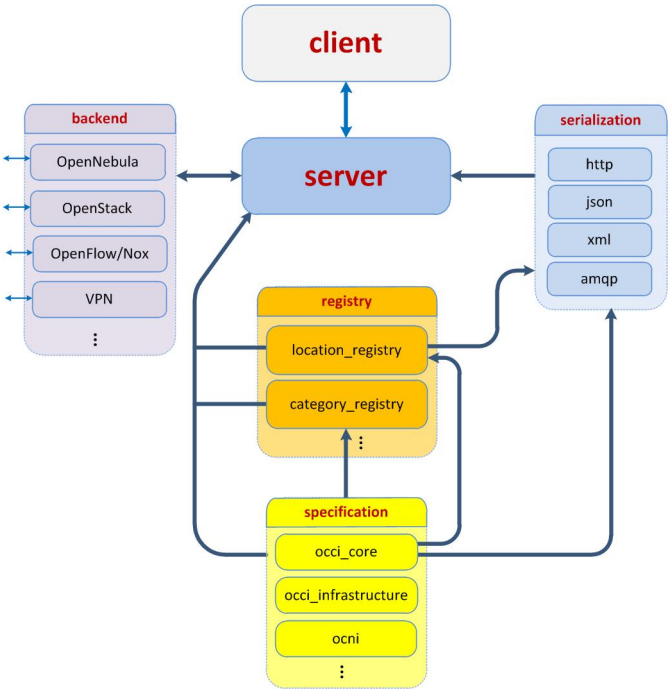


Figure 4.3: pyOCNI: a Python implementation of OCNI

- Registry: the component that store the resource information.
- Specification: the component that implement the OCCI and OCNI specification.
- Backend: the component that implement the CRUD instruction related to the controllers.

The properties of *pyOCNI* are:

- HTTP Restful providing Create, Read, Update, Delete (CRUD) operations to manage the network resources.
- Extension friendly.
- Platform independent.
- Supports multiple rendering (e.g. XML, JSON, HTTP headers etc.).
- Exposes a service model for network resources similiar to that offered by IaaS.

4.2.2 WAN Mixins

Along with the networking interfaces for traditional IaaS clouds, network interfaces are also needed to both expose and manage the virtual connectivity resources offered by network operators. This is key in creating flash network slices that span multiple domains.

To make these virtual connectivity services usable to persons with limited network technology skills it is desirable to strive for simple abstractions. Given the classes of use cases targeted by CloNe, two of the architectural choices made have been that of a logical switch for layer 2 connectivity service and a logical router for layer 3 connectivity service.

Figures 4.4 and 4.5 illustrates the model for these abstractions. On the left-hand side it can be seen that for the L2 service the WAN exposes a set of ports that semantically behaves like

a self-learning Ethernet switch with the exception that all ports do not need to have the same bandwidth and delays (due to the underlying network infrastructure and geographic distances). Readers familiar with E-LAN [33] and VPLS [34] will find many similarities with those services.

On the right-hand side the model for L3 VPN service is shown. The WAN exposes a logical routing function that has a set of targets. These targets can be a layer 2 network (inside a data centre or customer site) or may be another routing function by which routes can be exchanged using a routing protocol.

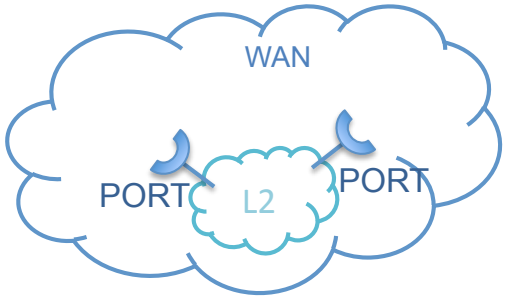


Figure 4.4: Abstraction for L2 VPN service

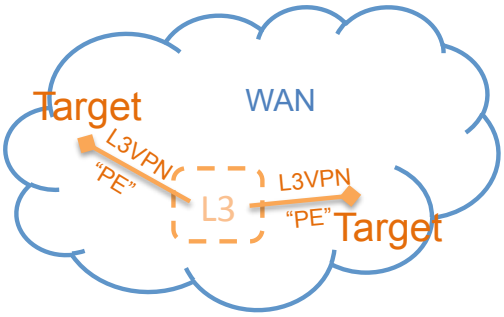


Figure 4.5: Abstraction for L3 VPN service

The following mixins (and associated connectivity services) for NO have been defined and implemented.

- L3 and L2 VPN (using IP/MPLS)
- OpenFlow

Attributes of these mixins and examples of it can be seen in Section 9.1.

4.3 Inter Domain

While VXDL along with the decomposition module provides mechanisms for requesting a distributed virtual infrastructure spread over multiple domains using hetrogeneous infrastructure, the necessary details (specifically network configuration) for connecting this virtual infrastructure among these domains is not part of this interface. This process involves the exchange of technical parameters in a domain-to-domain specific manner and some of those need to be negotiated.

CloNe's first architectural document [2], describes a category of protocols, interfaces and control operations that enable Infrastructure Service Providers to interact and exchange cross-domain administrative information. This way for exchanging control information needed administrative purposes is collectively called Distributed Control Plane (DCP). Reference resolution, notification and distributed information sharing were some interactions that were identified. During the prototyping phase of the project these aspects are further elaborated and detailed, specifically the inter-domain aspects, both at a conceptual level as well as the technical/implementation level.

This section starts by conceptually elaborating on three DCP services and its associated protocols, two new ones and one already mentioned in [2]: Cloud Message Brokering Service (CMBS) and Link Negotiation Protocol(LNP). Following sections provide further details on how these protocols are currently integrated in each administrative domain of the test bed, making a distinction between the data center domains (Section 4.3.2) and the WAN domains (Section 4.3.3).

4.3.1 Protocols

4.3.1.1 Cloud Message Brokering Service for interdomain communication

CMBS is a service for exchanging messages between the domains/providers in CloNe. Even for the inter cloud provider communication, the essential characteristics of cloud should also be satisfied. As mentioned in [35], one design pattern that suits these requirements and especially the on-demand self-service characteristic is the event-driven architecture (EDA). Considering that we are relying on an exchange of messages, the Message Oriented Middleware (MoM) is the kind of EDA that should be used. For the Message Oriented Middleware, there are three distinct layers that can be used: application layer (e.g. JMS), transport protocol wire layer with a broker (e.g. AMQP [36]) and a transport protocol wire layer without a broker (e.g. ZeroMQ). Since the main objective of CMBS concerns the information exchanged between providers and not native messaging functions (like publish/subscribe, broadcasting, etc), CMBS will be built on top of existing messaging frameworks.For prototyping we have choosen ZeroMQ [37].

Exchanging information between providers can follow several patterns and can occur in different ways like sending information to one or many providers, or sending information related to a specific topic. To satisfy requirements of this information exchange, CMBS is based on a message exchange pattern that has five layers as seen in Figure 4.6.

1. Layer 1 - Discovery space: The role of this layer is simply to exchange information about the type and characteristics of providers. Each provider who wants to be a member of the system should advertise about itself through this layer.

2. Layer 2 - Broadcast space: via this layer, a provider can send an information to all known members of the system.

3. Layer 3.1 - for provider X: via this layer, a provider can send an information to a particular provider (X).

4. Layer 3.2 - about topic Y: via this layer, a provider can send an information about a specific topic (Y) to all providers that are subscribed to this topic.

5. Layer 4 - for provider X about topic Y: via this layer, a provider can send an information about a particular topic (Y) to a specific provider (X).

Thus as shown in Figure 4.7, a decentralized message exchange framework has been defined that eases the cooperation and federation between different providers.

4.3.1.2 Link Negotiation Protocol

Link Negotiation Protocol(LNP) is used for negotiating connectivity related parameters needed to connect distributed VI. This protocol is responsible for creating and managing one or more virtual links belonging to the same VI but may be spanning multiple domains.

The design of this protocol was done with the following objectives:

- Simple and low level protocol agnostic.
- Support of various transport network solutions (L2, L3).
- Agnostic to any particular networking implementation.

The protocol uses some new terms which are defined as follows :

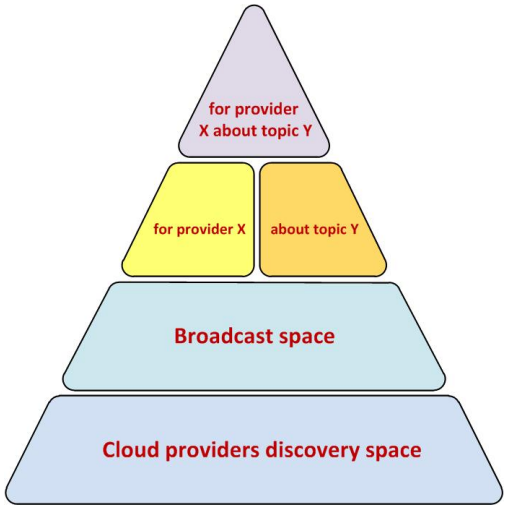


Figure 4.6: CMBS Layers

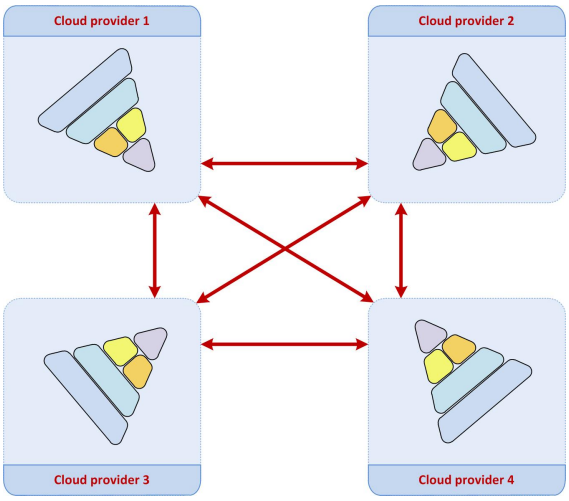


Figure 4.7: CMBS - Between providers

- **Virtual Link**: An identifier of the link between two entities as used in the VI. Only those links in the VI that spans domains are valid here. This is an end to end logical link and does not include individual links connecting different network elements together.
- **Service Provider Logical Link (SLL)**: An identifier of the logical data transmission link for sending data from one domain to another. This is usually installed and configured by the Network Operator also known as Service Provider. Each SLL has a reference to a physical or logical link, which in the latter case means that there might be aggregation happening below this layer at the physical links as well. It is important to note that SLLs are usually set up by manual negotiations and SLA. Hence in most cases no negotiation is needed for the establishment of SLLs.
- **Tenant Logical Link (TLL)**: An identifier of the actual logical (transport) link belonging to the tenant going between two domains. It has a one to one correspondence to a virtual link and creates the binding between the virtual link identifier coming from the VI description.This value is local and only valid between two domains participating in the protocol.

Creation, Updating, Deletion of TLL(s) are the three main functions that this protocol covers. There is another, the *route export* function for the case in which there is the need to explicitly export a route to a remote domain. The functions are detailed below.

Creation Function

The process of creating a TLL between two domains (referred in our description as Domain A and Domain B) is illustrated in Figure 4.8 via a message sequence diagram. Both domains have recieved an Infrastructure Service Request(ISR) that describes their individual "piece" of the whole Virtual Infrastructure. Under the natural assumption that both domains have received sub "pieces" of the same VI, whatever the nature of that VI is, the element spanning the two domains is a virtual link (one or several). For that reason we refer to "Link Request" in the diagram as the request for a virtual link that will cross and connect the VI in these two domains.

From a high-level perspective and depending on the type of link being established, (i.e. whether L2 or L3) the *link negotiation* process can have either one or two phases. For the establishment of a L2 TLL, the process comprises of only the L2 Negotiation phase. After receiving the request, one of the domains will trigger the negotiation process. The decision about which domain takes

Table 4.1: Link Negotiation: Overall message parameters

Parameters	Description
infra_id	identifier of the virtual infrastructure
msg_type	identifier of the type of message
sender	identifier of the message sender
service_type	identifier of the type of service
virtual_links	list of virtual links with the following information: virtual link id, in and out bandwidth, TLL information

the initiative is outside the scope of the protocol. (In practice it is usually the NO).In the case of multiple NOs, one policy may be that the NO with highest number initiates the process. In our example we assume Domain A to be the "initiator". After the trigger, the "initiator" starts the process by listing for each TLL, the SLLs are able to accommodate it.

The "initiator" (Domain A) sends the *Link_Offer* message the "receiver", Domain B. The parameters of the *Link_Offer* message can be seen in Table 4.1. These parameters are common to all messages in the protocol. The *infra_id* identifies the VI in question and the *virtual_link_id* identifies the virtual link crossing two domains. This information is known in advance by both domains (via the north bound interfaces). The remaining elements, *service_type*, *in_bw* and *out_bw* (in and out bandwidth) are used in the perspective of guaranteeing consistency, since this information is expected to be known in advance by both domains.

Upon receiving the list of SLLs for each TLL, Domain B selects one SLL per TLL, and sends that information in the *Link_Select* message along with the type of encapsulation scheme used for the establishment of each TLL. Domain A is then responsible for setting the encapsulation scheme configuration attributes of each TLL, sending that information to Domain B using the *Link_Config* message. Table 4.2 shows the message parameters used for this related to the TLL.At this point both domains have the necessary information to establish a L2 TLL(s).

Table 4.2: Link Negotiation: TLL parameters for layer 2 negotiation messages

Parameters	Description
TLL_id	identifier of the TLL
SLL_offer	list of SLL identifiers able to accommodate the TLL
SLL_id	identifier of the SLL selected to accomodate the TLL
encap_scheme	identifies the encapsulation scheme (type and attributes) for the TLL

For establishing a L3 TLL, the process continues and goes to the *L3 Negotiation* phase. The "initiator" sends the *L3_Cfg_Offer* message with the L3 configuration parameters, i.e. the IPs) to be configured in the endpoints of each TLL, and a list of the supported routing protocols by each TLL. The "receiver" selects the protocol and informs the "initiator" using the *L3_Cfg_Select*. The parameters for establishing a L3 TLL(s) is now known and the configurations can be enforced. The TLL parameters for the L3 Negotiation phase are presented in Table 4.4.

This process is detailed in chapter 5 according to the defined use-case scenario.

Update Function

Updating a VI or reconfiguring the parameters of its deployment in the CloNe environment are natural actions in the lifetime of a VI. Such actions can be triggered by an update of the VI, or by one of the domain's need to reconfigure its deployment (due to some management policies). In either cases if the change in VI is associated to a link, then the corresponding TLLs may need to be reconfigured. When the update to a TLL is due to an update on the VI, the domain who initiated

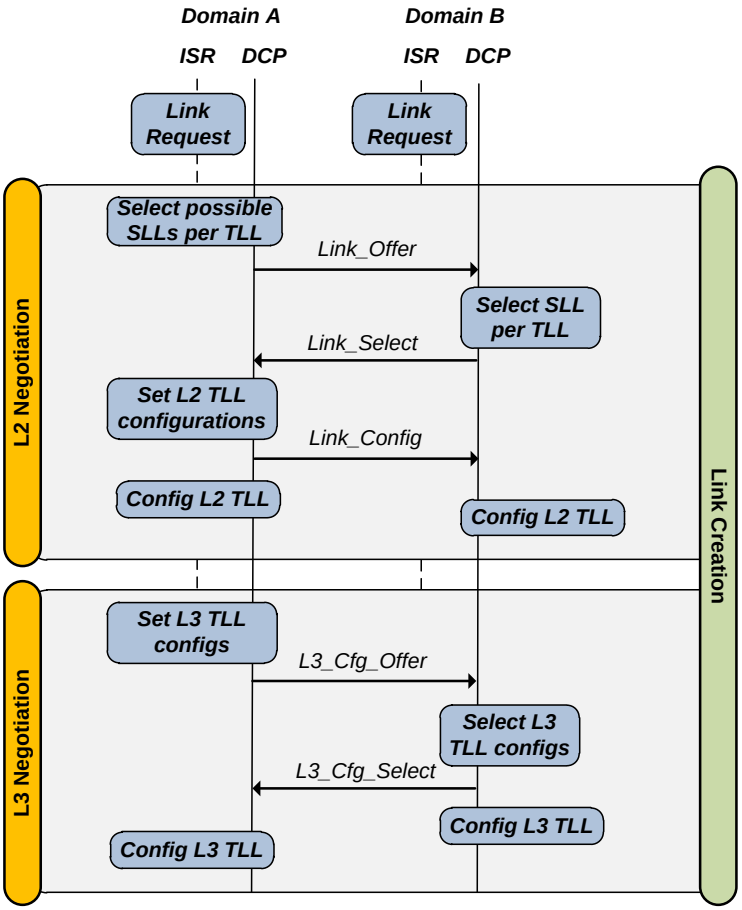


Figure 4.8: Creation Function - sequence diagram

Table 4.3: Link Negotiation: TLL parameters for layer 3 negotiation messages

Parameters	Description
TLL_id	identifier of the TLL
SLL_offer	list of SLL identifiers able to accommodate the TLL
SLL_id	identifier of the SLL selected to accommodate the TLL
encap_scheme	identifies the encapsulation scheme (type and attributes) for the TLL
L3_config	Layer 3 configuration parameters: IPs for the link endpoints (source and destination) and a list of the supported routing protocols.

the TLL creation process is also responsible for starting the update process. On the other hand, when the update is triggered by an internal domain policy either the "initiator" or the "receiver" must be able to start the process. Figure 4.9 showcases two possible scenarios, an update triggered by the "initiator" (case B) and an update triggered by the "receiver" (case A). In the former, case B, the process is initiated using the *Link_Reconf* message, which is a request for reconfiguring one or more TLLs in a certain VI. The "initiator", identical to the creation process, prepares a list of SLLs for each TLL and sends it to the "receiver" using the *Link_Update* (which in terms of parameters is equal to the *Link_Offer*). From this point on the process is similar to the creation process.

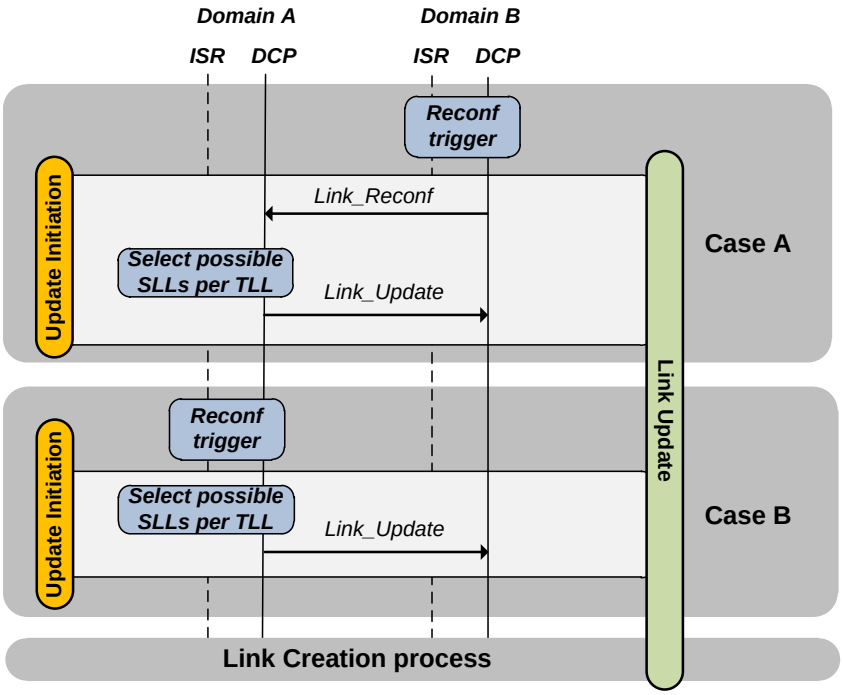


Figure 4.9: Update Function - sequence diagram

Delete Function

The delete process for a TLL, shown in Figure 4.10, is simple and is always triggered by the "initiator". Since both domains are expected to have received the delete request, the "initiator" domain usually takes the lead on this process. The *Link_Delete* message identifies the VI and the TLL(s) to be deleted.

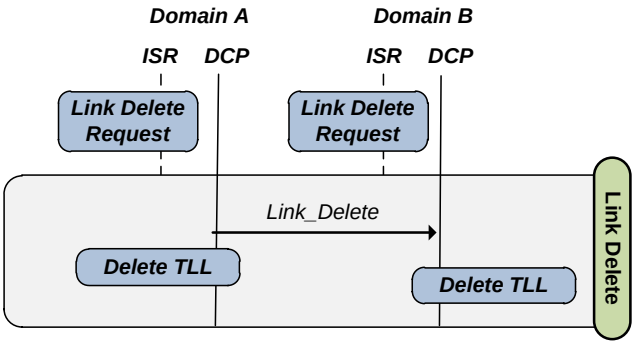


Figure 4.10: Delete Function - sequence diagram

Route Export Function

In the case of L3 TLL(s), for some domains, the defined routing protocol maybe static. For such cases, the route export function was defined to cover the need of exporting a route to another domain. To do that the domain wanting to export routes related to a certain VI uses the *Route_Export* message to send one or more routes to a remote domain. Figure 4.11 illustrates the process for the case in which Domain B is exporting a route to Domain A.

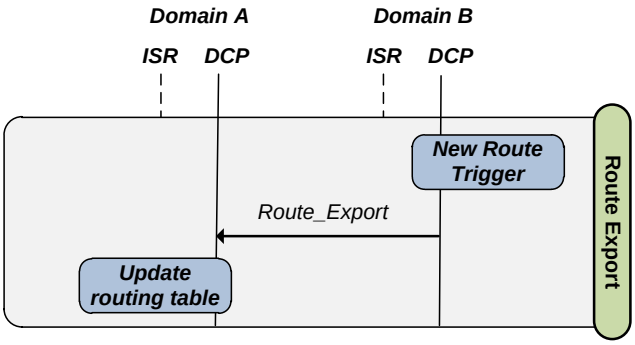


Figure 4.11: Route Export Function - sequence diagram

4.3.2 Data Center Components

This section describes the Data Centre components that were developed or modified to support inter domain interactions, specifically the Link Negotiation Protocol.

4.3.2.1 Link Negotiation Protocol Integration

The integration of the LNP within the test bed's Data Centre domains is detailed in this section. Each one of the four Data Centre domains that integrates the test bed is managed differently, however they all possess identical management entities. These generic entities are entitled as: Compute Manager, DC Network Manager, and the DCP controller. Figure 4.12 shows the various components involved in the link negotiation and setup process in a Data Centre.

- **DC Network Manager** - module responsible for creating and managing tenant's virtual networks. The DC Network Manager module together with the DCP module implements the functionality for link negotiation (i.e set up of TLL). This requires the set up of an intermediate network device for bridging the tenant's virtual network to the WAN. Also an encapsulation device needs to be configured as per the SLL used for the interconnection. The DC Network Manager module implements the functionality for creating the interconnecting network device. In the case of L3 VPNs this means a router for routing between the two subnets. This functionality may be implemented in software (running inside a VM or a routing service) or by delegating to and configuring a hardware based router. In the case of L2 VPNs this means a layer 2 device, namely a switch or even just a link. Here also the implemenation may be software based or delegated to an appropriate hardware device. Note that in both cases the netdevices need to expose two (virtual) interfaces. One of these virtual interfaces are then plugged into the bridge corresponding to the tenant network.

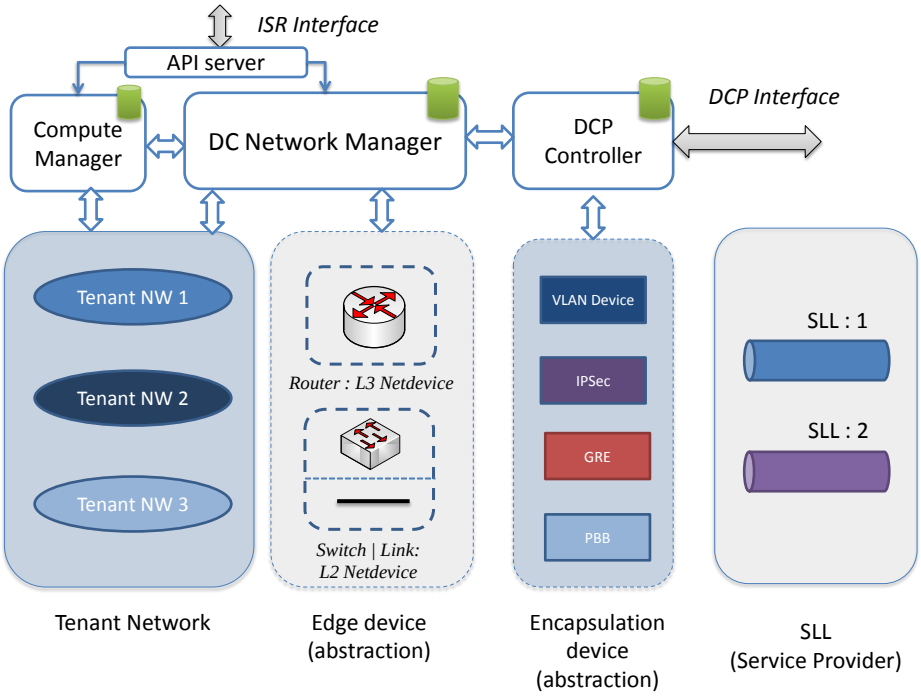


Figure 4.12: Components involved in the link negotiation and setup process in a DC domain

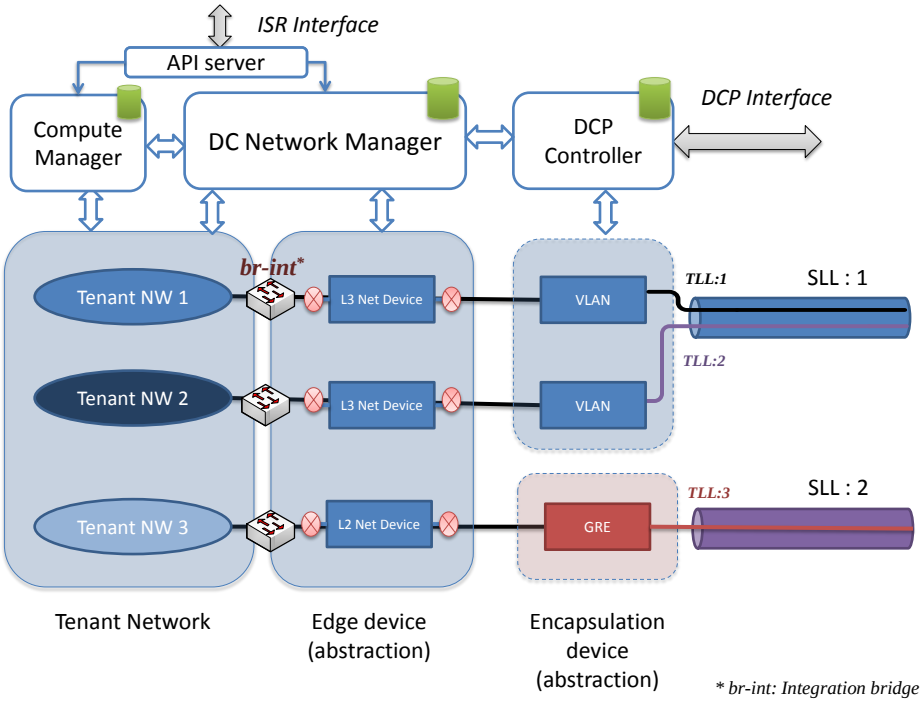


Figure 4.13: Runtime snapshot with some configured links and associated internal configuration

- **Compute Manager** - module responsible for creating and controlling VMs for the tenant and places them in the corresponding tenant network by interacting with the DC Network Manager module. Each has an associated database for keeping their configuration information.

- **DCP Controller** - module configures the encapsulation device for the SLL chosen for this link. This device provides the necessary encapsulation needed for separating/tunnelling the traffic in that link. Examples can include VLANs, tunnelling schemes like GRE or IPSec etc. Note that the encapsulation type for an SLL has already been pre-agreed and only the parameters of that scheme are negotiated and set here. Here also from an implementation perspective, both pure software only or a hardware accelerated setup are equally feasible.

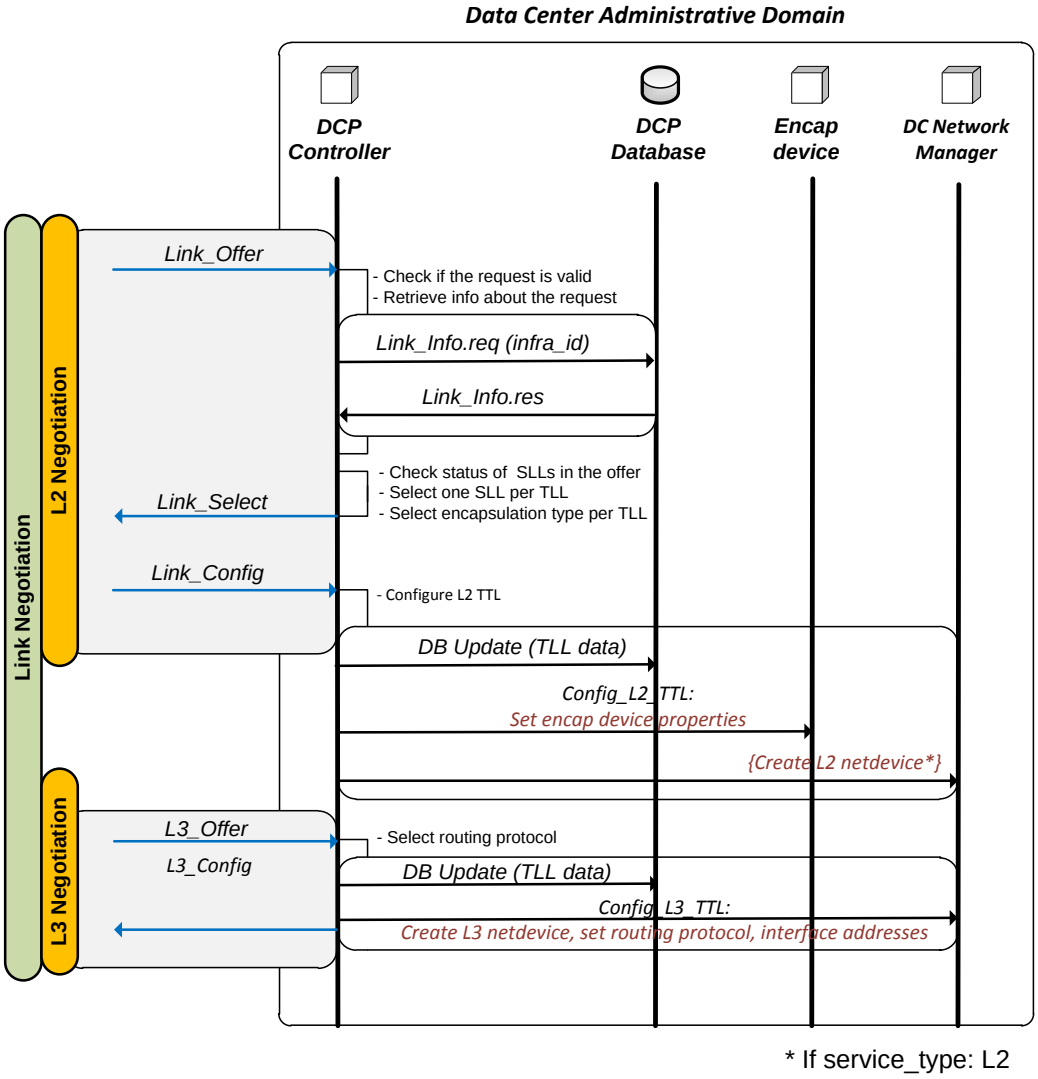


Figure 4.14: Link setup - DC network functions interactions

A runtime snapshot with some already configured links and their associated internal configuration is shown in Figure 4.13.

Due to the similarity in the management architecture behind each DC domain, the integration of the LNP is identical in the different domains. At this point in time the LNP creation function of the protocol is the only one implemented. Figure 4.14 shows the interaction among the modules based on the messages.

Table 4.4: Data Center generic entities mapping

DC/Domain	Compute Manager	DC Network Manager	DCP Controller
EAB	OpenStack Nova	OpenStack Quantum	Own module
PTIN	OpenNebula	OpenNebula	Own module
IT	OpenNebula	NOX	Own module
HP	Cell-as-a-Service	VNET	Own module

4.3.3 WAN Components

4.3.3.1 Integration with MPLS Networks

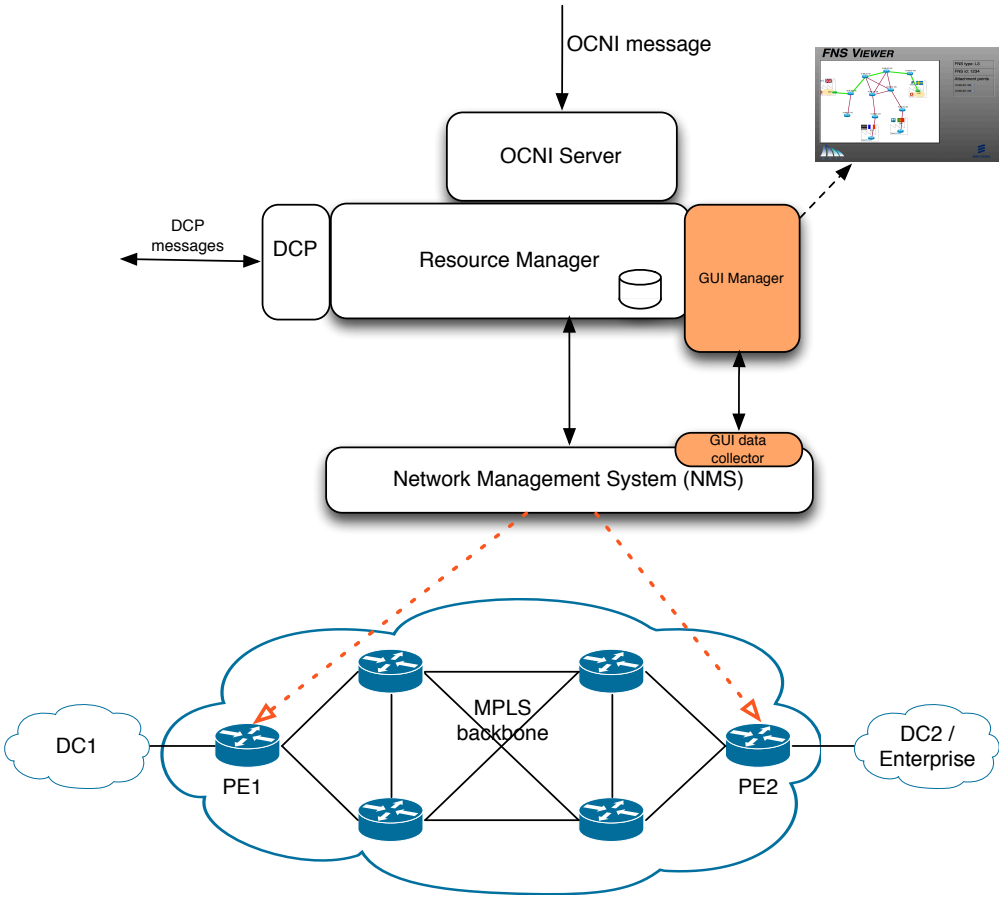


Figure 4.15: Components involved in FNS management in IP/MPLS NO domain

The integration of the Link Negotiation Protocol with the MPLS based Network Operator (NO) in the test bed is detailed in this section. There are two MPLS domains at Ericsson and PTIN premises. The basic generic components involved in both domains are shown in Figure 4.15.

The Resource Manager receives the OCNI request for a VPN link (L3 or L2). Example of this OCNI message is in Section 9.1. The Resource Manager checks whether the request is valid and whether the link can provisioned with the requested attributes. If so, the OCNI message is acknowledged and the LNP is initiated with the two end points as specified in the request, by means of the *Link_offer* message. The LNP depends on the type of VPN that is requested i.e. L2 or L3.

After the LNP is completed the Resource Manager has all the parameters to create the VPN. The

Resource Manager does not directly create the VPN, but delegates it to the Network Management System (NMS) used to control the network elements of the corresponding domain. In the case of PTIN, this is a commercial product used to control the physical network equipments. For Ericsson, a set of scripts was used to control the virtualized network nodes, emulating a basic NMS. The NMS contacts all the edge nodes and creates the L2 or L3 VPN.

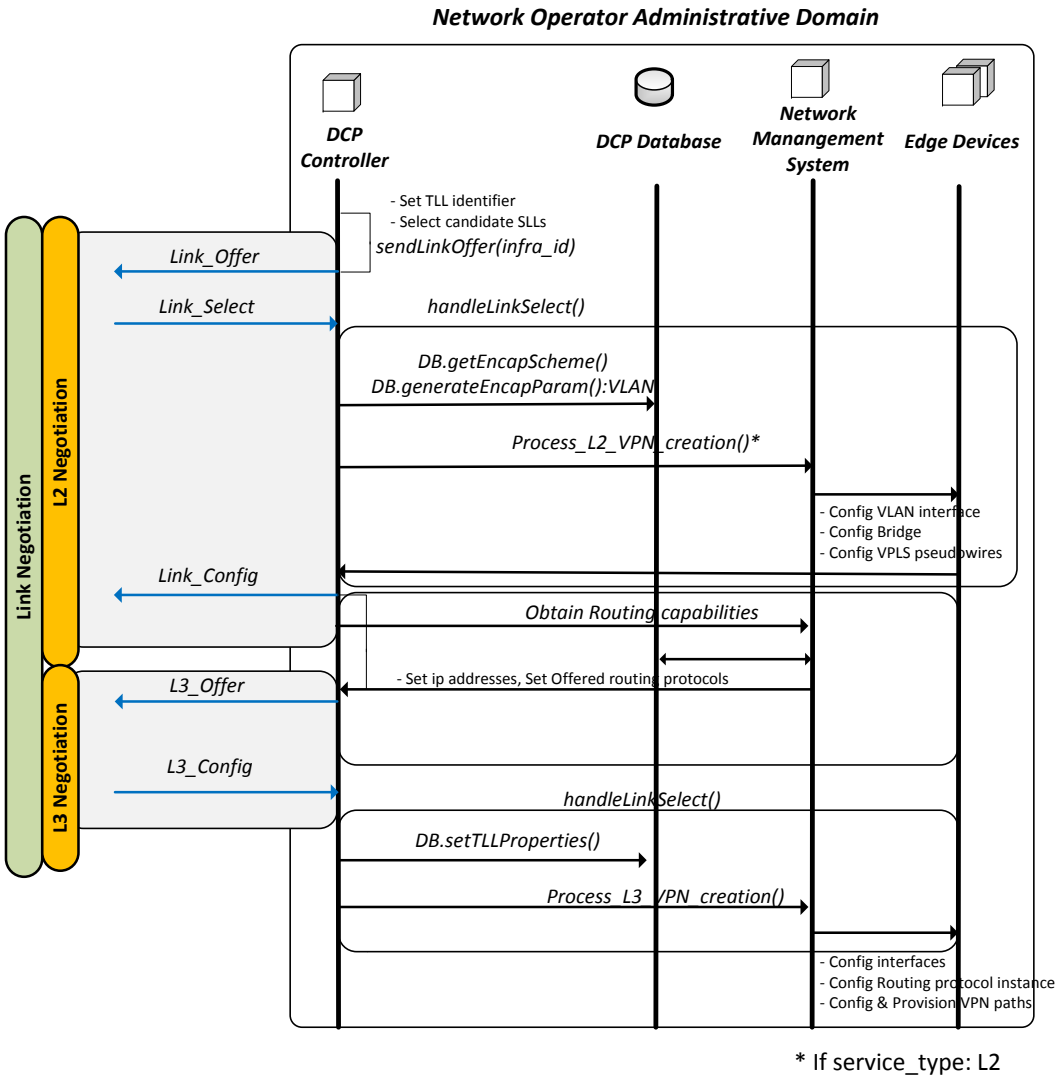


Figure 4.16: Link setup - NO network functions interactions

Figure 4.16 shows the LNP exchanges and the actions taking place within the network operator domain. Not shown in the figure is the OCNI request that has proceeded these steps as discussed above. The DCP controller in the NO begins by selecting a set of candidate SLLs that are sent to the DC in a *Link_Offer* message. Once the DC has selected one and reported that choice in the *Link_Select* message the DCP controller retrieves the encapsulations scheme for that SLL and obtains or generates necessary encapsulation parameters. In the case of the testbed network VLANs have been used for tenant isolation so the parameter generated is a VLAN tag.

If a L2 FNS has been requested in the previous OCNI request, the DCP controller instructs

the NMS to create a L2VPN and, when completed, sends back a *Link_Config* message to the DC informing it about the assigned VLAN. The LNP transaction sequence is then complete.

If instead a L3 FNS has been requested, the DCP controller instead interrogates the NMS about the routing capabilities the involved NO edge devices can support along with information about IP subnet to use between the DC and NO edge devices. This information is then sent to the DC in a *L3_Offer* message. Once the DC has made its choice of routing protocol to use and reported that in a *L3_Config* message back to the DCP controller the L3 VPN is instantiated with the help of the NMS. That marks the end of the LNP transaction sequence for the L3 FNS case.

4.3.3.2 Integration with Openflow Networks

The possibility of integrating the OCNI and the DCP with an OpenFlow controller has been explored. To this avail, the OpenFlow GUI implemented within the scope of WP-C was extended. Figure 4.17 shows the relationship between the components and the logic behind a path activation using the DCP.

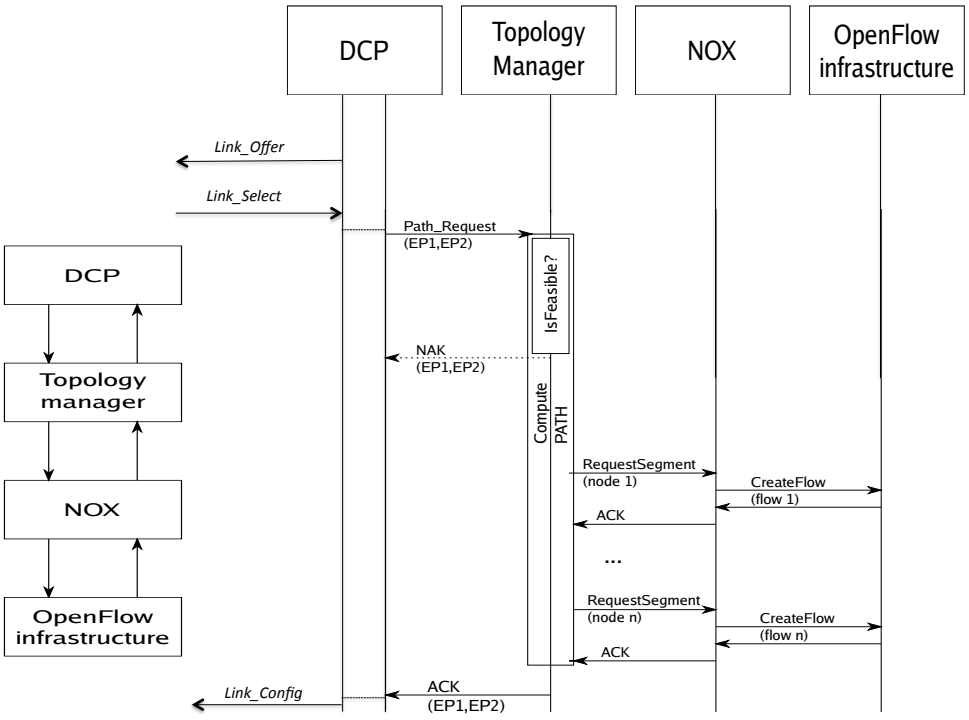


Figure 4.17: Integration of NOX OpenFlow Controller with DCP - Establishing a network path

The path activation message delivered by the DCP defines the two end points in the transit OpenFlow based network that need to be connected. This interconnection will be static and available until it is explicitly torn down by a request coming from the DCP. EP1 and EP2 are the end points of the connection. They are characterised by:

1. Identifiers of the provider edge routers: IP address of the PE router.
2. Physical and logical port identifiers (e.g., Ethernet interface with 802.1q and VLAN identifier).

The path request is sent to the Topology Manager where the feasibility of the path request is checked. Feasibility means that a path can be established between both provider edge nodes that is composed of OpenFlow in the different network nodes along the path that do not collide with any previously established (permanent and dynamic) OpenFlow rules. If the requested path is not feasible, a *NACK* is returned to the DCP.

If a path is feasible, the Topology Manager installs the computed OpenFlow rules in the NOX controller and the OpenFlow nodes. Once the full set of rules is installed, the Topology Manager confirms that the path has been established by sending an *ACK* message to the DCP.

A similar approach was followed in the integration of OpenFlow in the OCNI implementation. Listing 9.2 shows a path request message sent to the OCNI interface of the network.

4.3.3.3 Relation to WP-C

Integration between Work Package (WP)s WP-C and WP-D concepts is described in [38] from the point of view of WP-C.The natural platform to check the feasibility of different proposals was the OpenFlow data centre GUI developed within the context of WP-C initially.

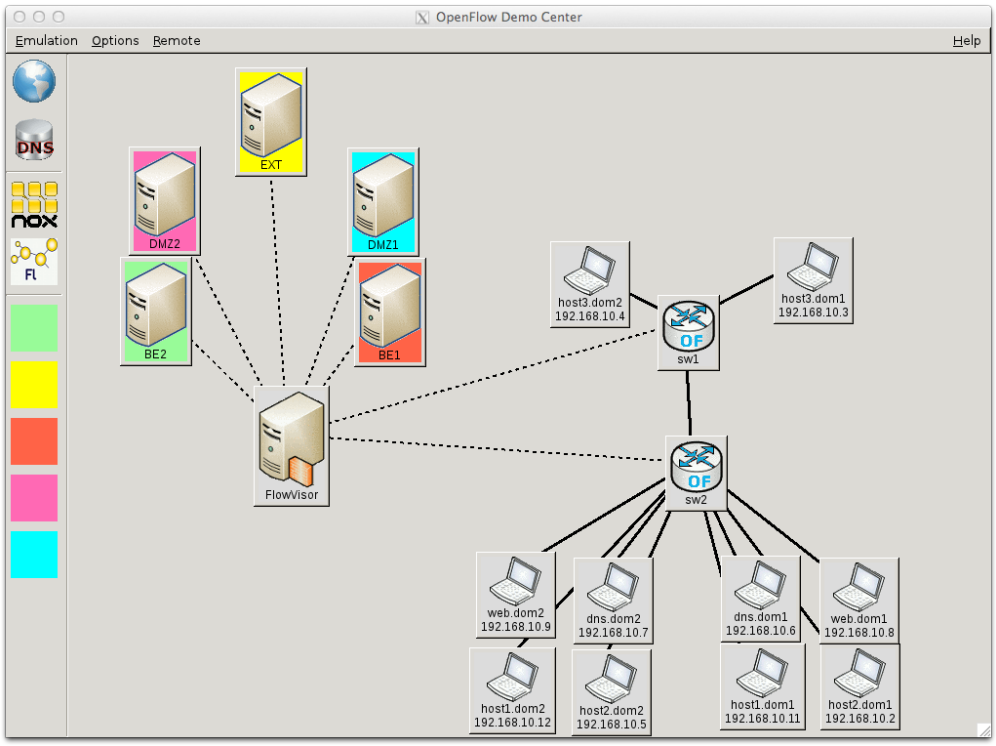


Figure 4.18: OpenFlow-in-a-box GUI

Figure 4.18 shows a screen capture of the Graphical User Interface (GUI). This GUI provides access to an OpenFlow-in-a-Box environment. The GUI can interact with an internal OpenFlow environment based on Mininet and an external OpenFlow environment. This environment was initially designed to model the behaviour of a data centre with OpenFlow to provide network functionalities like VPN isolation, firewall secured network realms, etc. By adding the possibility of switching user traffic between nodes in the OpenFlow network, the facility is now able to act like a Network Operator with distributed compute and storage capacities in addition to traditional networking functionalities.

4.4 Data Centre Networking

4.4.1 VNET

Within CloNe prototyping a solution for datacentre networking called VNET [5] has been developed. VNET can create separate layer 3 networks on top of a flat layer 2 infrastructure without the need

for added routers. This is done through changes in the hypervisor of the host machines as described below.

The data center part of HP uses a flat L2 network with hosts equipped with Linux 2.6.32 kernel, KVM hypervisor and OpenStack (Diablo). The prototype consists of a Python implementation of the OCCI. Typically, the OCCI client connects to the OCCI server located in the controller, the physical host where the main components of OpenStack's Nova resides for creating resources. In addition, it maps each network to specific subnet in the flat network space created. OpenStack's Nova has been modified to send information about events to the VNET module part of the network manager, in particular, the address allocation performed. Nevertheless, the control plane is fully managed by VNET. When OCCI server receives a request to create a network, it notifies the VNET drivers (in all hosts). The list of all subnets created is maintained by the cloud controller. Each host enforces Iptables/Netfilter [39] rules to achieve isolation between subnets. In addition, the controller configures the VNET module by reading/writing specific filesystem (SYSFS).

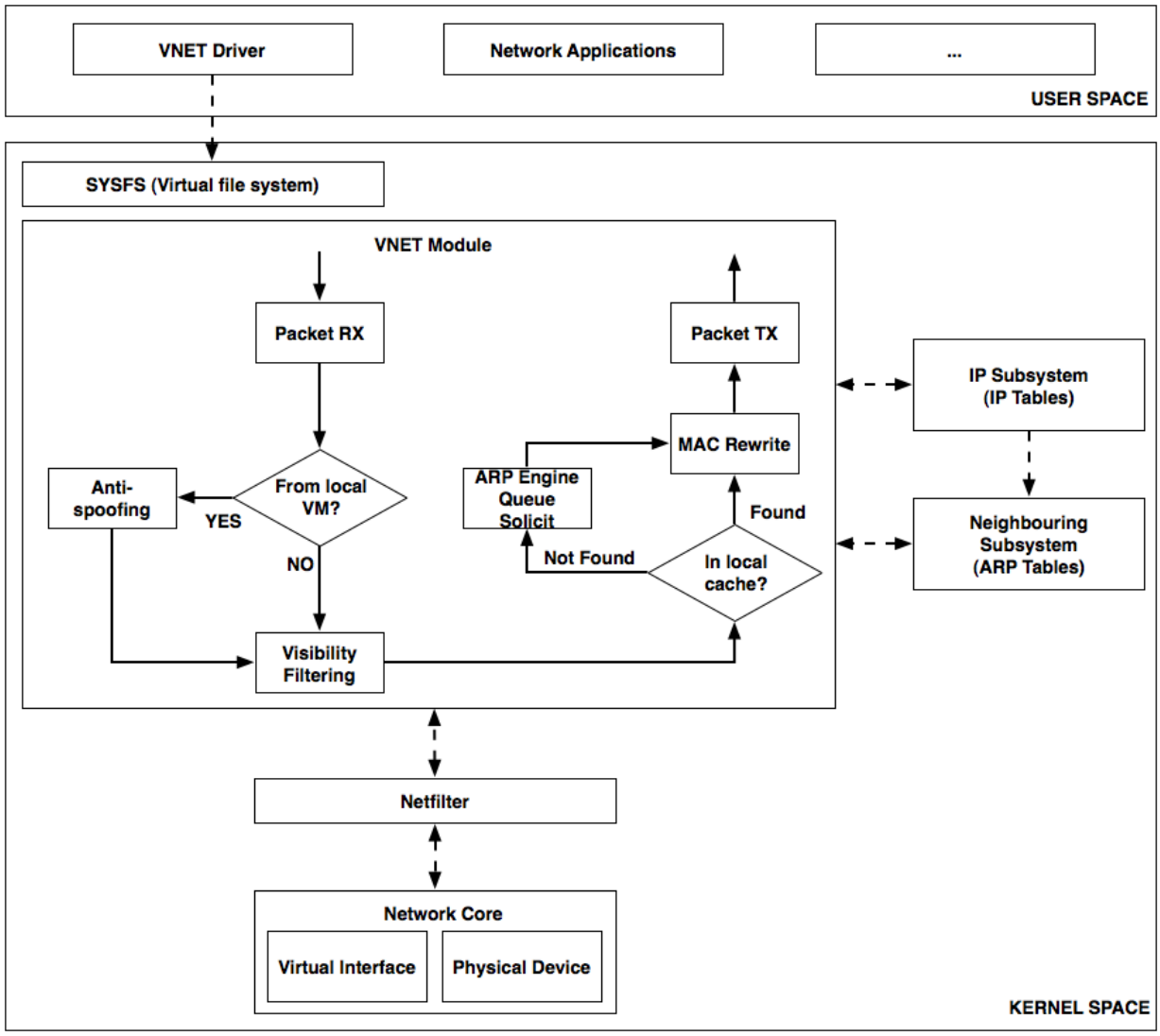


Figure 4.19: VNET interactions

Early VNET [5] work has been adapted to work alongside with OpenStack. Now, when a VM is created, OpenStack Nova modified with VNET passes the IP address of the physical machine

to the VNET driver. Although OpenStack allows subnetting through the use of VLANs, it may not scale well for cloud scale data centres. Hence, a mechanism has been added to guarantee that the IP assigned to a VM falls within the range specified by the user's request based on a given flat network. VNET generally intercepts and processes the packets from VMs or the host OS that reach the wire. It discards all the packets that violate filtering rules. Otherwise, it transmits them to another physical host or forwards them to the local VM or the host OS. Any packet arriving on the physical or virtual interface on the system needs to be processed by VNET. In general, local packets are anti-spoofing checked while all packets are filtered by the rules. VNET maps the IP addresses to either local virtual interfaces or physical MAC addresses of remote hosts. Packets for local VMs are determined by VNET for MAC rewriting while the ones for remote VMs are rewritten with the physical MAC addresses of the remote hosts. If no entry exists in the mapping table, the packets are queued until the mapping can be determined.Our VNET driver is generally pre-configured with the gateway of the OpenStack flat DHCP network so that the connectivity is initially guaranteed between any subnet in the data centre.

4.5 Management Components

A number of management concepts and functions have been developed in CloNe. Two of these have been prototyped. The first is a network virtualization library called LibNetVirt. LibNetVirt allows the management of network resources in uniform manner across a number of networking and virtualization technologies. The second is a generic resource management system for compute resources in a domain that can be instantiated to realize a number of management objectives.

4.5.1 Network Resource Management

The abstraction of underlying technologies is a common approach in modern systems. This abstraction offers flexibility of decoupling operations from the physical equipment.

Unlike machine virtualization, which has been around for some time and fairly well known, virtualizing networks is a different and more complex challenge. Networks are still coupled with the physical infrastructure [40]. Network virtualization is a mechanism for sharing a physical network. Virtual networks have been around for several years but nowadays they are one of the main focuses in the networking community. There are different technologies and approaches that can be used to share a physical network or substrate, such as, VLANs, Virtual Private Networks, virtual routers and programmable virtual routers.

Most of these technologies mainly need to be manually configured, which makes the adoption to dynamic environments quite impractical. A unified framework for network virtualization is necessary to expand network functionalities and to provide a single network view. A unified framework will help to provide a common interface for multiple technologies, providing a single, unified and consistent view towards the management layer or even end users. A single network view facilities this abstraction due to the fact that it hides the unnecessary complex details of the network . Thus the aim of this component is to provide a programmatic abstraction of network resources in a similar way to how compute resources are virtualized nowadays.

An abstraction library called LibNetVirt [6] was thus created and it uses single router abstraction [41] to describe a network. The description includes the endpoints of the network, the constraints of specific paths as well as type of forwarding required for the underlying network. LibNetVirt is composed of a common interface and a set of drivers (See Figure 4.20). Each driver implements the required configuration for a specific underlying technology. Both an OpenFlow driver and a L3 VPN driver are implemented as a proof of concept of this solution. LibNetVirt has been released as open-source and can be found at `http://danieltt.github.com/libNetVirt`.

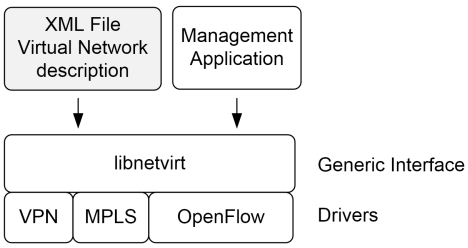


Figure 4.20: LibNetVirt architecture

LibNetVirt offers two APIs in C and Python to operate the network. LibNetVirt is used in different ways inside the test bed:

1. Integrated with the network manager of a data centre managed with OpenFlow. It allows the creation of different virtual networks between endpoints.

2. Directly integrated inside the network resource management system of the Network Operator to set up a MPLS network. When the network domain receives the request it calls LibNetVirt with the python API.

3. LibNetVirt has been integrated with pyOCNI and a specialized OCNI mixin for OpenFlow has been defined and implemented.

4. Set up the network automatically from a saved network description. LibNetVirt uses an XML format to describe the network.

OpenFlow driver uses a NOX controller which is used to create L2 network on demand. Only the endpoints are required to set up the network. More details of the interface and the OpenFlow driver can be found in [6]. The OpenFlow driver isolates traffic from different tenants and it has link fault recovery as well. When the controller detects that a link or switch is down, it automatically reroutes all the traffic to an alternative path. It allows also adding and removing endpoints on demand and without any disruption of the other traffic that is transiting the network.

MPLS driver is a set of scripts that send commands to the involved routers to set up the MPLS network. It configures the different interfaces as well as the protocols, such as OSPF and BGP, involved in the control plane of the WAN network.

4.5.1.1 LibNetVirt Integration

MPLS driver sits at the management node of the WAN network and allows creating and removing VPNs using the OCNI interface with its own mixin or loading the description from an XML file.

An OpenFlow network with 4 switches has been connected to one of the Provider Edge (PE) routers of the WAN network, using a GRE tunnel. See Figure 4.21 for more details. The OpenFlow square represents an OpenFlow domain, which can be either a data centre or an extension of a WAN. Using different VLAN tags in one of the endpoints to distinguish between tenants, it is possible to create and remove virtual networks on demand using LibNetVirt. Together with the LibNetVirt module in the WAN network, it is possible to establish end to end connectivity using only LibNetVirt.

4.5.1.2 Differences between LibNetVirt and pyOCNI

LibNetVirt and pyOCNI are complementary components that can be used in the CloNe infrastructure. LibNetVirt is a library that uses the single router abstraction to implement a network. It

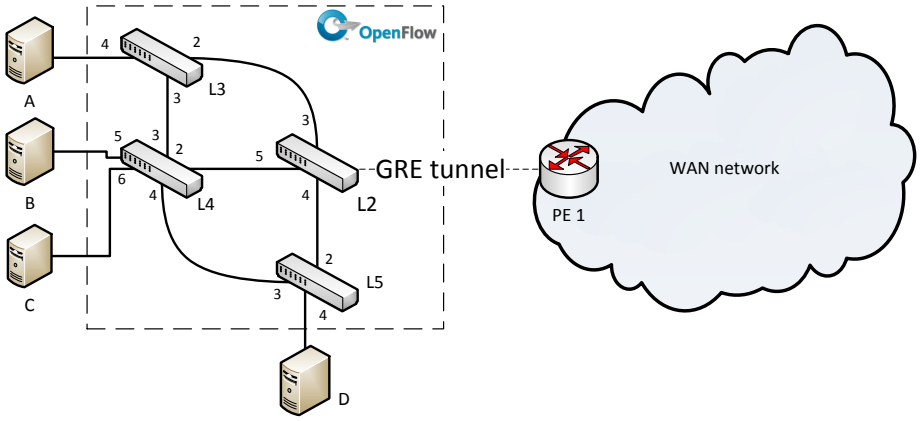


Figure 4.21: OpenFlow test bed for LibNetVirt

is composed by a set of drivers. Each driver implements the required configuration for a specific underlying technology. LibNetVirt aims to reduce the time for the creation and termination of virtual networks. A common set of calls is defined and used to instantiate different virtual networks, in a programmatic and on demand fashion. Only the definition of the endpoints is necessary to create the network.

PyOCNI (Python Open Cloud Networking Interface) is a Python implementation of an extended OCCI with a JSON serialization and a cloud networking extension. It is a RESTful interface towards the end-users.

The main differences between LibNetVirt and pyOCNI are:

- LibNetVirt is focused towards the administrator, while pyOCNI is focused on final user. In LibNetVirt some of the details for the endpoints are necessary to introduce, for instance, the physical switch and port where the endpoints are. On the other hand, in pyOCNI the user does not need to know the internal details of the endpoints and only provides a reference.

- PyOCNI can extend its functionalities adding more mixins. Currently there is one mixin and one back-end that use LibNetVirt for L2 networks. If the information provided by the user is not precise, it is necessary to have a middle application that converts this information to the proper parameters. For instance, if only an identifier of an endpoint is provided by the user, it will be necessary to obtain the switch identifier and port to pass as input to LibNetVirt.

We can see that both are complementary. LibNetVirt is designed as a library in order to be reused by different applications, with pyOCNI as one of them. LibNetVirt also offers an example application, which allows controlling a networks form a command line interface (CLI). LibNetVirt is an API and library, while pyOCNI is an interface.

4.5.2 Compute Resource Management

Cloud service providers define strategies according to which resources for computation, storage and networking of the cloud infrastructure are allocated to the customers' applications. Such strategies are expressed as *management objectives* for the cloud resource management system. Examples of management objectives include *balanced load* across the cloud devices, *minimal energy consumption* of the cloud infrastructure and support for *service differentiation* among different classes of cloud services.

D.D.1[2] identifies that resource management in a cloud network must include two controller functions: a *resource allocation* function (concerned with initial resource allocation) and a *resource adaptation/optimization* function (concerned with adapting an existing resource allocation). The SAIL prototype includes implementations of these two functions for 'balanced load' and 'minimal energy consumption' management objectives. The implementation is based on the OpenStack cloud platform and it currently supports management of compute resources.

4.5.2.1 Resource Allocation

The resource allocation function that is implemented in CloNe is based on the OpenStack Least Cost Scheduler (LCS) [42]. This scheduler is generic in the sense that it has two sets of abstract functions that need to be instantiated in order for it to be functional. First, an instance of the LCS should specify a set of *cost functions* and, for each function, an associated weighting factor. Second, the instance should specify a set of *filters*. When a request to start a VM is received by OpenStack, the active instance of the LCS schedules the VM for execution on a server selected as follows. First, it applies the filter function to the set of servers and selects a subset that is suitable to run the VM. Next, for each server in the subset, it computes the cost of running the VM as the weighted sum of the cost functions. Then, out of all the suitable servers, it selects the server that has the minimum cost. Figure 4.22 shows the flowchart of the LCS.

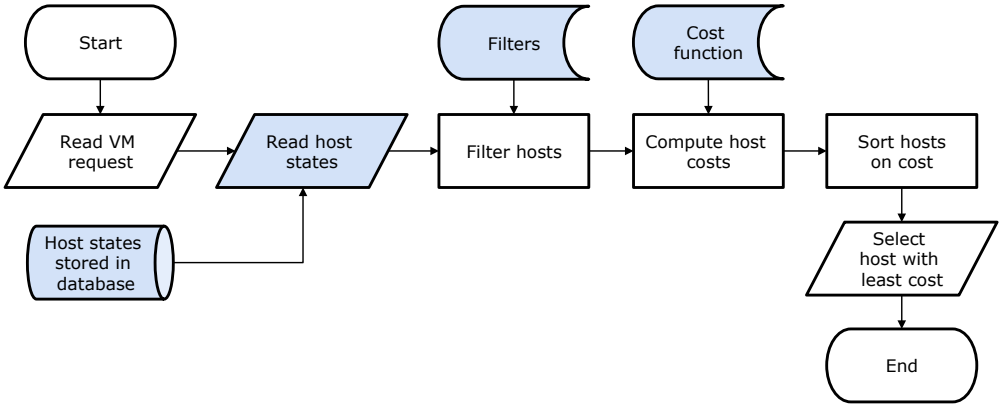


Figure 4.22: Flowchart of the OpenStack least-cost scheduler.

The prototype implementation that relates to resource allocation extends OpenStack in two ways. First, it extends the compute service of OpenStack such that up to date information regarding resource utilization is made available for the scheduler. This allows the cloud manager to define management objectives based on the actual resource utilization in the servers. Such objectives have the potential to result in an efficient use of the available resource in the datacentre. This extension is realized by (1) expanding host-information tables to include columns relating to resource utilization and (2) by extending the state-reporting loop of the compute service such that utilization figures are also reported periodically into the above tables.

Second, we develop a set of cost and filter functions that realize different variants of the load-balancing and energy-efficiency objectives. Specifically, two variants for each management objective, one based on the resource allocation (i.e., the aggregate resource capacities of VMs) and another one based on the actual resource utilization on the servers, are implemented.

4.5.2.2 Resource Adaptation/Optimization

OpenStack (and most other open source cloud management frameworks) lack the capability to dynamically optimize an existing allocation. Today's cloud environments are dynamic in the sense that

there is a continuous flow of requests to start and stop virtual infrastructures that are provisioned in the cloud. In addition to this, the resource demand of applications that run on these virtual infrastructures are often time-dependent. This means that the allocation of physical resources to virtual infrastructure needs to adapt all the time.

The prototype implements this function through a gossip-based protocol called GRMP that is listed in Figure 4.23. The details of the protocol were reported in Section 5.3.4 of D.D.1 [2]. The protocol is implemented in a multi-threaded python process that runs on all the compute servers in the domain. During the execution of the protocol, a server selects at random another server and they both exchange state information that includes the resource demands of the VMs that they run, their resource capacities as well as their resource utilizations. The servers then decide which VMs to move from one server to another in order to realize a specific management objective.

```
initialization
1: initInstance()
2: start passive and active threads;
```

```
active thread
1: while true do
2:   read current state A_n
3:   n' =choosePeer()
4:   send(n', A_n); A_n' =receive(n')
5:   updateState(n', A_n')
6:   execute A_n
7:   sleep until end of round
8: end while
```

```
passive thread
1: while true do
2:   read current state A_n
3:   A_n' =receive(n'); send(n', A_n)
4:   updateState(n', A_n')
5:   execute A_n
6: end while
```

Figure 4.23: Protocol GRMP runs on each compute server n

GRMP is a generic protocol in a similar sense to OpenStack's LCS. Specifically, in order for it to realize a specific management objective, three abstract methods, `initInstance`, `choosePeer` and `updateState`, have to be instantiated. `initInstance` implements the specific initialization routines for the instance, which typically includes reading in some state information from the server. `choosePeer` returns the server with which the current server exchanges state information and possibly VMs. `updateState` decides, given the current states of the two servers and the management objective, which VMs to move between the two servers.

Two instantiations of the GRMP protocol that realize the load-balancing objective and the energy-efficiency objective have been implemented for the CloNe prototype. The details of the instantiation for the load-balancing objective is available in [43] while for the energy efficiency objective in [44]. We have performed a performance evaluation of the instantiations on a small testbed and documented the results in [45]. To mention two of the key findings of the evaluation, first, the resource allocation is within some 5% of the optimal for both management objectives under no-churn and low-churn scenarios. Second, the performance of the resource management system degrades with increasing churn rate, suggesting that dynamic adaptation is cost-effective only upto a certain churn rate.

4.6 Security Components

The security architecture [2] is designed to manage the current and future security challenges which may affect the CloNe infrastructure. An initial list of security challenges was based upon a well-defined and comprehensive security analysis covered in [46]. Subsequently, design of the CloNe security architecture and prototyping of security modules has further modified and concretized the list of security challenges that might affect the CloNe infrastructure. These security challenges in turn lead to the formulation of security requirements. The most important security requirements include Identity management and misuse protection [46].

An access control policy module forms the backbone for an identity management module. It is responsible for determining whether a user is permitted to access a particular resource. An authorisation logic has been developed as a part of the CloNe security architecture which is responsible for enabling fine-grained access control of virtual resources. Furthermore, a Trusted Platform Module (TPM) based auditing module has been developed to attest the geographic location of the physical resources provisioned to the cloud user. The TPM based module prevents misuse of resources by certifying their geographic location by using a Certification Authority (CA).

4.6.1 Delegation Aware Authorisation Service

Authorisation is a security function that determines whether a user is permitted to access a specific resource. The resource owner defines an access policy that specifies which users can access the resource. Based on the access policy, user access on a resource is validated. There are two common approaches for specifying access policies, namely, Access Control List (ACL) (simply listing the authorised users) and Role Based Access Control (RBAC) (roles are authorised and users are assigned roles).

The integration of the authorisation logic into the CloNe infrastructure is further complicated by the delegation approach and the information hiding requirements. A user has the right to specify access policies for a resource and may delegate this privilege to another user. Further, the provider also has the right to delegate implementation of a resource to another provider. Users and providers are not required to disclose their delegations to each other.

A practical example of this scenario is shown in Figure 4.24, where a provider has delegated a virtual infrastructure amongst two providers. Each provider further re-delegates their corresponding part. In order for two providers to connect their infrastructures, they need to establish a link between their respective access points. Clearly, the original provider does not know details of the ultimate access points. Furthermore, the original provider also does not know the linking provider's identity while establishing the link.

4.6.1.1 Authorisation Logic

The authorisation of a user to access a resource follows the chain created by successive virtual infrastructure delegations and access right delegations. One end identifies the user that is authorised to access the resource and the other end identifies the resource itself. The links in this chain represent the individual acts of delegation and are only known to entities involved in the delegation. Multiple delegation chains may exist in any infrastructure and may be initiated by any entity. Hence, a single root of authority is not required.

We have used an authorisation logic [47] that is capable of encoding individual delegations as grants and using them to prove authorisation. In the authorisation logic, *issuers* are responsible for specifying grants. The grants relevant to our delegation are of two types: authorisations and name definitions.

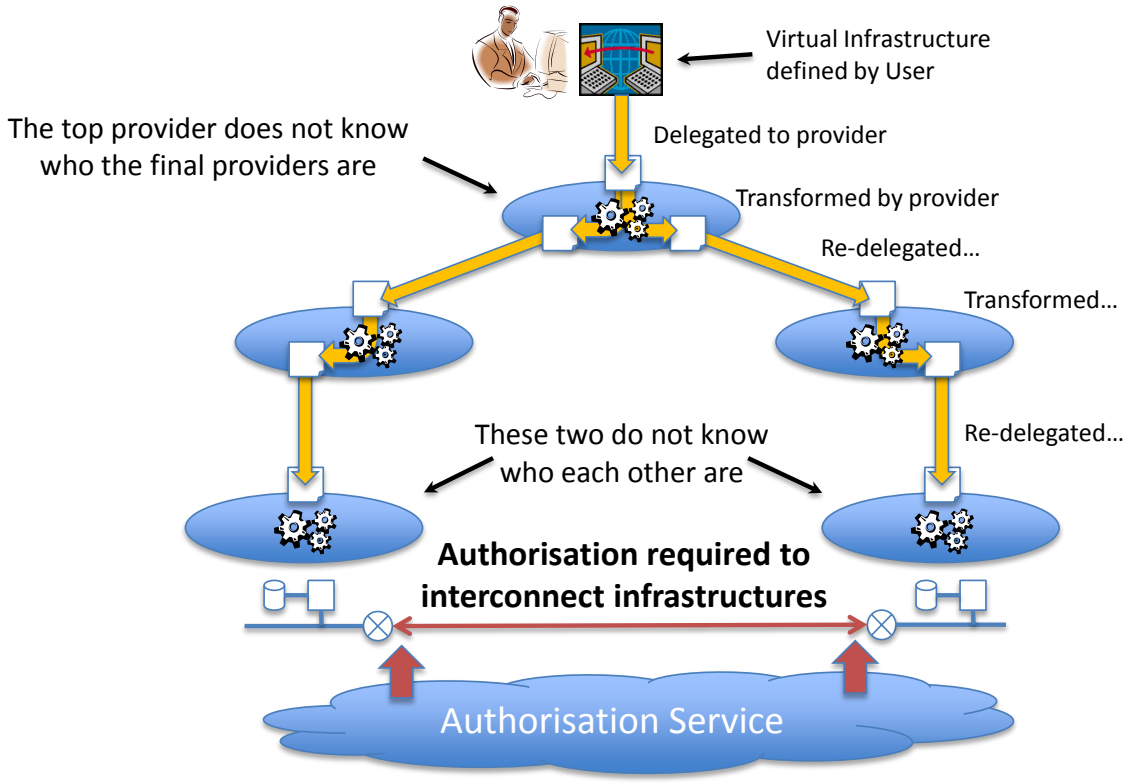


Figure 4.24: Authorisation with delegation

An authorisation grant specifies that, according to the issuer, a particular user or name is permitted to perform a given operation on a specified resource. For example:

HP grants : $user5@aisec.fraunhofer.de$ link access to $//hp.com/accesspoints/accesspoint5$

A name definition grant specifies that, according to the issuer, a name is defined as another name or a user. For example:

HP grants : $X = user5@hp.com$

HP grants : $X = Y$

Names can be multiply defined, making them capable of representing groups.

These grants can be used to allow another organisation to specify access rights to a resource without knowing the name of the resource. For example:

HP grants : (X issued by AISEC) link access to $//hp.com/accesspoints/accesspoint5$

$AISEC$ grants : $X = user5@aisec.fraunhofer.de$

In the above name definition, HP specifies the access right of the name 'X' and AISEC defines the meaning of the name 'X'. This is similar to RBAC, whereby 'X' would be defined in a similar way. However, 'X' would be treated as a role and not as a name definition.

Moreover, AISEC can allow a third organisation to specify the access rights. For example:

HP grants : (X issued by AISEC) link access to $//hp.com/accesspoints/accesspoint5$

$AISEC$ grants : X = (Y issued by EAB)

EAB grants : $Y = user2@eab.com$

In this case, AISEC does not need to know the name of the resource being accessed or the user that will access the resource. This grant chain can be used in the delegation example depicted in Figure 4.24. The top level provider delegates an access point to one provider and the right to link to the access point to another. During this process, the provider is unaware of the ultimate name

of the access point or who implements it. Furthermore, the top level provider is also not aware of the ultimate provider linking to the access point or the identity used to perform the access.

4.6.1.2 Authorisation Service Prototype Implementation

HP Labs has developed an authorisation server that can receive grants from issuers and can perform authorisation proofs. In addition to performing these tasks, the server uses its own logic to control access to its own services, including the privilege to grant access to the authorisation server. The server is written in Java and has REST [48] and Java Message Service (JMS) [49] based remote interfaces and a Java client end API.

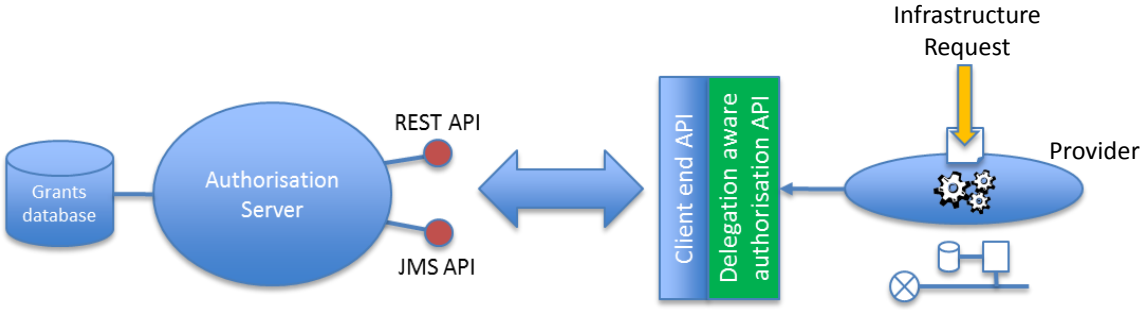


Figure 4.25: Delegation aware authorisation service prototype

We used the authorisation server to implement a global authorisation service as shown in Figure 4.25. We implemented a wrapper on the client API to encode model transformation and delegation actions as name and authorisation grants and to allow a provider to determine resource access authorisation.

The client API for this service includes methods:

- *authorise(resource reference, operation)* to register operations that are permitted on a given resource;
- *transform(resource reference1, resource reference2)* to register virtual infrastructure model transformations in the form of mappings from one resource name (reference) to another;
- *delegate(resource reference1, resource reference2, provider identity)* to register delegation of a resource to another provider;
- *addUser(resource reference, operation, user identity)* to register user identities along with their corresponding access rights to resources;
- *hasAuthority(resource reference, user identity)* to determine if a given user identity is authorised to access a local resource.

In each case the issuer is the calling provider, as established when opening a session with the service. Similar methods exist to de-register each action.

The prototype uses multiple providers (and so multiple issuers) to register the model transformations and delegations performed in various scenarios such as the one depicted in Figure 4.24. Authorisation of DCP interactions such as reference resolution and link negotiation have been tested. Only the owner of a resource can determine if a user has authority to access the resource, as authorisations are relative to the issuer. According to the logic, issuers are not able to determine authorisations granted by other issuers.

4.6.1.3 Alternative Implementations

In this prototype we implemented a single authorisation server. This implies that a single trusted third party provides the authorisation service. Another alternative involves each provider to have its own grants database and to use a distributed implementation of the authorisation proving algorithm. In this case no provider would need to share its grants with any other. The same authorisation logic is applicable for both these cases.

5 Implemented System: Bringing components together

This chapter illustrates how the various components described in Chapter 4 and the test bed described in Chapter 3 have been used in the Dynamic Enterprise scenario.

An overview on the implemented system is provided in Figure 5.1. This illustrates one possible deployment of the virtual infrastructure concerning the use-case presented in Section 2.3.In the example, the enterprise has its headquarters in Stockholm (Sweden) and wants to deploy a virtual infrastructure that spans to to Aveiro (Portugal). A high-level description of the virtual infrastructure can be seen in the Figure 5.1.

The remaining of this Chapter is organized as follows: Section 5.1 provides an overview on the integration of components; Section 5.2 tackles the tenant's requirement providing a deeper understanding on the virtual infrastructure request formulation (step 1 of the use-cases workflow, shown in Section 2.2); and Section 5.3 looks at the interaction among domains, i.e., delegation and inter-domain connectivity establishment (steps 3 and 4).

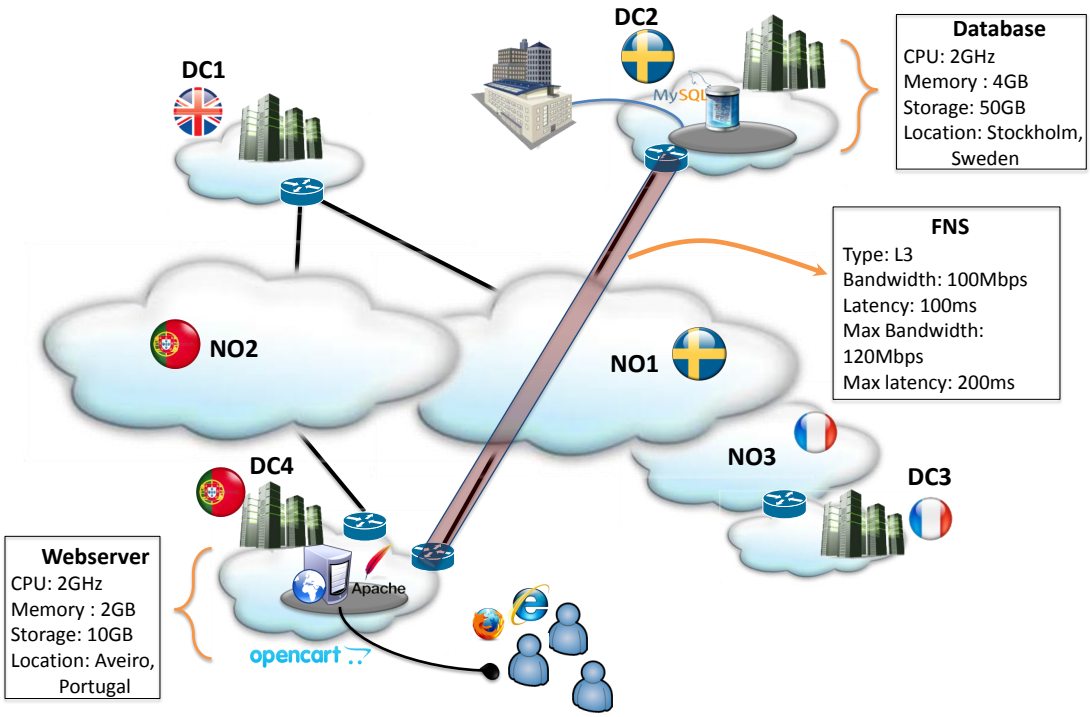


Figure 5.1: Enterprise scenario: Use-case

5.1 Overall integration of components

An illustration of the overall integration of components is provided in Figure 5.2. Note that there is a clear demarcation between the three layers in which the components are organized (see Figure 2.2).

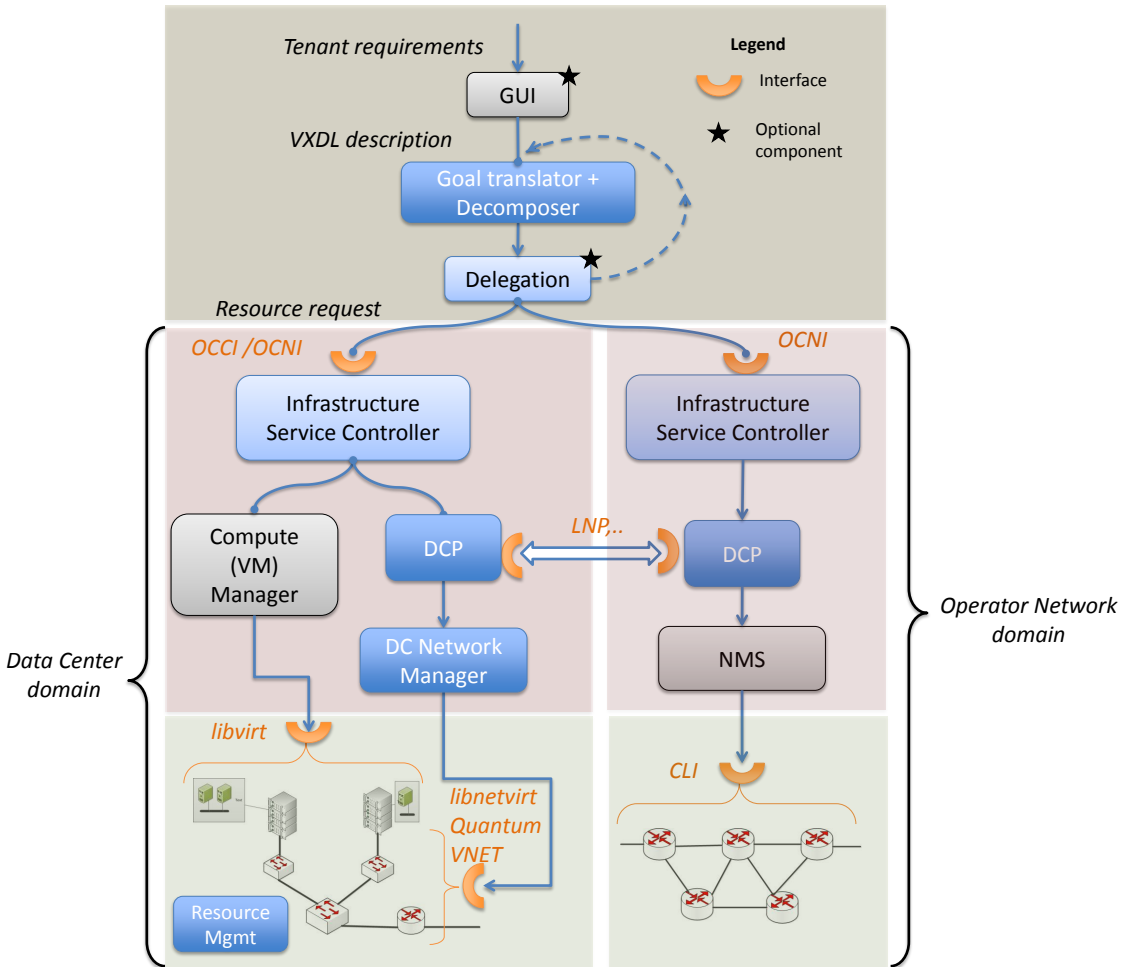


Figure 5.2: Enterprise scenario: Work Flow

The first step refers to the description of the tenant's requirements. Using an optional GUI the requirements are captured using the modelling capability offered by VXDL. The requirements are then analysed and decomposed by the Decomposer module (Section 4.1.2) with goal translation capabilities. The output might be a decision to partially or completely delegate the request, which is then done by the Delegation module.

The request for resources, Infrastructure Service Request (ISR) are sent to the various domains (data centres or network operators) through the Infrastructure Service Interface (namely the OCCI and OCNI) (Section 4.2). These are parsed and interpreted by the corresponding Infrastructure Service Controllers. Any necessary inter-domain coordination needed is done via the DCP (Section 4.3), for example link negotiation.

Finally the physical resources need to be provisioned or configured. In the case of network this is done by means of abstraction libraries like libNetVirt (Section 4.5.1) or with network technologies like VNET (Section 4.4.1) In the case of compute resources an improved management function has been integrated, both in the resource allocation phase as well as in the resource adaptation/optimization phase (Section 4.5.2).

5.2 Tenant Requirements

While the requirements of an enterprise tenant in the real world can be quite broad, for the purpose of this illustration it is considered that the tenant is an Internet based business that runs an e-commerce site, i.e. a web shop. The application in this case is a simple two-tier web application utilizing a front-end webserver (for hosting the website) and a database server (for hosting the data). Both these servers are deployed as virtual machines. The specification of these machines are described in Figure 5.1.

An important fact to note is the distributed nature of the virtual infrastructure, i.e., the virtual machines are not in the same location, due to business reasons. The location tag captures this. Another thing to note is the requirements on the FNS connecting these two nodes. This includes the type of the FNS (L3 in this case) and the bandwidth and latency restrictions on the FNS properties.

The tenant requirements can be modelled by means of the VXDL description as shown in Listing 5.1. Otherwise, the modelling can be simplified by utilizing a GUI as shown in Figure 5.3. The output of the GUI will be similar to the Listing 5.1. Note that the GUI may be more restrictive in the sense that it may not be able to model the entire range of the requirements that can be described using VXDL.

Listing 5.1: VXDL description of the enterprise virtual infrastructure

```
<?xml version="1.0" encoding="UTF-8"?><description>
  <virtualInfrastructure id="E commerce deployment" totalTime="P0Y0M0DT0H0M0.000S"
      owner="" description="E commerce deployment with two tier deployment over a L3
       network">
    <vLink id="1" totalTime="P0Y0M0DT0H0M0.000S" location="" exclusivity="false"
        destination="Webserver" source="DB1">
      <tag>
        <key>Layer</key>
        <value>3</value>
      </tag>
      <bandwidth>
        <forward simple="100.0" unit="Mbps"/>
        <reverse simple="100.0" unit="Mbps"/>
      </bandwidth>
      <latency unit="Mbps"/>
    </vLink>
    <vNode reliability="DC-EAB" id="DB1" totalTime="P0Y0M0DT0H0M0.000S" location="
        Stockholm" exclusivity="false" image="image://db" region="Sweden">
      <cpu architecture="x86_64">
        <cores unit="" simple="1"/>
        <frequency simple="2.0" unit="GHz"/>
      </cpu>
      <memory unit="GB" simple="4.0"/>
      <storage simple="50.0" unit="GB"/>
    </vNode>
    <vNode reliability="DC-PTIN" id="Webserver" totalTime="P0Y0M0DT0H0M0.000S"
        location="Aveiro" exclusivity="false" image="image://webserver" region="
        Portugal">
      <cpu architecture="x86_64">
        <cores unit="" simple="1"/>
        <frequency simple="2.0" unit="GHz"/>
      </cpu>
      <memory unit="GB" simple="2.0"/>
      <storage simple="10.0" unit="GB"/>
    </vNode>
  </virtualInfrastructure>
</description>
```

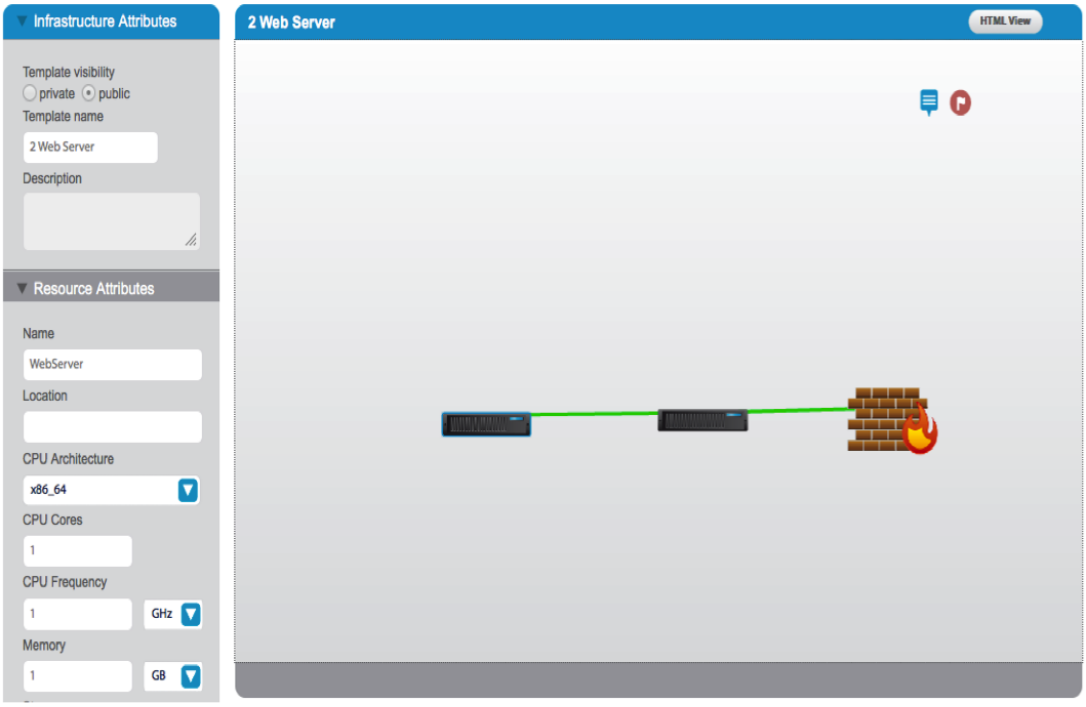


Figure 5.3: Tenant Requested Virtual Infrastructure - in a graphical interface

5.3 Domain interaction

The Virtual Infrastructure (VI) needs to be mapped and instantiated from a (physical) infrastructure which are managed by infrastructure providers. Each infrastructure provider is an administrative domain. The CloNe test bed has seven administrative domains. There are three network operators (NO): EAB, PTIN and IT and four data centres (DC): EAB, PTIN, HP or IT. There is also a Cloud Networking (CloNe) provider which can be Lyatiss (in the Broker role) or PTIN. Figure 5.1 shows the physical infrastructure providers as well.

The message work flow diagram between the domains involved in the enterprise scenario can be seen in Figure 5.4. In the diagram it is possible to see two types of interaction, at the Infrastructure Service Request(ISR) level (messages 1 to 4) and at the DCP level (messages A1 to A5, B1 to B5, C1 to C5). The former relates to the delegation process and the latter to the link creation process, part of the DCP Link Negotiation Protocol (LNP). Looking at Figure 5.2, the tenant request is received and processed (i.e. translated, decomposed and delegated) by the CloNe provider, and in this particular example (which involves not only one network operator, but two) it is decomposed in four parts that are delegated to four of the underlying domains through OCCI (messages 1 and 4) and OCNI (messages 2 and 3). Upon the receipt of the ISR message and acceptance of the request each domain will inform the DCP Controller which may or may not trigger the LNP, depending on a simple set of pre-defined policies.

In this particular instance the referred policies related to the link creation process (i.e. the link creation policies) are: 1) when the process involves a network operator and a data centre, the network operator is responsible for initiating the negotiation; 2) when the process involves two network operators, the network operator with the highest AS number initiates the negotiation; 3) a network operator that needs to negotiate with one or more data centres and one or more network

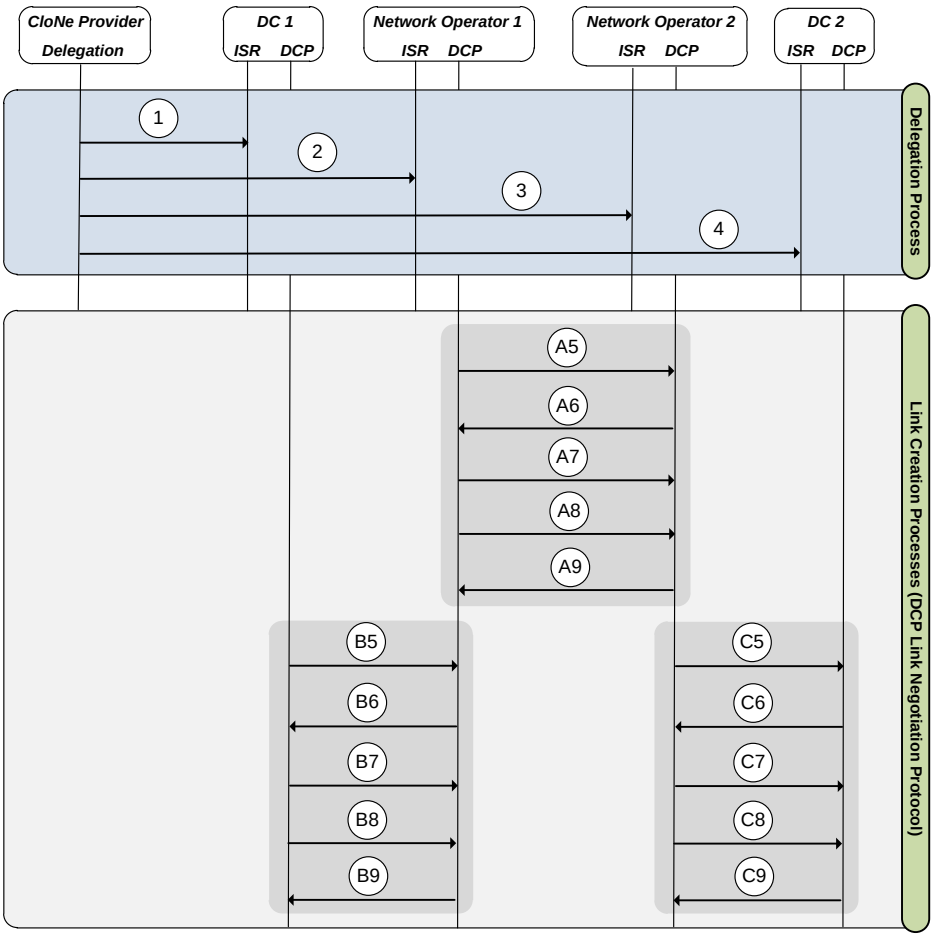


Figure 5.4: Sequence diagram

operators only starts the negotiation with the data centres after having finalized the negotiation with the remote network operators.

5.3.1 Infrastructure Service Request messages

Messages 1 and 4 are OCCI messages for creating a virtual network (L2) and a VM in this network. Listing 9.3 in the appendix shows examples of the actual messages sent to the Infrastructure Service Interface.

Messages 2 and 3 are OCNI messages for creating the FNS (L3) between these VMs. Listing 5.2 shows an example of the OCNI message sent to one of the network operators. In this message one can see the mixin currently being used i.e. *L3VPNMixin* (lines 16-21). Other parameters of interest are the *infrastructure_id* of this virtual infrastructure (line 45) and *service_type* (lines 47 - 50). The *service_description* (lines 51 - 70) shows the properties of the end points, namely the bandwidth and latency characteristics as expressed in the tenant requirements, Figure 5.1.

Listing 5.2: OCNI message to Network Operator

```
1 {
2  POST userID/CloNeLink HTTP/1.1
3  Host: localhost:80
4  Accept: application/ocni+json
```

```
User-Agent: curl/7.13.1 (powerpc-apple-darwin8.0) libcurl/7.13.1 OpenSSL/0.9.7b
    zlib/1.2.2

{
   "id":"ResourceID",
   "title":"CloNe Connectivity 1",
   "kind":{
      "term":"CloNeLink",
      "scheme":"http://schemas.ogf.org/occi/ocni#",
      "class":"kind",
      "title":"Cloud networking Link"
   },
   "mixins":[
      {
         "term":"L3VPNMixin",
         "scheme":"http://example.com/occi/ocni/l3vpn_mixin#",
         "class":"mixin",
         "title":"l3 vpn mixin"
      }
   ],
   "summary":"the description of this CloNeLink instance",
   "links":[
       1
   ],
   "attributes":{
      "availability":[
         {
            "start":"08:00",
            "end":"12:00"
         },
         {
            "start":"14:00",
            "end":"18:00"
         }
      ],
      "bandwidth":"100Mbps",
      "latency":"100ms",
      "jitter":"",
      "loss":"0.01%",
      "routing_scheme":"unicast",

      "infrastructure_id" : "1234",
      "customer_id" : "5",
      "service_type" : {
              "type" : "CaaS",
              "layer" : "L3",
              "class_of_service" : "guaranteed"      },
      "service_description" : [
          {
              "endpoint_id" : "2",
              "in_bandwidth" : "100Mbps",
              "out_bandwidth" : "100Mbps",
              "max_in_bandwidth" : "120Mbps",
              "max_out_bandwidth" : "120Mbps",
              "latency" : "100ms",
              "max_latency" : "200ms"
          },
          {
              "endpoint_id" : "4",
              "in_bandwidth" : "100Mbps",
              "out_bandwidth" : "100Mbps",
```

```
65                "max_in_bandwidth" : "120Mbps",
66                "max_out_bandwidth" : "120Mbps",
67                "latency" : "100ms",
68                "max_latency" : "200ms"
69            }
70        ],
71        "elastisity" : {'
72            "supported" : "yes",
73            "reconfiguration_time" : "120s"
74        },
75        "scalability" : {
76            "supported" : "yes",
77            "setup_time" : "10s"
78          }
79 }
```

5.3.2 Interdomain messages

The network operator receives the OCNI message (e.g. Message 3) to create a FNS between two sites , as shown in Listing 5.2. Since in this case the request is for a L3 FNS, the network operator sets up a L3 VPN between the two sites (endpoints in the OCNI message). But at each of the endpoints the edge nodes at both domains need to be configured for the proper setup of the L3 network. The LNP as described in Section 4.3.1.2 is used for the same. LNP messages are sent between the administrative domains to create a virtual link.

Looking to the diagram 5.4 one can see three link creation processes: one between the two network operators (messages A5 to A9), one between DC(1) and network operator (1) (messages B5 to B9), and another between DC(2) and network operator (2) (messages C5 to C9). Having in mind the above mentioned policies, the chronological sequence of the three link creation processes is clear: first the negotiation between the two operators, and only after the negotiation between each network operator and the corresponding DC takes place. Note that messages B5 to B9 are not related to messages C5 to C9 in any temporal manner.

Table 5.1 shows the matching between the messages in the diagram and the LNP messages.

Table 5.1: Messages matching

Diagram message	Protocol message
A5, B5, C5	*Link Offer*
A6, B6, C6	*Link Select*
A7, B7, C7	*Link Config*
A8, B8, C8	*L3 Offer*
A9, B9, C9	*L3 Config*

The various fields used in the messages have already been described in detail in Section 4.3.1.2. Here we will try to exemplify thembased on the tenant requirement for further understanding.

Listing 5.3 shows the *Link_Offer* message send by the initiator. Here header parameters as specified in Table 4.1 can be seen. The *Link_Offer* message creates the binding between the *virtual_link_id* and the TLL, specifically the *TLL_id*. The *SLL_offer* fields contain the references to the SLL that has been offered for transporting this link.

Listing 5.3: Link Offer message

```
{
    "infra_id": 1234,
    "msg_type": "LINK_OFFER",
```

```
    "sender": "no.se.ericsson",
    "service_type": "L3",
    "transaction_id": 161803399,
    "virtual_link": {
        "TLL": {
            "SLL_offer": [
                4,
                5
            ],
            "TLL_id": 7300
        },
        "in_bw": 100,
        "out_bw": 100,
        "virtual_link_id": 1
    }
}
```

Listing 5.4 shows the *Link_Select* message. Here the *virtual_link* field has been further filled with the SLL that has been selected (*SLL_id*. Another important field is the encapsulation type to be used for this link as specified in the subfield of the *encap_scheme* field (VLAN in this case).

Listing 5.4: Link Select message

```
{
    "infra_id": 1234,
    "msg_type": "LINK_SELECT",
    "sender": "dc.pt.ptinovacao",
    "service_type": "L3",
    "transaction_id": 161803399,
    "virtual_link": {
        "TLL": {
            "SLL_id": 4,
            "SLL_offer": [
                4,
                5
            ],
            "TLL_id": 7300,
            "encap_scheme": {
                "attributes": null,
                "encap_type": "VLAN"
            }
        },
        "in_bw": 100,
        "out_bw": 100,
        "virtual_link_id": 1
    }
}
```

Listing 5.5 shows the *Link_Config* message which finalizes the link. Here the attributes that are defined are the properties of the encapsulation scheme, in this case the VLAN number as decided in the *vlan_id* field.

Listing 5.5: Link Config message

```
{
    "infra_id": 1234,
    "msg_type": "LINK_CONFIG",
    "sender": "no.se.ericsson",
    "service_type": "L3",
    "transaction_id": 161803399,
```

```
        "virtual_link": {
            "TLL": {
                "SLL_id": 4,
                "SLL_offer": [
                    4,
                    5
                ],
                "TLL_id": 7300,
                "encap_scheme": {
                    "attributes": {
                        "vlan_id": 30
                    },
                    "encap_type": "VLAN"
                }
            },
            "in_bw": 100,
            "out_bw": 100,
            "virtual_link_id": 1
        }
    }
```

The information in the above messages provides the details to configure a basic link between two domains. But this is not enough for configuring a L3 connection. The next messages do that. Listing 5.6 shows the *L3_Offer* message which proposes the L3 configuration parameters, specifically the IP address of the edge nodes (fields *ip_src* and *ip_dst*) and the possible routing protocols that can be used for routing between the domains (field *routing_protocol_offer*). This messages is always sent by the initiator like the *Link_Offer* message. Note also that the source and destination are relative to the sender, in this case the initiator of the message.

Listing 5.6: L3 Offer message

```
{
    "infra_id": 1234,
    "msg_type": "L3_OFFER",
    "sender": "no.se.ericsson",
    "service_type": "L3",
    "transaction_id": 161803399,
    "virtual_link": {
        "TLL": {
            "L3_config": {
                "ip_dst": "10.0.0.4/30",
                "ip_src": "10.0.0.3/30",
                "routing_protocol": null,
                "routing_protocol_offer": [
                    "OSPFv2",
                    "RIPv3",
                    "ISIS"
                ]
            },
            "SLL_id": 4,
            "SLL_offer": [
                4,
                5
            ],
            "TLL_id": 7300,
            "encap_scheme": {
                "attributes": {
                    "vlan_id": 30
                },
                "encap_type": "VLAN"
```

```
                }
            },
            "in_bw": 100,
            "out_bw": 100,
            "virtual_link_id": 1
        }
    }
```

The final message is the *L3_Config* message which finalizes the L3 configuration as illustrated in Listing 5.7. Here the routing protocol is agreed (field *routing_protocol*). This might include any extra attributes on the routing protocol itself.

Listing 5.7: L3 Config message

```
{
    "infra_id": 1234,
    "msg_type": "L3_CONFIG",
    "sender": "dc.pt.ptinovacao",
    "service_type": "L3",
    "transaction_id": 161803399,
    "virtual_link": {
        "TLL": {
            "L3_config": {
                "ip_dst": "10.0.0.4/30",
                "ip_src": "10.0.0.3/30",
                "routing_protocol": {
                    "attributes": null,
                    "type": "OSPFv2"
                },
                "routing_protocol_offer": [
                    "OSPFv2",
                    "RIPv3",
                    "ISIS"
                ]
            },
            "SLL_id": 4,
            "SLL_offer": [
                4,
                5
            ],
            "TLL_id": 7300,
            "encap_scheme": {
                "attributes": {
                    "vlan_id": 30
                },
                "encap_type": "VLAN"
            }
        },
        "in_bw": 100,
        "out_bw": 100,
        "virtual_link_id": 1
    }
}
```

6 Potential extensions

The previous chapters described the components we developed during prototyping phase of the project and the test bed where we deployed them. Further extensions have been envisioned to those components, either adding new features to them or correcting shortcomings. Those will be herein described.

6.1 Virtual Infrastructure Modification

A tenant might need to modify the deployed VI in order to adapt to new conditions. For example, the tenant might want to resize links bandwidth and add new web servers because of an increase in the workload caused by use of the application. Or the VI might need to be downsized during night time when nobody is using it, to save costs.

In order to adapt the tenant's VI to these objectives, a mechanism for VI modification is under development. The tenant will use the same description language, VXDL, for defining the part of the modified VI. A RESTful API or a graphical VXDL editor could be used in order to select the part of the VI to modify, change the description and update it. Possible operations would be delete, modify, or add any VXDL element.

The decomposer keeps track of where each part of the VI is deployed. When it receives a new description of an existing part of the VI, it is able to find the data centre or network operator that deployed previously this portion of the VI.

The goal translator will then check if the new constraints still match with the data centre (or the network operator) currently used by the element. If it is the case, an update message is sent that corresponds to the new description of the VI element. If not, a delete message is sent for this VI element and the decomposer will ask another data centre (or another network operator) to deploy the element with the new constraints.

In prototyping both the decomposer and goal translation functions are currently implemented as a broker independent of any infrastructure provider.

6.2 Virtual Infrastructure Monitoring and Elasticity

Once a tenant has deployed a VI, monitoring the different elements of it can be useful to ensure service continuity. The network providers and data centres need to implement an interface that could be queried so as to retrieve monitoring information for a specific element. This could be consolidated and presented graphically.

Elasticity can also be implemented via this monitoring capability. An external software controller could interface via a well defined API, through which it would be able to retrieve this monitoring information, and be automatically warned if some events are triggered. Thus the VI can be adapted according to the new conditions, by sending a new VI description.

6.3 Delegation between Virtual Infrastructure Providers

Though the implemented prototype uses a broker model, CloNe's architecture does not necessarily foresees a CloNe Provider to be a broker element alone. In this sense it has been under development

(by PTIN) a CloNe provider, which has some similarities to the current broker. This component receives VI requests expressed in VXDL, has a goal translation module, a decomposer module, and a delegation module which similar to current broker module delegates the request using OCCI and OCNI. Developing such a component from scratch allows a better dimensioning of the component. Moreover, since it is under PTIN's administrative domain it can also work directly over the PTIN's network management system having therefore a deeper knowledge about the network with no need to delegate through OCNI.

With two CloNe providers, it has been under study the delegation between CloNe providers, namely between Lyattis Broker (CloudWeaver) and PTIN's CloNe provider.

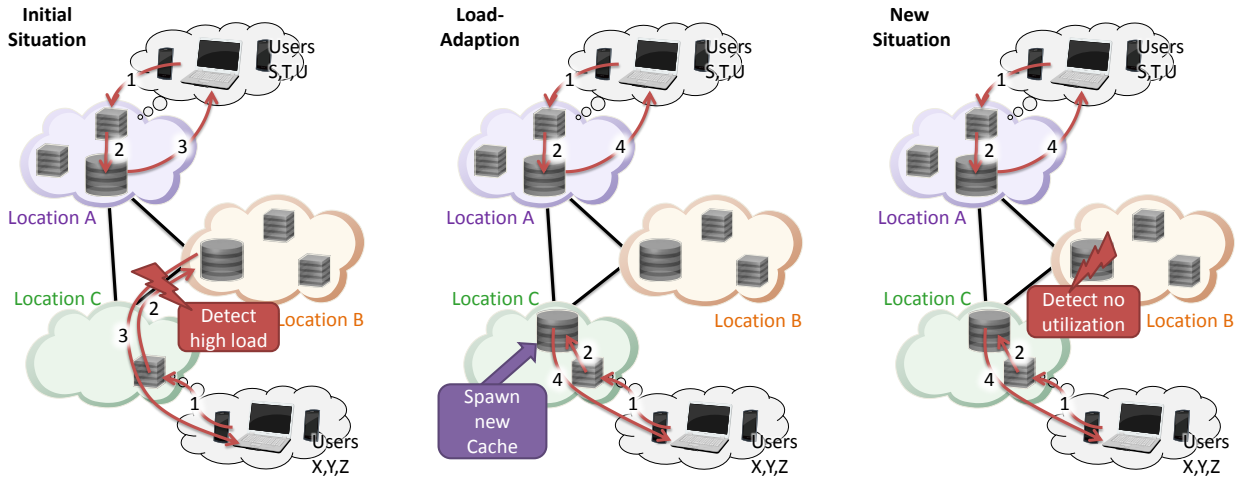
6.4 Load Adaptive Deployment

This approach will fill the traditional gap between *allocating* a Virtual Infrastructure (VI) (as done by data centres) and *using* it (as done by data centre users like application/service providers). This blurs today's sharp line between Infrastructure-as-a-Service and Software-as-a-Service by exchanging management information between Cloud Networking (CloNe) infrastructures and the application. This can range from software related data like usage, utilization and performance metrics to infrastructure information like alternative offerings or cheaper offers of virtual resources. Two benefits arise: a more abstract and unified application management paradigm and with that the potential to automate VI scaling dynamically. This can be used to scale the VI used by an application automatically according to a certain (performance) metric. As a result only necessary resources are allocated and thus costs are saved.

Framework The technical realization requires two parts. First, a broker, that manages and coordinates the VI. Second, an elastic, distributed, and scalable application composed with VMs. The elasticity reflects the capability of the application to shrink and grow during runtime with respect to the number of VMs or in the resources assigned to a single VM. This implies for example data migration to other running VMs if that data is otherwise lost while VMs are shut down, or integration of new nodes/VMs during VI growth. Another property of elasticity is to adapt the application to different data centre locations (e.g., efficiently utilize heterogeneous hardware of different DCs). A central or distributed broker component manages the applications VI and propagates changes to the application layer.

Such a broker component offers high level operations for customers, e.g., for an application provider, to change the application VI as a whole or scaling up, down or migrating parts of the infrastructure (see subsection 6.1). To enable applications to be elastic and change VI on-the-fly, a steering concept is used. It forwards information about what happens on the VI level to the application and vice versa. The application can interact with the infrastructure level, e.g., if more resources are needed somewhere. Thus the steering interface allows exchange of information between the broker component (application level) and VMs (infrastructure level). This way, the broker component can send information to a newly deployed VM about the current VI deployment or context information about the current data centre. Either the application implements such an interface itself or a steering daemon is installed. This daemon has hooks, e.g., calling scripts upon different events. Many useful events/triggers are possible, e.g., before a snapshot, suspend or shut down, enabling the application to change state or save its data. As a result, the application can react appropriately on any VI change. Managing an application VI as a whole allows automating the technical details of VI allocation and application configuration.

Load adaptation With such a framework as described in previous paragraphs, an application can provide (performance) metrics or Online Performance Indicators (OPIs) and the broker component



(a) Detection of high load (b) React by spawning a new cache (c) Detect low utilization

Figure 6.1: Scenario C: Load-adaptive NetInf deployment

can shrink or grow application VI in a load-dependent manner. The application can provide load, usage, or performance data whereas the infrastructure (CloNe) can provide resource utilization data (for example FNS monitoring data), similar to subsection 6.2. Using this data, a placement algorithm decides how to scale the application and where to place the VMs in order to optimize a certain goal, like application service quality, under- or over-utilization, etc. In this sense it is a special application level cross-domain resource manager.

Prototype The prototype is a cross work between Network of Information and CloNe. It develops an adaptive Network of Information (NetInf) deployment across multiple data centres. While different scenarios are described in detail in Chapter 4 of D.A.9[50], the last scenario (Scenario c) describes load adaptive placement of NEC NetInf Router Platform (NNRP) nodes.

We envision a situation where requests for content are already served by the nearest content cache (Location A and C). Customers from Location C are now requesting content, which is not available at Location C but can be served from Location B. As a result, a lot of (inter-provider) traffic is caused (Figure 6.1a). CloNe can detect the high load on the inter-provider link and react with a pre-defined, NetInf-specific action. This action can be part of the deployment request from the NetInf provider. In our example this action is to add a new cache in Location C (Figure 6.1b). As a result the requested objects will also be cached in Location C. After some time no further inter-provider traffic is produced (except for updates). Because CloNe not only monitors the load on the inter-provider links but also the CPU load of the VMs, it now detects that the cache in Location B is no longer required (Figure 6.1c). Again using a predefined policy, this un-utilized node at Location B is shut down to save cost.

Architecture The architecture (shown in Figure 6.2) has an application which has both NetInf and CloNe parts. The NetInf component is running on VMs on CloNe resources and thus can be spread out geographically across infrastructure provider, connected by wide area networks.

The prototype architecture is composed of three components working together on different levels: the NetInf nodes, a Management component, and the CloNe-enabled infrastructure. This architecture distinguishes between two providers: the NetInf service provider and the CloNe infrastructure provider. Both can be the same legal entity, but we focused on a more flexible model in which an

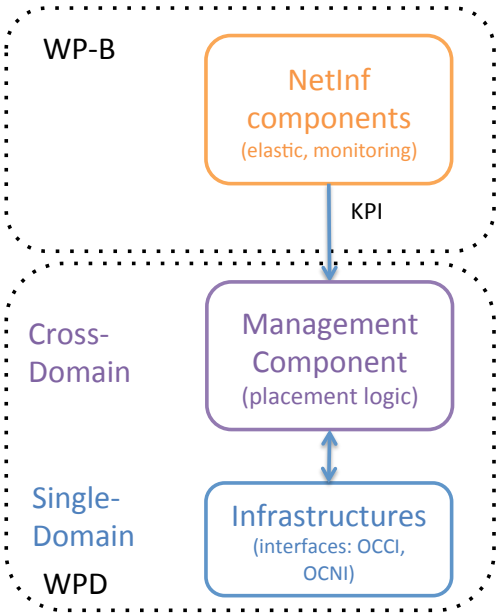


Figure 6.2: System Architecture and Responsible Cross Work

arbitrary NetInf company can deploy its software on arbitrary CloNe providers. Hence, a NetInf provider uses the service offered by a CloNe provider.

More details about the architecture and prototype are provided in Chapter 4 of D.A.9 [50] and in D.D.3[51].

6.5 TPM-based Auditing

The security goal translation function forms the backbone of the CloNe security architecture. It receives security constraints from the different entities involved in the CloNe infrastructure, and translates them to concrete resource specifications. For example, a cloud tenant specifies the geographic location, where its resources need to be hosted. The location constraint is translated by the security goal translation function and forwarded to the auditing and assurance module. The auditing and assurance module invokes the auditing function, which attests the geographic location of the physical resources provisioned to the cloud tenant. The auditing function facilitates the cloud tenant's policy compliance.

The setting of the prototype for the geo-positioning function is shown in Figure 6.3. The geo-positioning function uses the TPM [52] of the physical machine as a trust-anchor for validating the geo-position of virtual machines. The TPMs have to be registered by a trusted authority and the identity and public keys of the TPM and its geo-location needs to be stored at a CA. A VM placed on a machine with such a TPM can access the identifier of the TPM via the hypervisor and check the TPM's location at the CA. To guarantee that this process of validating the geo-position cannot be manipulated, we plan to build the prototype as follows.

We will use XEN as hypervisor and create a XEN module "location check" that implements the access to the TPM. Additionally, we will implement a hardware driver inside the VM system that implements the access to the XEN location check function and requests the geo-position of the TPM at the CA. In the prototype, the CA is a simple web service that maps a TPM key to a geo-position, signs this information, and sends it back to the caller. To guarantee that the access to the TPM cannot be altered, we check the correctness of the location check function during boot time by using the attestation function of the TPM.

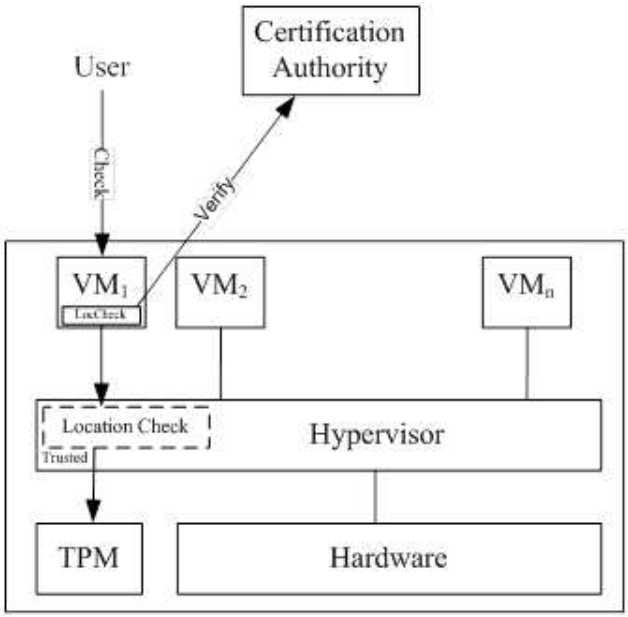


Figure 6.3: Setting for TPM-based Geo-positioning

To show the functionality of the geo-positioning process a user can call the location check function inside its VM. This function generates a challenge and calls the location check module with this challenge as input. The XEN location check module connects to the TPM and gets the challenge signed by the TPM private key. This signed challenge together with the TPM identifier goes back to the VM extension via the XEN module. The VM extension requests the CA for the geo-position of the TPM by using the TPM's identifier. The CA sends back the signed geo-position and the TPM's public key. The VM's geo-position function can check whether the messages have been received from the legitimate CA. By verifying the message signature, the geo-position function can validate the correctness of the signed challenge and the geo-position of the TPM.

7 Prototyping remarks

Prototyping in any large project is a challenging task. Implementing an integrated prototype in a large scale, pan-European heterogeneous test bed with multiple teams from various partners spread over a continent increases the scale of the challenge considerably. The work involves not only the actual code that must be written or the systems that must be configured, but also a significant level of coordination, communication and technical decisions that are needed to finally show a proof of concept demonstrator.

The top priority from the beginning of CloNe has been its experiment driven nature and the validation of its ideas by successful implementation of the key proposed technical solutions and methods. This makes prototyping an essential task.

Prototyping in CloNe in earnest only started after the first version of architecture was finalized. This laid out the basic blueprint for the work with the terminology and conceptual models. Though the architecture provided the foundation for DCP, a critical piece of the puzzle for cross-domain connectivity, the LNP was entirely defined and implemented by the prototyping phase. LNP was defined in an iterative manner with the first version using a standard L3 VPN model and terminology. This was later developed into the second and current version which took feedback from the implementation and is much more flexible and technology agnostic. This work is now fed back into architecture.

7.1 Networking aspects

While the implementation of the prototype brought many benefits, there were certainly hindrances and difficulties that were encountered along the way. The development of the test bed was one such example. CloNe did not have any existing test bed to rely upon and the entire test bed was created from scratch consuming significant amount of time. Open source platforms (OpenStack and OpenNebula) were utilized as the data centre cloud platform in our setup. The issues the technical teams had to face and cope with includes the well-known drawbacks of open source, lack of support, compatibility issues and adequate or up-to-date documentation. Both platforms (especially OpenStack) were also under constant evolution during the prototyping phase with multiple releases for new features and bug fixes. Upgrading the platform to take advantage of new features is highly desirable but most of the time this also involved changes in the system level APIs that broke CloNe developed components. An example of the same is the development of the Quantum project in OpenStack, which overhauled the entire networking subsystem. The technical teams have been pragmatic and have overcome this challenge quite successfully as seen in the integrated prototype.

The WAN and interconnecting domains part of the test bed setup had also its own challenges. Since there was no dedicated connectivity between the domains, IPSec tunnels had to be setup between the domains over Internet. Technical teams had to work with issues involving corporate firewalls and administrative policies as well as practical issues involved in creating IPSec tunnel with equipment of different vendors in each end. Lack of L2 connectivity over IPSec involved setting up of GRE tunnels over IPSec.

The network operator domains in the test bed used MPLS as the networking technology. Though in use widely, MPLS networks capable of providing VPN services are fairly complex to setup and

configure properly. One of the domains (Ericsson) had a virtualized network, where the network nodes were virtual machines with both control and data plane running in software. This allowed constructing a fairly large network with lower number of physical machines. But this kind of setup is both fairly advanced as well as novel. In fact that initial IP stack investigated for the purpose was found lacking in the ability to create multiple Virtual Routing Function (VRF)s. Without that, support for connecting multiple tenants through separate tunnels would be impossible. This was fixed and fully supported during the prototyping phase.

Though IPSec provided point-to-point links between sites, realizing the full interplay among the test bed needed a far more complex network setup at the sites. Heavy use of software based virtual switching was made, especially OpenVSwitch for this as illustrated in Section 3.3. OpenVSwitch was found to be extremely flexible and adaptable for experimental test beds of this nature.

Networking requirements in WANs are mature and well understood, but not so in cloud management platforms for Data centres. CloNe is mandated to look into these aspects, but certain basic features were always assumed to exist in these platforms and expected to be only adapted where needed. Because of the inter-domain focus of CloNe prototyping, this was a necessity too. But an important note from the prototyping experience was that networking in today's cloud management platforms, especially open source are highly geared towards the specific use cases they are supporting, namely private IaaS clouds or public IaaS clouds like Amazon. As such even certain basic networking capabilities that are not common in these use cases tend to be unavailable, especially in the earlier days of prototyping. A case in point is the ability to create a VM connected to multiple networks or the ability to route between two tenant networks. Some of these limitations were fixed over time or else had been developed by CloNe to fill the gap.

7.2 Application, Interface and Implementation aspects

There were application related aspects to take care as well. The prototyping demonstration shows an enterprise web application deployed by a webshop. An open source e-commerce application [4] was used which used Apache as the webserver and MySQL as the backend. It was customised for distributed deployment, specifically a zero touch config that enables the application to be utilized as soon as the FNS is up. This also involved the deployment of custom virtual machine images with the needed config. Though this enables a fully automated setup in our prototype, most public clouds do not allow custom images to be used. Further, there does not exist any good solutions for using one image across multiple providers either.

As described earlier, the use of OCCI enabled a rich data model and extension mechanism enabling OCNI. But there were challenges arising from the fact that OCCI allows multiple representations (or renderings) of the same underlying data model. There are differences in the implementations used in the different DCs used in the test bed. This forces a more intelligent client side implementation. We have tried to simplify and keep a consistent interface across platforms but this is still an on going work.

In a few occasions different implementations of the same solution were made (see the LNP implementation made at Ericsson, PT, IT and HP) because different cloud management solutions or versions of the same were used (OpenStack vs. OpenNebula). In the end, that has demonstrated the strength of our solution, which works across heterogeneous technologies.

7.3 Other issues

Difficulties faced and experiences gained from the hands on work in prototyping have contributed to identify issues that had not been foreseen by the architecture. This includes remote network discovery and overlapping address spaces.

Remote network discovery is needed to properly configure internal firewalls used by data centres. Most L2 networks created for hosting VMs use host based firewall rules to limit traffic in and out of the created network for both L2 and L3 traffic. This causes issues for traffic intended to remote nodes in a distributed virtual infrastructure. Exact details of the remote networks needed to be available before these firewalls can be appropriately configured. A remote network discovery protocol is needed, potentially as part of the DCP to configure this.

Another related issue is the IP address schemes to be used for the network nodes acting as the edge nodes (CE, PE) between the data centre and WAN, especially in L3 networks. The addresses used for the network interfaces should not overlap with the addresses used in the tenant networks. Unless globally specified by the decomposer element or equivalent as part of the ISR, there can be collisions in the address space, which needs to be resolved. Some ways of tackling it involves the use of public addresses, reserving address spaces for CE-PE networks and similar proposals[53], or a new conflict detection and resolution protocol as part of the DCP. A similar issue is the end-to-end coordination of VLANs in a distributed L2 network.

These practical issues need to be addressed for a real life deployment of CloNe.

7.4 Agile Development practices

Another challenge that shall not be forgotten is the coordination to implement one integrated prototype within the work package. Multiple teams from different partners working independently had to work together and agree upon interfaces, platforms, network technologies, and many more. The prototyping team used many of the techniques and tools used in agile development practices like daily developer meetings, shared collaboration tools like Etherpad and communication tools like Skype extensively for this purpose during the development and integration periods.

The team also used developer only meetings, integration sprints (dedicated week or two for integration and testing), shared code repositories and in some cases shared template/boiler plate code (e.g., LNP implementation).

8 Conclusion

This document described the work performed under the prototyping task in the Cloud Networking work package. The concepts developed in the architecture task laid ground for the creation of a comprehensive prototype running on real hardware over live private networks and the Internet. Obviously, implementation details closed the gap between the concepts developed in the architecture, and the concrete running prototype we have now. Those levels of detail would have never been achieved without prototyping and experimentation.

One of the main purposes of prototyping was to test implementation feasibility of the architecture. In that respect, we can state that the task has succeeded. Key architectural concepts were implemented and tested; those include:

- A Distributed Control Plane (DCP) that binds resources in the data centre and wide area network together, more specifically the Link Negotiation Protocol (LNP) and the Cloud Message Brokering Service (CMBS) for inter-domain communication.
- Flash Network Slices implemented through L2VPNs, L3VPNs and OpenFlow networks.
- A virtual infrastructure description language (VXDL) used to describe full virtual infrastructures and resource requirements.
- A network abstraction layer (libNetVirt) that allows for easy management of wide-area and data centre networks.
- A network service interface (OCNI) for allocation of network resources that fits the cloud service model.
- An advanced VM scheduling mechanism for OpenStack utilizing real time measurements to achieve load balancing, service differentiation and server consolidation.
- A delegation aware global authorization service for multi-domain delegation.

All of those initially abstract concepts as defined in the architecture were made concrete in the implementation phase, increasing the understanding of their nature and need. A clear example of that is the distributed control plane, which was precisely defined in the D.D.1, but was only made palpable by the implementation of the Link Negotiation Protocol.

One of the practical principles adopted during implementation was the use of existing de facto cloud computing tools. CloNe has successfully integrated its implemented components with existing cloud management systems, such as OpenStack and OpenNebula. Through experimentation we have shown that the concepts we are proposing can be implemented on top of widely accepted existing cloud management solutions. Where those tools have shown limitations to provide the CloNe vision, we have implemented new protocols and interfaces (e.g., DCP) and extended others (e.g., OCNI). The likelihood of acceptance of our solution increases as we increment existing accepted software.

Finally, the developed prototype was publicly demonstrated at FuNeMS 2012[1]. WPD can proudly state that all of the main elements of the architecture have been implemented and validated through prototyping.

9 Appendix

9.1 OCNI Mixins

9.1.1 L3 and L2 VPN Mixin attributes

The VPN mixin for OCNI defines the following attributes.

Attribute	Type	Multiplicity	Mutability	Description
infrastructure_id	Integer	1	Mutable	Identifier of the infrastructure this link belongs to
customer_id	Integer	1	Mutable	Identifier of the customer who sent this request
service_type	(type, layer, class_of_service)	1	Mutable	Type of service requested
service_description	Class	0..	Mutable	Details of the service
elastisity	(supported, reconfigura-tion_time)	0..1	Mutable	Elasticity requirements for this service
scalability	(supported, setup_time)	0..1	Mutable	Scalability requirements for this service

9.1.2 OCNI VPN Message Example

Listing 9.1: OpenFlow path establishment message using the OCNI interface

```
{
 POST userID/CloNeLink HTTP/1.1
 Host: localhost:80
 Accept: application/ocni+json
 User-Agent: curl/7.13.1 (powerpc-apple-darwin8.0) libcurl/7.13.1 OpenSSL/0.9.7b
     zlib/1.2.2

{
   "id":"ResourceID",
   "title":"CloNe Connectivity 1",
   "kind":{
      "term":"CloNeLink",
      "scheme":"http://schemas.ogf.org/occi/ocni#",
      "class":"kind",
      "title":"Cloud networking Link"
   },
   "mixins":[
      {
         "term":"L3VPNMixin",
         "scheme":"http://example.com/occi/ocni/l3vpn_mixin#",
         "class":"mixin",
         "title":"l3 vpn mixin"
      }
```

```
    ],
    "summary":"the description of this CloNeLink instance",
    "links":[
        1
    ],
    "attributes":{
       "availability":[
          {
             "start":"08:00",
             "end":"12:00"
          },
          {
             "start":"14:00",
             "end":"18:00"
          }
       ],
       "bandwidth":"100Mbps",
       "latency":"100ms",
       "jitter":"",
       "loss":"0.01%",
       "routing_scheme":"unicast",

       "infrastructure_id" : "1234",
       "customer_id" : "5",
       "service_type" : {
               "type" : "CaaS",
               "layer" : "L3",
               "class_of_service" : "guaranteed"      },
       "service_description" : [
           {
               "endpoint_id" : "2",
               "in_bandwidth" : "100Mbps",
               "out_bandwidth" : "100Mbps",
               "max_in_bandwidth" : "120Mbps",
               "max_out_bandwidth" : "120Mbps",
               "latency" : "100ms",
               "max_latency" : "200ms"
           },
           {
               "endpoint_id" : "4",
               "in_bandwidth" : "100Mbps",
               "out_bandwidth" : "100Mbps",
               "max_in_bandwidth" : "120Mbps",
               "max_out_bandwidth" : "120Mbps",
               "latency" : "100ms",
               "max_latency" : "200ms"
           }
       ],
       "elastisity" : {'
           "supported" : "yes",
           "reconfiguration_time" : "120s"
       },
       "scalability" : {
           "supported" : "yes",
           "setup_time" : "10s"
         }
}
```

9.1.3 OCNI OpenFlow Mixin attributes

The OpenFlow mixin for OCNI defines the following attributes.

Attribute	Type	Multiplicity	Mutability	Description
infrastructure_id	Integer	1	Mutable	Identifier of the infrastructure this link belongs to
customer_id	Integer	1	Mutable	Identifier of the customer who sent this request
state	String	1	Mutable	state of the link
bandwidth	Integer	1	Mutable	Bandwidth of the link
latency	Integer	1	Mutable	Latency of the link
jitter	Integer	1	Mutable	Jitter of the link
loss	String	1	Mutable	Loss of the link
routing_scheme	String	1	Mutable	Routing scheme to be used
IPv4	Class	1	Mutable	IPv4 properties
Availability	Class	1	Mutable	Availability properties

9.1.4 OCNI OpenFlow message example

Listing 9.2: OpenFlow path establishment message using the OCNI interface

```
{
  "kind": {
    "term": "CloNeLink",
    "scheme": "http://schemas.ogf.org/occi/ocni",
    "class": "kind"
  },
  "occi.core.id": "OF_Link1",
  "occi.core.title": "OF_Link1",
  "occi.core.summary": "CloNeLink with openflow mixin",
  "mixins": [
    {
      "term": "OpenFlowLink",
      "scheme": "http://schemas.ogf.org/occi/ocni",
      "class": "mixin"
    }
  ],
  "links": [ 1 ],
  "attributes": {
    "infrastructure_id" : "1234",
    "customer_id" : "5",
    "ocni.OpenFlowLink.state":"active",
    "ocni.OpenFlowLink.bandwidth":"100Mbps",
    "ocni.OpenFlowLink.latency":"100ms",
    "ocni.OpenFlowLink.jitter":"",
    "ocni.OpenFlowLink.loss":"0.01%",
    "ocni.OpenFlowLink.routing_scheme":"unicast",
    "ocni.OpenFlowLink.IPv4.src":"12.0.0.12",
    "ocni.OpenFlowLink.IPv4.dst":"12.0.0.8",
    "ocni.OpenFlowLink.IPv4.proto":["tcp","icmp"]
    "ocni.clonelink.availability":[
      {
        "ocni.availability.start":"08:00",
        "ocni.availability.end":"12:30"
      }
    ],
  }
}
```

9.2 OCCI Message Examples

Listing 9.3: OCCI messages to Data Center

```
Creating a Network
------------------

Request:
--------

POST /network/ HTTP/1.1
User-Agent: curl/7.19.7 (x86_64-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k zlib
    /1.2.3.3 libidn/1.15
Host: localhost:8888
Accept: */*
X-OCCI-Attribute: occi.core.title="OCCI Test Network",occi.core.summary="OCCI test
    interface VLAN network",occi.network.allocation="static",occi.network.address
    ="192.168.4.0/24",occi.network.vlan=4,occi.network.label="sail_test_vlan",occi.
    network.infrastructure_id=1234
Category: network;scheme="http://schemas.ogf.org/occi/infrastructure#";class="kind
    ";,ipnetwork;scheme="http://schemas.ogf.org/occi/infrastructure/network#";class
    ="kind";

Response:
---------
HTTP/1.1 200 OK
Content-Length: 45
Content-Type: text/html; charset=UTF-8
Location: http://localhost:8888/network/35a362b8-a7ab-4d72-a79d-18afa205f003
Server: pyocci OCCI/1.1

/network/35a362b8-a7ab-4d72-a79d-18afa205f003


Creating a VM on that network:
------------------------------

Request:
--------
POST /compute/ HTTP/1.1
User-Agent: curl/7.19.7 (x86_64-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k zlib
    /1.2.3.3 libidn/1.15
Host: localhost:8888
Accept: */*
Link: </network/35a362b8-a7ab-4d72-a79d-18afa205f003>;rel="http://schemas.ogf.org/
    occi/infrastructure#network";category="http://schemas.ogf.org/occi/
    infrastructure#networkinterface";scheme="http://schemas.ogf.org/occi/
    infrastructure#networkinterface#ipnetworkinterface";
X-OCCI-Attribute: occi.core.title="OCCI Test VM",occi.core.summary="OCCI test
    interface VM",occi.compute.architecture="x64",occi.compute.cores=1,occi.compute.
    memory=1
Category: compute; scheme="http://schemas.ogf.org/occi/infrastructure#";class="kind
    ";

Response:
---------
HTTP/1.1 200 OK
Content-Length: 45
Content-Type: text/html; charset=UTF-8
Location: http://localhost:8888/compute/7ff3c81e-fded-4ddc-b18e-503ff9dd4f3c
```

```
Server: pyocci OCCI/1.1

/compute/7ff3c81e-fded-4ddc-b18e-503ff9dd4f3c


Starting the created VM:
------------------------

Request:
--------
POST /compute/7ff3c81e-fded-4ddc-b18e-503ff9dd4f3c?action=start HTTP/1.1
User-Agent: curl/7.19.7 (x86_64-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k zlib
    /1.2.3.3 libidn/1.15
Host: localhost:8888
Accept: */*
Category: start; scheme="http://schemas.ogf.org/occi/infrastructure/compute/action
    #";class="action";

Response:
---------
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: text/html; charset=UTF-8
Server: pyocci OCCI/1.1

OK

Deleting the VM:
----------------

Request:
--------
DELETE /compute/7ff3c81e-fded-4ddc-b18e-503ff9dd4f3c HTTP/1.1
User-Agent: curl/7.19.7 (x86_64-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k zlib
    /1.2.3.3 libidn/1.15
Host: localhost:8888
Accept: */*

Response:
---------
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: text/html; charset=UTF-8
Server: pyocci OCCI/1.1

OK

Deleting the Network:
---------------------

Request:
--------
DELETE /network/35a362b8-a7ab-4d72-a79d-18afa205f003 HTTP/1.1
User-Agent: curl/7.19.7 (x86_64-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k zlib
    /1.2.3.3 libidn/1.15
Host: localhost:8888
Accept: */*

Response:
---------
HTTP/1.1 200 OK
```

```
Content-Length: 2
Content-Type: text/html; charset=UTF-8
Server: pyocci OCCI/1.1

OK
```

List of Acronyms

ACL Access Control List

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

AS Autonomous Systems

BGP Border Gateway Protocol

CA Certification Authority

CE Customer Edge

CloNe Cloud Networking

CLI Command Line Interface

CMBS Cloud Message Brokering Service

CRUD Create, Read, Update, Delete

DC Data Centre

DCP Distributed Control Plane

DHCP Dynamic Host Configuration Protocol

EC2 Elastic Compute Cloud

FNS Flash Network Slice

GRE Generic Routing Encapsulation

GRMP Generic Resource Management Protocol

GUI Graphical User Interface

IaaS Infrastructure as a Service

IP Internet Protocol

IPSec Internet Protocol Security

ISI Infrastructure Service Interface

ISR Infrastructure Service Request

JMS Java Message Service

KVM Kernel-based Virtual Machine

LCS Least Cost Scheduler

LDP Label Distribution Protocol

LNP Link Negotiation Protocol

MAC Media Access Control

MPLS Multi-Protocol Label Switching

NetInf Network of Information

NMS Network Management System

NO Network Operator

NNRP NEC NetInf Router Platform

OCA OpenNebula Cloud API

OCCI Open Cloud Compute Interface

OCNI Open Cloud Network Interface

OGF Open Grid Forum

OPI Online Performance Indicator

OSPF Open Shortest Path First

OVS Open Virtual Switch

PE Provider Edge

RBAC Role Based Access Control

REST Representational State Transfer

RIP Routing Information Protocol

SAIL Scalable Adaptive Internet soLutions

SLA Service Level Agreement

SLL Service Provider Logical Link

TLL Tenant Logical Link

TPM Trusted Platform Module

VI Virtual Infrastructure

VLAN Virtual Local Area Network

VM Virtual Machine

VXDL Virtual eXecution Description Language

VPN Virtual Private Network

VPLS Virtual Private Line Service

VPWS Virtual Private Wire Service

VRF Virtual Routing Function

WAN Wide Area Network

WP Work Package

XML-RPC eXentisible Markup Language - Remote Procedure Call

List of Figures

List of Tables

References

[1] Future Network Mobile Summit 2012, Berlin, Germany. http://www.futurenetworksummit.eu/2012/.

[2] Paul Murray et al. Cloud Networking Architecture Description. Deliverable FP7-ICT-2009-5-257448-SAIL/D.D.1, SAIL project, July 2011. Available online from http://www.sail-project.eu.

[3] Benoit Tremblay et al. Description of project wide scenarios and use cases. Deliverable FP7-ICT-2009-5-257448-SAIL/D.A.1, SAIL project, April 2011. Available online from http://www.sail-project.eu.

[4] OpenCart - open source shopping cart solution. http://www.opencart.com/.

[5] Aled Edwards, Anna Fischer, and Antonio Lain. Diverter: a new approach to networking within virtualized infrastructures. In Jeffrey C. Mogul, editor, *WREN*, pages 103–110. ACM, 2009.

[6] Daniel Turull, Markus Hidell, and Peter Sjödin. libNetVirt: the network virtualization library. In *Workshop on Clouds, Networks and Data Centers (ICC'12 WS - CloudNetsDataCenters)*, Ottawa, Canada, June 2012.

[7] OCCI - Open Cloud Computing Interface. Online URL: http://www.occi-wg.org/.

[8] VXDL Forum. http://www.vxdlforum.org.

[9] EC2 API. http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/using-query-api.html. Accessed: 31/05/2012.

[10] vCloud API. http://communities.vmware.com/community/vmtn/developer/forums/vcloudapi. Accessed: 31/07/2012.

[11] XenServer. http://http://www.citrix.com/English/ps2/products/product.asp?contentID=683148. Accessed: 31/07/2012.

[12] Kvm. http://www.linux-kvm.org/page/Main_Page. Accessed: 31/07/2012.

[13] OpenStack website. http://www.openstack.org. Accessed: 31/05/2012.

[14] Iptables - linux man page. http://linux.die.net/man/8/iptables.

[15] Ebtables - linux man page. http://linux.die.net/man/8/ebtables.

[16] libvirt. http://libvirt.org/.

[17] OpenNebula website. http://opennebula.org/. Accessed: 31/05/2012.

[18] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[19] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[20] Ustin Pettit, Jesse Gross, Ben Pfaff, Martin Casado, and Simon Crosby. Virtual switching in an era of advanced edges. In *2nd Workshop on Data Center - Converged and Virtual Ethernet Switching (DC CAVES)*, September 2010.

[21] Guilherme Koslovski, Pascale Vicat-Blanc Primet, and Andrea Schwertner Charão. VXDL: Virtual Resources and Interconnection Networks Description Language. In *GridNets 2008*, Oct. 2008.

[22] N. M. Mosharaf Kabir Chowdhury and Raouf Boutaba. Network virtualization: State of the art and research challenges. volume 47, pages 20–26, July 2009.

[23] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. volume 38, pages 17–29, New York, NY, USA, 2008. ACM.

[24] N. M. Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. April 2009.

[25] Y. Zhu and M. Ammar. Algorithms for assigning substrate network resources to virtual network components. pages 1–12, April 2006.

[26] G. Koslovski, S. Soudan, P. Gonçalves, and P. Vicat-Blanc. Locating Virtual Infrastructures: Users and InP Perspectives. In *12th IEEE/IFIP International Symposium on Integrated Network Management - Special Track on Management of Cloud Services and Infrastructure*, Dublin, Irland, 2011.

[27] Jens Lischka and Holger Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In Guru M. Parulkar and Cédric Westphal, editors, *VISA*, pages 81–88. ACM, 2009.

[28] Guilherme Koslovski, Wai-Leong Yeow, Cedric Westphal, Tram Truong Huu, Johan Montagnat, and Pascale Vicat-Blanc Primet. Reliablility Support in Virtual Infrastructures. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 49–58, Indianapolis, USA, November 2010. IEEE.

[29] Apache Deltacloud API. `http://deltacloud.apache.org/`.

[30] pyOCNI - a Python implementation of an extended OCCI with a JSON serialization and a cloud networking extension. Online URL: http://occi-wg.org/2012/02/20/occi-pyocni/.

[31] Eventlet. `http://eventlet.net/`. Accessed: 31/07/2012.

[32] ZODB. `http://www.zodb.org/`. Accessed: 31/07/2012.

[33] Metro Ethernet Forum - ELAN. `http://metroethernetforum.org/PDF_Documents/technical-specifications/MEF6-1.pdf`.

[34] Virtual Private Line Service. `http://tools.ietf.org/html/draft-ietf-l2vpn-vpls-ldp-09`.

[35] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimanyi. EQS: An Elastic and Scalable Message Queue for the Cloud. In Costas Lambrinoudakis, Panagiotis Rizomiliotis, and Tomasz Wiktor Wlodarczyk, editors, *CloudCom*, pages 391–398. IEEE, 2011.

[36] AMQP. `http://www.amqp.org/product/architecture`.

[37] ZeroMQ - The Intelligent Transport Layer. Online site: http://www.zeromq.org/.

[38] The SAIL Consortium. Architecture and Mechanisms for Connectivity Services. Deliverable FP7-ICT-2009-5-257448-SAIL/D.C.2, SAIL project, July 2011. Available online from http://www.sail-project.eu.

[39] Netfilter. http://www.netfilter.org/. Accessed: 31/05/2012.

[40] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, New York, NY, USA, 2010. ACM.

[41] Eric Keller and Jennifer Rexford. The "platform as a service" model for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, page 4, Berkeley, CA, USA, 2010. USENIX Association.

[42] OpenStack Least Cost Scheduler. `https://github.com/openstack/nova/blob/stable/essex/nova/scheduler/least_cost.py`.

[43] F. Wuhib, R. Stadler, and M. Spreitzer. A gossip protocol for dynamic resource management in large cloud environments. *Network and Service Management, IEEE Transactions on*, 2012.

[44] Rerngvit Yanggratoke, Fetahi Wuhib, and Rolf Stadler. Gossip-based resource allocation for green computing in large clouds. In *International Conference on Network and Service Management*, October 2011.

[45] Fetahi Wuhib, Rolf Stadler, and Hans Lindgren. Dynamic resource allocation with management objectives : Implementation for an openstack cloud. Technical Report 2012:021, KTH, Communication Networks, 2012. QC 20120528.

[46] Schoo, Peter and Fusenig, Volker and Souza, Victor and Melo, Marcio and Murray, Paul and Debar, Herve and Medhioub,Houssem and Zeghlache,Djamal. Challenges for Cloud Networking Security. In Kostas Pentikousis, Ramón Agüero Calvo, Marta García-Arranz, and Symeon Papavassiliou, editors, *MONAMI*, volume 68 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 298–313. Springer, 2010.

[47] Jose M. Alcaraz Calero, Nigel Edwards, Johannes Kirschnick, Lawrence Wilcock, and Mike Wray. Toward a Multi-Tenancy Authorization System for Cloud Services. *IEEE Security and Privacy*, 8(6):48–55, November 2010.

[48] Hyuck Han, Shingyu Kim, Hyungsoo Jung, H.Y. Yeom, Changho Yoon, Jongwon Park, and Yongwoo Lee. A RESTful Approach to the Management of Cloud Infrastructure. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, pages 139–142, sept. 2009.

[49] Hsiu-Hui Lee and Chun-Hsiung Tseng. A software framework for Java message service based Internet messaging system. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 161–165, aug. 2003.

[50] Benoit Tremblay et al. Description of Overall Prototyping Use Cases, Scenarios and Integration Points. Deliverable FP7-ICT-2009-5-257448-SAIL/D-2.9, SAIL project, June 2012. Available online from http://www.sail-project.eu.

[51] Paul Murray et al. Cloud Networking Architecture Description. Deliverable FP7-ICT-2009-5-257448-SAIL/D.D.3, SAIL project, October 2012. Available online from http://www.sail-project.eu.

[52] Dan Wang and Dengguo Feng. A Hypervisor-Based Secure Storage Scheme. In *Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on*, volume 1, pages 81–86, april 2010.

[53] Address allocation for pe-ce links within a provider provisioned vpn network. `http://http://tools.ietf.org/html/draft-guichard-pe-ce-addr-03`.