2013 IEEE 37th Annual Computer Software and Applications Conference

ProViNet - An Open Platform for Programmable Virtual Network Management

Wanderson Paim de Jesus, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{wpjesus, jwickboldt, granville}@inf.ufrgs.br

***Abstract*—The disheartening and thorny path followed while deploying new solutions in the core of current computer networks, culminates in a low rate of emergence of innovation. Several researches proposed applying Software Defined Networks (SDN) and the Network Virtualization (NV) concepts to reverse this rate. However, many gaps and open challenges were observed in the application of these concepts together. In this paper, we propose the ProViNet platform, which merge the SDN and NV concepts to provide a fast and safe deployment of innovations in the existing network infrastructure. ProViNet advance the state-of-art introducing the concept of Network Programming as a Service, in which applications are built simply by composing control plane services. Such approach encourages End-Users to develop and share novel network solutions, fostering innovation. To prove concept and technical feasibility, we evaluate ProViNet with a prototype that allowed demonstrating virtual network infrastructure provisioning and programming.**

I. Introduction

Today, the core of computer networks is, when compared with servers, desktops, and mobile devices, an unfriendly environment for innovation. In the case of the Internet, this fact is often refereed to as the Internet ossification [1]. Solutions like IPv6 and IPSec, for example, proposed for over 10 years, have been struggling to be barely adopted. Among the factors that contribute to such hostile environment for innovation are: *(i)* the necessity of deep and global modifications in order to adopt new solutions, *(ii)* the slow standardization process required to grant interoperability with legacy protocols, *(iii)* the dependency on the profit-oriented interest of network equipment vendors, and *(iv)* the tendency of network core devices on using proprietary software and hardware, ruling out any possibility of customization from End-Users.

With the promise to reverse this state of ossification, the research community is investing hard on the concepts of Network Virtualization [2] and Programmable Networks [3]. Both concepts are merged together to provide sets of virtual networks with different behaviors, forming the so-called Programmable Virtual Networks (PVN) [4]. One of the approaches adopted to deploy PVN follows the format of Software Defined Networking (SDN) [5], which defines the decoupling of control and data planes. Some implementations of the control plane employ the Service Oriented Architecture (SOA) to provide high level services through a standardized interface. Thus, network applications can be programmed in different languages, becoming less dependent on the technology used in the control plane.

On the one hand, the decoupling of network applications, control plane, and data plane in SDN architectures, has shown to be flexible and scalable [6]. On the other, it induces a great management complexity. Management models used in common networks are not suitable for programmable virtual networks since they do not deal with the dynamic deployment of new network services. Moreover, once virtual networks tend to be more open and innovation friendly, in a near future End-Users may be able to implement and deploy their self-developed network applications in order to attend to their specific demands. Nevertheless, harmonizing applications, users and virtual networks, while maintaining the reliability and scalability of services deployed is an open issue.

PVN management proposals vary according to the deployment environment and the requirements of users. Shared experimental facilities (testbeds) usually employ proposals focused on slice provisioning, such as ProtoGENI [7] and OFELIA Control Framework [8]. However, the management of network programmability upon the slices is not provided or is limited. Similarly, market oriented solutions typically found in Cloud Computing environments, such as XenServer Distributed vSwitch Controller (CITRIX) and OnePK (CISCO), are intended to provide network programmability control for Cloud providers, not for End-Users. Instead, we believe that in order to achieve higher rates of innovation attending to new virtual network specific demands, it is important to include the End-User in the process of creation and deployment of novel network applications, turning this process more democratic.

This paper tackles the research problem of how to manage PVN infrastructure, leveraging its capabilities to provide programmability to End-Users and fostering innovation in this context. With this in mind, we propose the **Pro**grammable **Vi**rtual **Net**work (ProViNet) management platform. ProViNet introduces the concept of Network Programming as a Service, in which applications are graphically composed simply by dragging and dropping control plane services. Such approach encourages End-Users to develop and share novel network solutions, promoting innovation. We have implemented a prototype to demonstrate the feasibility of the concepts proposed within ProViNet platform. Also, a case study is presented in this paper to illustrate the applicability and benefits that can be achieved by employing the proposed approach.

The rest of the paper is organized as follows. In Section II, we outline the main scientific papers that address PVN man-

0730-3157/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSAC.2013.58

agement and a overview of background concepts. In Section III we present ProViNet platform, discussing the conceptual architecture and concepts used. The ProViNet prototype is detailed in Section the IV and evaluated in Section V. Finally Section VI concludes the paper with final remarks and perspectives for future work.

II. Related Work and Background

The concepts of Network Virtualization [4] [2] and Network Programmability [9] [10] have been well discussed in the literature. From the joint application of these concepts emerge Programmable Virtual Networks (PVN). PVNs are most commonly applied in two environments: (*i*) testing platforms, also known as testbeds, and (*ii*) private clouds. The studies presented in this section are organized according to these two environments, aiming to emphasize their level of abstraction of virtual infrastructure, the approach used to provide PVN support, and the usage license.

Among the proposals under testbed environments, we highlight the OFELIA Control Framework (OCF) [8], which is specifically focused on providing network programmability based on OpenFlow technology [11]. This framework is a derivation of the Expedient [12] platform proposed by Stanford University and assists researchers in creating slices using resources from several federations. Additionally it also offers researchers the ability to associate these slices to previously configured OpenFlow controllers. Despite this functionality being available through the Graphical User Interface (GUI), the OCF does not provide management capabilities for the deployment of network applications, considered a major concern in network virtualization [2].

Another proposal, still in the context of testbeds, was created in GENI project [13] and is called ProtoGENI [7]. This proposal also implements a Web interface to provide researchers with the ability to create slices with resources from different federations. When interacting with the GUI of ProtoGENI, researchers can instantiate virtual nodes and connect them dynamically. Like most proposals developed in the context of testbeds, the focus is on providing the slice, not on its programmability. In general these solutions are free to use for academic purposes, however there tend to be much bureaucracy and strict rules for the use and access to resources provided.

Solutions in the context of Cloud computing differ from testbeds, mainly due to their commercial focus. Cloud environments generally require large amounts of network resources in order to provide services with high availability rates and quality of service. Therefore, the creation of customized network services aiming to meet special demands currently draws the attention of Cloud providers. To comply with these demands, solutions have emerged to increase the customization of services provided through network virtualization in datacenters. Some of commercial solutions already exist, such as Nexus 1000v and CISCO OnePK and the Distributed Virtual Switch Controller (DVSC) from CITRIX. Some other solutions are open source, such as Open vSwitch, Ryu, and restproxy plugins for the OpenStack platform [14].

In both Cloud and testbed environments, there is an increasing number of proposals employing concepts of SDN architecture to provide programmability solutions in virtual networks. Rubio *et al.* [15], for example, proposed an orchestration plane, which is complementary to the originally defined control and data planes in SDN architecture. The purpose of this new plane is to dynamically control the behavior of virtual networks in response to constant variations, thus generating an autonomic management system. Although this system has advantages such as quick response to failures, it requires standardized and well tested solutions. As a result, the End-User is still discouraged from creating and deploying their own applications in programmable virtual networks. This fact contributes to the low rate of innovation in networks, since it poses restrictions and keeps the task of developing and deploying network-oriented solutions only possible to a small number of people.

In summary, although there are recent proposals involving programmable virtual networks, none of the aforementioned studies promotes the management of programmability in a PVN infrastructure, considering the ease of access by multiple End-Users. Moreover, most of the proposals analyzed are limited to provisioning of virtual resources (controlling and configuration of virtual machines and virtual networks), not providing functionality for managing network applications that can be installed on these programmable virtual networks dynamically.

A. Background

In a virtualized environment, a variety of virtual machines are interconnected by a virtual network. This virtual network can be provided in different ways (such as VLAN, VPN, and application level overlay [4]) depending on the underlying technologies supported by the infrastructure. Whatever the approach taken is, the result is that users will have their isolated virtual networks interconnecting their own virtual machines, though they all share the same physical infrastructure. It is convenient and usual referring to the set of network, computing, and storage resources belonging to a user as *Slice*.

Current Cloud providers (public or private) are not capable of setting custom virtual network topology for Slices. At most, they are able to allocate virtual machines in different broadcast domains, forming the so-called tenants. By contrast, such limitation does not occur in testbeds, where the user is able to define exactly the topology of the Slice. Our solution integrates the flexibility of testbeds to the Cloud organization model. Such approach makes possible the complete migration of a physical computing set up, including the hosts and the network topology. Considering that ProViNet works over programmable networks, the network services in the physical network could be easily migrated as a network application, such as NATs, firewalls and load balancers. As will be presented in the Section III.

We argue that to promote the emergence of innovation, it is necessary higher accessibility and availability of programmable resources. In doing so, the more available and accessible are the resources, the more users will be able to develop, deploy and share novel network solutions.

III. Conceptual Solution

As shown in Figure 1, unlike other solutions found in common PVN scenarios, such as testbeds and some Clouds deployments, the main focus of ProViNet is End-Users. Nothing prevents other types of users from adopting ProViNet, but there are already several solutions focused on attending their specific demands. Differently from these solutions, ProViNet aims at providing the necessary tools to enable users with less technical knowledge to create their own network solution. Our challenge is to find a good trade off among simplicity and granularity (or programmability level). A fine grained programming approach may require much technical knowledge, which induce complexity. However, if we provide much abstractions, promoting simplicity, it may generate limitations to novel network applications.

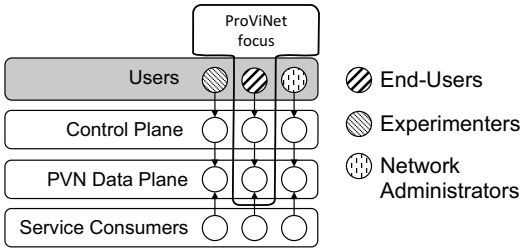

Figure 1. ProViNet provides End-Users access to the Control Plane

The four layers depicted in Figure 1 represent Software Defined Networking architecture, which defines that control and data planes are decoupled. Such separation represents a great opportunity to enable End-Users to control the behavior of networks, because control plane come closer to network edges. Exploring this characteristic, ProViNet is able to manage network applications and their relationship with PVN slices. In this opportunity, we highlight that our target public has no special permissions or privileges other than the inherent to its own interests. We call this target public End-Users. It is very important to emphasize that End-Users may encompass the most Internet users, including software developers, network engineers, or any network enthusiast interested in developing its own virtual network services. Occasionally, it may occur that ProViNet End-Users may not be the same of network service consumer, though our intention is to enable also the consumer to develop its services.

We put experimenters outside the focus because researchers may not found the high flexibility required in research tasks. Moreover, there are a plenty of testbed solutions that focus on providing them the according experimentation support.

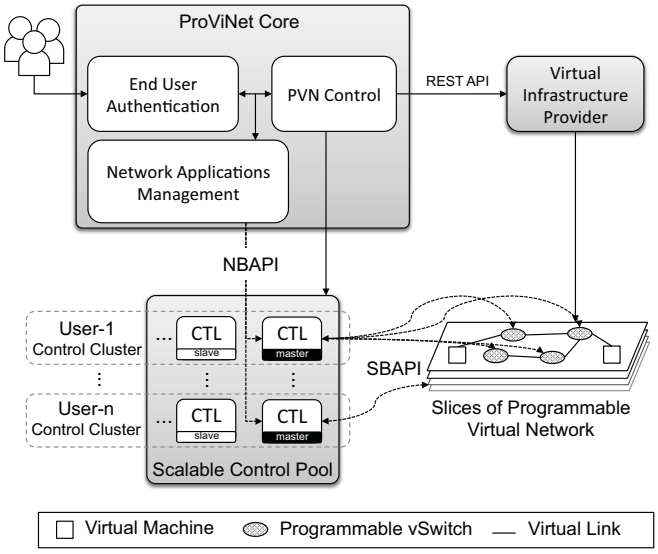

Figure 2. Conceptual Architecture

A. Conceptual Architecture

The architecture shown in Figure 2 illustrates the conceptual components of ProViNet platform as well as their high level relationships. In fact, it presents four components: *ProViNet Core*, *Virtual Infrastructure Provider* (VIP), *Scalable Control Pool*, and finally *Slices of Programmable Virtual Network*. All components are decoupled and the communication between them is provided by well-known protocols.

In the context of SDN, the terms Southbound API (SBAPI) and Northbound API (NBAPI), presented in the architecture, have been recently used for referring to the communication protocols that enable respectively: the interaction between control-plane and data-plane and between control-plane and network applications. Although there is no consolidated standard on either API technologies, in practice most SDN deployments use OpenFlow as SBAPI and Representational State Transfer (REST) based protocols as NBAPI.

In the proposed architecture, the NBAPI is employed in the requests from network applications and the master Controllers (*CTL*). The former is managed by *Network Application Management* module, while the second is part of the *Control Clusters*. A *Control Cluster* is created in the *Scalable Control Pool* for each End-User and provides high availability to the control plane. In turn, the *CTL* attends the End-User service requests by exchanging SBAPI calls with the Slice.

ProViNet Core was developed as a Web application. Therefore, the End-User interacts with the platform through a Web interface that graphically exposes the functionalities provided by ProViNet modules. As illustrated in Figure 2, there are three modules. The *End-User Authentication* module handles access and authentication required to use the platform. Once the user is authenticated, this module controls the authenticity of inter-modules communication. In the next section we present the VIP and its relationship with *ProViNet Core*. Then, except

for the *End-User Authentication* module, which behavior is quite straightforward, all others are detailed in the following subsections.

B. Virtual Infrastructure Provider

ProViNet Core is not responsible for *Slice* provisioning, which includes the tasks of allocating virtual machines and configuring virtual networks according to physical infrastructure capacity. As illustrated in Figure 2 such tasks are delegated to a *Virtual Infrastructure Provider (VIP)*. The VIP we use in the current development of ProViNet runs over a platform proposed and implemented in a previous work of ours [16], which is actually the only private Cloud platform capable of supporting the custom programmable virtual network topology required by ProViNet. This platform is able to configure a network topology of virtual switches and configure them to receive control information from external controllers.

ProViNet Core communicates with the VIP through Web Service calls, on which authentication data and an XML file with a virtual infrastructure description are transmitted. There are several models for elaborating this infrastructure description file, such as Rspec (GENI), NDL-OWL (RENCI), NMC (OGF) and Virtual Resources and Interconnection Networks Description Language (VXDL) [17]. We chose to use the last one (VXDL) in ProViNet because such language is able to represent in simple and clean XML all the details of a virtual infrastructure, including virtual machines, switches, and links between them forming the topology. In VXDL it is still not possible to determine the controllers that must be associated with the virtual switches present in the virtual topology. In order to better understand this language, we highlight the following sample of VXDL representing a simple topology of two machines, two switches, and three links. One link from each machine to different switches and one link connecting the two switches:

```
<virtualInfrastructure id="SimpleTopo" owner="admin">
  <vNode id="Node1">
    <cpu>...</cpu>
    <memory>...</memory>
    <storage>...</storage>
    <image>...</image>
    <interface>...</interface>
  </vNode>
  <vNode id="Node2">...</vNode>
  <vRouter id="Switch1">
    <controlPlane layer="" routingProtocol="" type="">
    </controlPlane>
  </vRouter>
  <vRouter id="Switch2">...</vRouter>
  <vLink id="Link1">
    <bandwidth>...</bandwidth>
    <latency>...</latency>
    <source>
      <vNode>Node1</vNode>
      <interface>net0</interface>
    </source>
    <destination>
      <vRouter>Switch1</vRouter>
    </destination>
  </vLink>
  <vLink id="Link2">...</vLink>
  <vLink id="Link3">...</vLink>
</virtualInfrastructure>
```

We extended the default VXDL specification with a new tag *controllerList* once most SDN switches accept the configuration of one master and many slave controllers. Moreover, note that we included an attribute `type=""` in the `controller` tag, which is used to inform whether the controller in the list is master or slave. In addition, each controller also uses a specific port, protocol, and IP address to communicate with the switches, thus tags for this elements were also added. Taking the same sample of VXDL used before, the configuration of the two controllers would fit like:

```
...
<vRouter id="Switch1">
  <controlPlane layer="ETHERNET"
    routingProtocol="OpenFlow" type="dynamic">
    controllerList1</controlPlane>
</vRouter>
<vRouter id="Switch2">
  <controlPlane layer="ETHERNET"
    routingProtocol="OpenFlow" type="dynamic">
    controllerList1</controlPlane>
</vRouter>
<controllerList id="controllerList1">
  <controller type="master">
    <connection_type>tcp</connection_type>
    <ipAddress>xxx.xxx.xxx.xxx</ipAddress>
    <port>6633</port>
  </controller>
  <controller type="slave">...</controller>
</controllerList>
...
```

C. Programmable Virtual Network Control

According to the sequence diagram shown in Figure 3, after that the *End-User* has prepared the VXDL file, he/she is able to upload it using a Web form in the ProViNet GUI. Along with the VXDL specification, the *End-User* is able to set a redundancy level. This level represents the number of controllers that will form a hight available control plane for the *Slice* described in the VXDL file, called *Control Cluster*. Note that the VXDL uploaded by the user contains only information about the *Slice* (*i.e.* nodes, switches, links, and so on), ProViNet adds all the controller information automatically.

The controllers are created by remote calls sent to the *Scalable Control Pool*, which is in fact a set virtualization servers (hypervisors). There is in this Pool, a virtual machine configured in advance with the software of a controller technology. So, by sending a cloning call to the Pool, it creates a copy of such virtual machine. Regarding the controller technology, there are no great restrictions, other than supporting Web services, once the configuration calls to the controller will be sent remotely. In the current ProViNet implementation, we deployed FloodLight controller because it support REST calls and has a plenty of services available.

Still following the diagram, when the PVN Control module receives the controller(s) information, it begins the process of editing the VXDL file, adding the entries to represent the controller(s) created, such as IP address, connection type, port, and whether the controller is master or slave. After

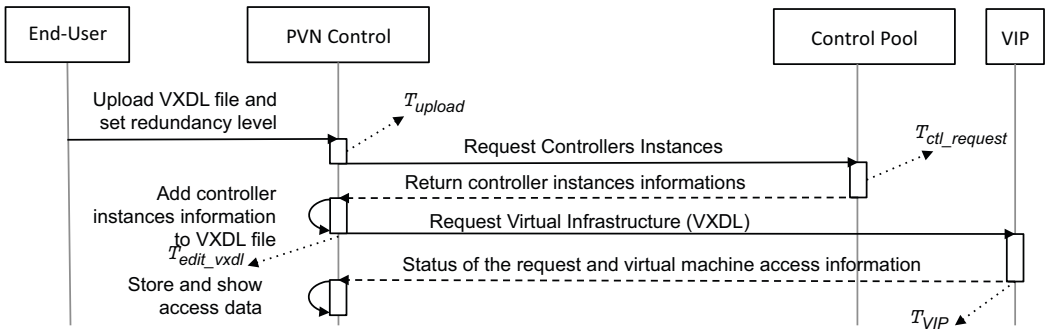

Figure 3. Sequence Diagram of Slices Provisioning Procedure

that, the file is sent to the VIP via Web service request, automatically generated by the platform. The provisioning of the virtual infrastructure is under responsibility of the VIP, which will also configure the vSwitches to forward control plane information back to controllers at ProViNet's *Scalable Control Pool*.

When the response is received from the VIP, informing that the virtual infrastructure is properly provisioned, ProViNet stores it in an internal database and show in the GUI interface the information related to accessing resources in the *Slice*. In order to diminish complexity, such information is graphically represented and allows the End-User to interact via browser with the virtual machine instances of the Slice.

D. Network Applications Management

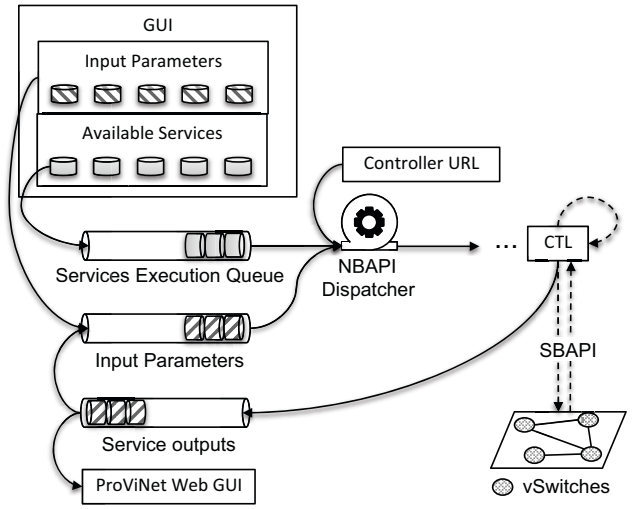

Figure 4. Queuing approach for a user-friendly Network Programming

Once the PVN is established, *i.e.*, the user has a Slice and a control plane properly configured, the next step is to develop and run network applications. Starting by the development, the approach we propose is similar to services composition concept. As depicted in Figure 4, all the services available by the controller technology are listed in the ProViNet GUI interface. Interacting with such interface, the End-User is able to order services into a *Services Execution Queue*.

The NBAPI Dispatcher, an element of Network Application Management module, consumes the services in the queue. The *Dispatcher* role is to send HTTP requests to the master controller of the users' *Control Cluster*. The address of such controller (URL) is available in ProViNet internal database after the execution of the process previously described and illustrated in Figure 3. In its turn, the controller attends each service request exchanging SBAPI calls with the vSwitches in the Slice, or consulting its internal system.

Some services require input parameters, which can be defined using the input fields in the interface or redirected from the outputs of a previous service in the queue. After each service execution, the results can be displayed in the *ProViNet Web GUI*. To do this, the user must set *ProViNet Web GUI* as the output destination.

Each *Services Execution Queue* may implement a network application, or specific for the slice topology described in the VXDL file, or generic to run over any topology. In order to better understand how a ProViNet network application looks like, we present below a very simple Firewall implementation. Worth mentioning that all programming procedure occurs through the Web GUI. The example is presented using XML format because it may represent Web GUI states.

The objective of the Firewall example is to instruct a specific switch to drop all the packets having port 22 as destination. To that end, the services *getTopology* and *setFlow* must be queued. The first will retrieve the switches IDs, which appear in the input parameter options list of *setFlow* service GUI representation.

```
<service id="getTopology">
  <input></input>
  <output><switch>...</switch>...</output>
  <description></description>
</service>
<service id="setFlow">
  <input>
  <switchList></switchList>|<switch></switch>
  <match></match>
  <actions></actions>
  <priority></priority>
  </input>
  <output><string></string></output>
```

```
  <description></description>
</service>

<executionqueue id="BorderFirewall">
  getTopology, setFlow
</executionqueue>
<inputparameters id="BorderFirewall">
  <none />,
  <get_prev_output>
    <switch id="1"></switch>
  </get_prev_output>
  <match>tp_dst=22<match>
  <actions>output=<actions>
  <priority>64900</priority>
</inputparameters>
<serviceoutputs>
  <switch id="1"></switch>... ,
  <gui><string>Flow set successfully!</string></gui>
</serviceoutputs>
```

The first two root tags shows the services specification, where are described inputs, outputs and a briefly description of the services. The third represents the *Services Execution Queue*, which uses comma to separates the services ids in the execution sequence. Then, comes the input parameters queue, where is set no parameter for the first service, and four for the second, as specified in the services specifications. And finally the service outputs queue, showing the output of the *getTopology* service and the output of the *setFlow* service.

The approach of network programming described in this section has the main objective to be simple, so End-Users can easily create novel network applications. As mentioned in the beginning of this Section, our challenge is to provide a simple approach without loosing granularity or level of programmability. Queuing services is a simple task and may not represent much complexity to the End-User. Regarding the granularity, it is directly related to the services available by the controller technology. For instance, with the current technology deployed, which is Floodlight, the available services allow the development of applications to control network from Physical Layer (L1) to Transport Layer (L4).

IV. Prototyping ProViNet Platform

Aiming to demonstrate the feasibility of the conceptual architecture detailed in Figure 2 Section III, we developed a prototype. Our prototype implements the three components of ProViNet Core, the *End User Authentication*, *PVN Control* and *Network Application Management*. These components were implemented using framework Django 1.4.3, Python 2.7.3 and PostgreSQL 9.1.6 database.

The End User Authentication handles login and registration features of ProViNet. The PVN Control component has all the routines and APIs necessary to communicate with Virtual Infrastructure Provider and with the Scalable Control Pool (SCP). The SCP was deployed with XenServer [18] and the communication with this server were enabled by XenAPI, provided by CITRIX. Using this API, remote systems is able to control a Pool of XenServers.

One use case of this controlling action, is when users request virtual infrastructure and select a redundancy value. A loop will iterate calls of clone function (session.xenapi.VM.clone(vm_base_name, vm_cloned_name)) as much as that redundancy value. When XenServer receives cloning request, with the name of a previous configured VM, it clone the VM matching that name. In our implementation, we previously configured a virtual machine with an installation of FloodLight, which was configured to start automatically after the OS boot. When the VM at XenServer is finish booting, PVN Control does another call to retrieve IP address of the VM just cloned. Such address is added to VXDL file using a Python library called xml.etree.ElementTree.

The second communication of PVN Control component is performed with Virtual Infrastructure Provider. After the upload of VXDL, the VM cloning procedure, and VXDL editing, PVN Control parse VXDL to a encoded string. Such string and additional authentication data are used as parameter of a HTTP POST request targeted to VIP. After properly instantiation and configuration of the virtual infrastructure, VIP sends back a response. In case of failure, the cause of failure is specified.

Finally, the Network Application Management (NAM) component was developed using python routines, which, in its turn integrates with the templates of Django framework. Such template was built using WireIt javascript library, which enables to graphically connect boxes using wires. Each composition represents a network application that is runnable by an engine also developed. Such engine is in the Django view that handles template events. So, when the ”run” button is hit, a execution routine iterates the connections and boxes (which represents services) filling the execution queue. Then, the response of each service is presented in a terminal viewable by the user.

We choose the FloodLight as the controller implementation, because it provides an API allowing restfull consumption of services available in the controller via HTTP calls. Other implementations could be used, since it support remote calls of network applications.

In order to give a better overview of our prototype we recorded a demonstration video. The video as well as detailed information are available online[1]. Our prototype source code and documentation are found at Github with the name ProViNet.

V. Experimental Verification of ProViNet

Firstly in this section, we present a qualitative comparison with the proposals presented earlier in the Section II. Then, in order to illustrate an applicability of our solution, we describe an use case, which underlies the performance analysis presented later. The qualitative comparison demonstrates the uniqueness of ProViNet, and the performance analysis illustrates that the prototype developed to prove the concept performs its tasks with acceptable amounts of time.

A. Qualitative Comparison

Considering the End-User point of view, we characterized some features that we consider essential to make network

[1]http://www.futureinternet.br/index.php/independent-research/32-provinet

Feature	**ProViNet**	**OFELIA Control Framework**	**ProtoGENI**	**CITRIX DVS**	**OpenStack+Quantum**
End-User Access Policy	Marketable Access	Restricted Access	Restricted Access	Restricted Access	Marketable Access
Resource Organization Method	Data Center	Federations	Federations	Data Center	Data Center
Network Topology	Physically Independent	Physically Dependent	Physically Independent	Virtually Limited	Virtually Limited
Resource Description	VIDL Compatible (VXDL)	VIDL Not Compatible	VIDL Compatible (RSPEC)	VIDL Not Compatible	VIDL Not Compatible
Resource Request Method	At Once Request Submission	One-by-one Requests	Both	One-by-one Requests	One-by-one Requests
Granularity or Programmability Level	L1 to L4	L1 to L7	L1 to L7	L2 to L4	L1 to L7
Control Plane Management	High Level	Middle Level	Low Level	High Level	Low Level
Target Public	End Users	Researchers	Researchers	Cloud Operators	Cloud Operators

Table I
COMPARISON OF RELATED PROPOSALS

programming simpler and manageable. In what follows we describe such features and then, in the Table I we present the remarks of each solution. The analysis of the proposals considered the on-line available material, such as white papers, tutorials, videos and softwares.

End-User Access Policy: Regarding End-User access limitations, network environments are classified in: *Unaccessible*, such as the network core. *Restricted Access*, as the testbeds, where users need to request permissions that are usually given after a good justification letter. And, finally, the access can be with *Marketable Access*, *i.e.*, the user is able to freely acquire virtual resources, similar to the approach performed in Cloud Computing deployments.

Resource Organization Method: Different infrastructures could be used to instantiate virtual networks and virtual machines. Generalizing, such resources could be from Data Centers, Universities or Research Laboratories. The testbeds, for instance, adopts a pluggable architecture, in which Research Laboratories and Universities can follow a technical pattern to share its physical resources within a global deployment. Then we generalize the proposals classifying them in *Federation* or *Data Center* organization method.

Network Topology: Some proposals allow users to require virtual network and virtual machines disposed in a custom topology, totally independent of the physical substrate. Others proposals provides virtual network topologies limited to the physical nodes and links. At last, there are proposals that despite being independent of physical topology, it is still limited to one kind of topology. Concerning this feature the proposals can be *Physically Dependent*, *Physically Independent* or *Virtually Limited*

Resource Description: There are some standard Virtual Infrastructure Description Languages (VIDL) that may be used to exchange virtual infrastructure informations among different proposals. The use of VIDL may turn the solution compatible to different deployment environments. Moreover, during the definition of the network topology, the End User may take advantage of third party applications that offers a graphical front end to facilitate the building of VIDL files. The proposals are classified whether *VIDL Compatible* or *VIDL Not Compatible*.

Resource Request Method: An End User interested in having a virtual infrastructure may found easier to graphically draw the Slice before committing its request. However, graphically drawing Slices has the drawback of one-by-one definition, what may be a problem in larger slices. If the user can just send the whole slice definition at once, using a VIDL file, it may be faster. Of course there may be a previous moment when the user prepare the file, but it could be easily automatized. So the proposals may vary in *One-by-one Requests*, *At Once Request Submission* and *Both* support.

Granularity or Programmability Level: The operational scope of network applications may be different among each solution. Usually higher programmability levels comes with higher complexity. Depending on the programmer experience and knowledge, higher levels of granularity would allow broader programming scopes. From the standpoint of End Users, the programming procedure must be simple but with enough granularity to build groundbreaking network applications. The proposals can allow from *L1* to *L7* programming.

Control Plane Management: In SDN architecture, the control plane runs out of switch box, in an external machine (called controller). This controller could be placed in a physical or virtual machine, since it has network access to the programmable switches under its control. Furthermore, most of switches implementation provides initial support for High Availability (HA) by accepting configuration of more than one controller. Then, in case of ones failure the other can assume the control master role. Most of proposals leaves the control plane management to the users. However, if such control is done by the platform, the user can concentrate on developing the network applications. Considering the exposed, we classified the proposals by its levels of management. If the proposal support HA, automatized creation of controller

instances and automatized configuration of the switches we say it has *High Level* of control plane management. If it support just two of the mentioned management features, it has *Middle Level* and if just on is supported it has *Low Level*.

Target Public: In general, the proposals were developed to attend specific users requirements. The testbeds, for example, intend to provide experimental resources to *Researchers*. Conversely, Cloud providers tend to develop proposals of network programmability having *Cloud Operators* as the main costumer. And finally, solution like ProViNet focus in the *End-Users*. The Researchers are subset of End-Users, see definition in the Section II-A

Among the proposals presented in the Table I, OFELIA Control Framework, CITRIX DVS and ProtoGENI were presented in the section II. In addition to them, we compare ProViNet to a generically solution that may employ OpenStack and Quantum technologies. The former is a platform for managing Cloud environments, and Quantum is one module that enable the installation of third party plug-ins to control the virtual network specifically. These proposals represent the state-of-art in the matter of programing virtual networks.

Analyzing the comparison presented, we can notice that only ProViNet and OpenStack plus Quantum has marketable access to End-Users. It can also be noticed that ProViNet and CITRIX DVS are the unique to provide the three functionalities described earlier and so remarked as High Level on Control Plane Management. Considering the features altogether ProViNet is the most complete.

B. Case Study and Performance Analysis

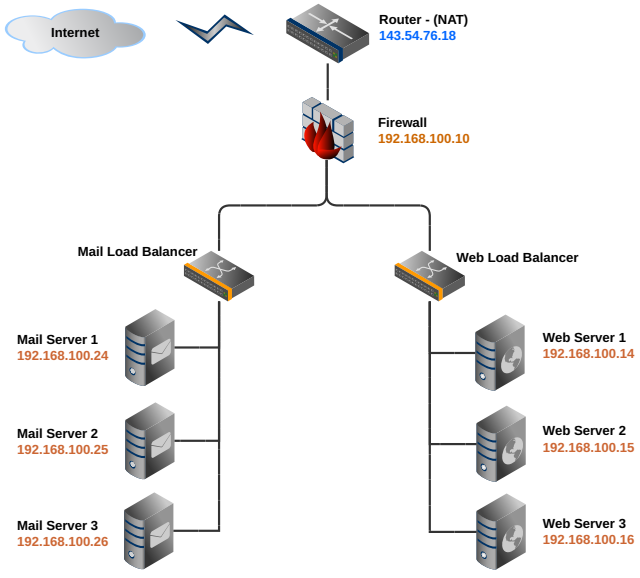

Figure 5. Use case scenario of physical network migration

In order to evaluate the prototype described in Section IV, we elaborate a case study. In this case study we will analyze a typical situation, in which, due to economic factors, network administrator was requested to migrate a physical network set up to the Cloud. Considering migrate all functionalities together, *i.e.* Firewalls, Load Balancers, NAT and so on. The physical infrastructure mentioned is depicted in Figure 5. It is composed by two load balancer appliances, one Firewall appliance, one border switch with NAT, three Web servers, three Mail servers and ten links. Using such case study we evaluated two main procedures performed by ProViNet: PVN Provisioning and Network Programming.

The measurements of both evaluations was taken over a physical scenario with a physical machine Intel Xeon E3-1220 3.1GHz CPU, 4GB RAM, configured with XenServer 6.1, representing the Scalable Control Pool. As already mentioned, in the current ProViNet deployment, OpenFlow controller technology used is Floodlight v0.90. Such technology was deployed in a Virtual Machine Ubuntu 12.04 with 1 vCPU and 384MB RAM pre-configured in XenServer. To run ProViNet Core we used a laptop Intel Core i7 2.8GHz and 4GB RAM.

The process of PVN Provisioning starts by translating physical topology just described in Figure 5 in a virtual topology following VXDL grammar. It will be similar to the example already given in Subsection III-B. The appliances will become common switches in virtual topology, once in a programmable virtual network, applications are outside the switches. The following steps are represented in the sequence diagram depicted in Figure 3.

Considering the sequence diagram mentioned, we refer to the time taken for upload VXDL as T_{upload}, and to the time spent creating controller instances as $T_{ctl-request}$. Meanwhile, the time for adding controller information in the VXDL file is represented as $T_{edit-vxdl}$, and finally the time taken by the VIP to instantiate virtual resources as T_{VIP}. In doing so, the time related to the whole process of provisioning virtual infrastructure, including configuration of control plane, is referred to as T_{total} and represents the sum of the others, so $T_{total} = T_{upload} + T_{ctl-request} + T_{edit-vxdl} + T_{VIP}$. All values presented in Table II refers to the average time of 30 executions, in seconds, taken to conclude each procedure. Considering 95% confidence, the standard deviation obtained was irrelevant for the values presented. We considered that the End User required a redundancy of two controllers.

It is noteworthy that the time taken by VIP to provide virtual infrastructure is not under control of ProViNet, but we still consider it to give an idea about overall performance or our solution. The time taken by VIP includes the tasks of instantiating virtual machines, creating vSwitches, creating links and connecting virtual resources. Considering that the infrastructure required, has six virtual machines, four vSwitches and ten links, the time of 43.0345 seconds is a good remark.

The most significant time presented in the table is $T_{ctl-request}$, which is directly dependent of the hypervisor performance to clone and start two virtual machines. From the 49.6581 seconds average, 23.0554 seconds were taken just for cloning and 26.4257 seconds for starting. ProViNet needs to wait virtual machine booting, once just after that a valid IP is given to the VM and can be added in VXDL file.

T_{upload}	$T_{ctl-request}$	$T_{edit-vxdl}$	T_{VIP}	T_{total}
0.029394	49.6581	0.038839	43.0345	92,7608

Table II
PROGRAMMABLE VIRTUAL NETWORK PROVISIONING

The next step in the migration use case is developing network applications that will control the switches. Accessing ProViNet Web GUI the user will create the execution queue with the necessary services, which will follow the same logic of the example given in Section III-D. Once the queues of Firewall, Load Balancer and NAT is ready it triggers the *NBAPI Dispatcher*, the component under study. To measure the performance of this component we ran three types of services. One for adding a flow in a switch, another to read flow, and the last to delete flow. After 30 executions, the average times between sending requests (HTTP GET or POST) and receiving answer are presented in Table III.

Request	Average Time
Add Flow	0.148070
List Flow	0.061958
Delete Flow	0.124699

Table III
NBAPI DISPATCHER PERFORMANCE

VI. FINAL REMARKS AND FUTURE WORK

As a consequence of the successful Cloud Computing architecture, with elastic and on-demand resources, there is a large adoption of virtual networks. The flexibility of Cloud architecture enable the use of its resources to address a diversity of client demands. Furthermore, the network below such environments has a great advantage over the network core; it is much more susceptible to changes. At least, the consequences have smaller proportions as it would have in network core. However, the development of specific network solutions to attend novel and divergent client demands are no longer role of network equipment manufacturers. Solutions compatible with virtual and physical switches may fit best the requirements of Cloud providers. Network solutions developed in the past, and implemented in accordance with the wishes of network device manufacturers, may not withstand novel and divergent client demands. In this new context, the emergent SDN concept came to the spotlight as a promising solution.

At this point, it is convenient to elucidate that the four issues listed in Section I can be seen as perennial challenges on computer networks. They have been addressed by programmable network proposals since 90's, when many works where published in Forums such as OPENARCH [19] and OPENSIG [20]. In this paper we proposed ProViNet management platform, an attempt to advance the research toward management of SDN on virtualized environments. It worth noting that the solution presented in this paper does not aims at being a novel programmable network proposal. Instead, it leverages SDN capabilities and strengths to build a resilient (with high availability in Control Plane), end-user driven, and aligned with ONF recommendations (such as using north and south bound APIs).

SDN proposes the separation of the control and data planes, causing an approximating of the control point to the network edge. In this place it can be better managed to run novel network applications. Meanwhile, management models used in common networks are not suitable for programmable virtual networks since they do not deal with the dynamic deployment of new network services. By the use of the platform proposed we aims at encouraging the creation of a new business model in networks, specially virtualized ones. In such model called Network Programming as a Service (NPaaS), the client would be able to develop and deploy its own virtual network.

Regarding the network application development process itself, we presented in this work, a user friendly approach for programing virtual networks. Employing concepts of Web Service composition we elaborate an approach in which the network programmer (role assumed by end-users or any interested) can queue a sequence of northbound API services in order to control previous provisioned virtual Slice.

In a wider point of view, at a first moment, ProViNet wouldn't be the best choice to manage programmability in the Internet core. Such environments are not mature enough to provide network control to end-users. Then, the scope of deployment would depend on Virtual Infrastructure Provider, which, as presented, has the role of providing isolated layers of programmable virtual networks. The best scenario would be private or public Cloud, which would benefit with NPaaS business model.

In order to demonstrate the feasibility of the concepts proposed within ProViNet platform. We have implemented a prototype and presented a case study to illustrate the applicability and benefits that can be achieved by employing the proposed approach. We also presented a qualitative comparison of ProViNet with other four proposals, highlighting the uniqueness of our solution and letting clear our contributions.

As future work we intend to improve the network developing approach with mature techniques of services composition such as BPMN and BPEL. This would allow a compatible services aggregation, having as a consequence, the possibility to mix network and general Web Services. In a specific example, a developer could set the system to send an email after a network state changing. Moreover, it would be possible to use already established running engines, which may provide validations of parameters and check for inconsistencies.

VII. Acknowledgments

The authors would like to thank the CNPq (National Counsel of Technological and Scientific Development - Brazil) and FAPERGS for financial support.

References

[1] D. Hausheer, A. Parekh, J. Walrand, and G. Schwartz, "Towards a compelling new internet platform," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, may 2011, pp. 1224 –1227.

[2] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Comput. Netw.*, vol. 54, no. 5, pp. 862–876, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2009.10.017

[3] Y. Kanaumi, S. Saito, and E. Kawai, "Deployment of a programmable network for a nation wide randd network," in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, april 2010, pp. 233 –238.

[4] N. Chowdhury and R. Boutaba, "Network virtualization: state of the art and research challenges," *Communications Magazine, IEEE*, vol. 47, no. 7, pp. 20 –26, july 2009.

[5] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[6] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: a slice abstraction for software-defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 79–84. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342458

[7] "protogeni," 08 2012. [Online]. Available: http://www.protogeni.net/trac/protogeni

[8] W. Kopsel, "Ofelia - pan-european test facility for openflow experimentation," 11 2011. [Online]. Available: http://www.fp7-ofelia.eu/

[9] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, "A survey of programmable networks," *COMPUTER COMMUNICATION REVIEW*, vol. 29, pp. 7–23, 1999.

[10] P. Lin, J. Bi, H. Hu, T. Feng, and X. Jiang, "A quick survey on selected approaches for preparing programmable networks," in *Proceedings of the 7th Asian Internet Engineering Conference*, ser. AINTEC '11. New York, NY, USA: ACM, 2011, pp. 160–163. [Online]. Available: http://doi.acm.org/10.1145/2089016.2089044

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[12] Expedient, "Expedient: A Pluggable Platform for GENI Control Frameworks [Online]," 04 2012. [Online]. Available: https://alpha.fp7-ofelia.eu/doc/index.php/Working_with_the_OFELIA_Control_Framework

[13] GENI, "Global environment for network innovations," 06 2011. [Online]. Available: http://www.geni.net/

[14] OpenStack, "Open source software for building private and public clouds," 2011, available at: http://www.openstack.org/. Last access in: Julho 2012.

[15] J. Rubio-Loyola, A. Galis, A. Astorga, J. Serrat, L. Lefevre, A. Fischer, A. Paler, and H. Meer, "Scalable service deployment on software-defined networks," *Communications Magazine, IEEE*, vol. 49, no. 12, pp. 84 –93, december 2011.

[16] J. A. Wickboldt, L. Z. Granville, F. Schneider, D. Dudkowski, and M. Brunner, "A new approach to the design of flexible cloud management platforms," in *8th International Conference on Network and Service Management (CNSM)*, Las Vegas, USA, October 2012, pp. 155–158.

[17] G. P. Koslovski, P. V.-B. Primet, and A. S. Charao, "Vxdl: Virtual resources and interconnection networks description language," ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, P. V.-B. Primet, T. Kudoh, and J. Mambretti, Eds., vol. 2. Springer, 2008, pp. 138–154.

[18] "XenServer Administrator's Guide 5.5.0," Citrix Systems, Tech. Rep., Feb. 2010. [Online]. Available: http://support.citrix.com/servlet/KbServlet/download/20636-102-427354/reference.pdf

[19] "1999 ieee second conference on open architectures and network programming," in *Open Architectures and Network Programming Proceedings, 1999. OPENARCH '99. 1999 IEEE Second Conference on*, 1999, pp. i–.

[20] A. T. Campbell, I. Katzela, K. Miki, and J. Vicente, "Open signaling for atm, internet and mobile networks (opensig'98)," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 1, pp. 97–108, Jan. 1999. [Online]. Available: http://doi.acm.org/10.1145/505754.505762