UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

WANDERSON PAIM DE JESUS

Network Programming as a Service: An Innovation Friendly Business Model

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Lisandro Zambenedetti Granville
Advisor

Porto Alegre, Jan 2014

CIP – CATALOGING-IN-PUBLICATION

Jesus, Wanderson Paim de

Network Programming as a Service: An Innovation Friendly Business Model / Wanderson Paim de Jesus. – Porto Alegre: PPGC da UFRGS, 2014.

82 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Lisandro Zambenedetti Granville.

1. Network Virtualization. 2. Network Programmability. 3. Software Defined Networking. 4. Cloud Computing. I. Granville, Lisandro Zambenedetti. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Pró-Reitor de Coordenação Acadêmica: Prof. Rui Vicente Oppermann
Pró-Reitora de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Alexander Borges Ribeiro

"If I have seen farther than others,
it is because I have stood on the shoulders of giants."
— Sir Isaac Newton

ACKNOWLEDGMENTS

Certainly my first thanks goes to my parents for the incredible perseverance demonstrated in difficult times and for being my example of discipline, courage, humility, and honesty. For uncountable times they comforted me saying *it's gonna be ok*, simple but essential to keep me calm in the most difficult times. This achievement was only possible due to you. I also thank from the heart to my girlfriend Fernanda, that even physically far, was always comprehensive and near to support me. Being at her side made me a better person, personally and professionally, because I learned to value communication and human relationships.

I thank my advisor Lisandro for believing in my potential, for being an example of dedication and excellence, and a good friend, whom I could share philosophical doubts about life decisions. Being part of the Network Research Group at UFRGS added values to me that goes beyond Master's degree. I learned that there are no barriers and that anyone can produce great research works with dedication and hard work. I also thank professors Luciano and Marinho for always requiring the best of us. Certainly my skills in producing research was improved a lot, especially concerning to work presentations.

Living far from family is not easy, but in Porto Alegre I found so many friends that supported me and made me feel in Home. I thank my biggest partner and room mate Daniel for the fellowship during long two years, period that we shared not only a home, but several unforgettable experiences. I could not forget also to thank my other room mates Kleber (Catatau I), Javier (Catatau II), Juan (El Campesino), and Pedro (Barriga verde). With you guys, I lived great moments and learned valuable things such as being restless and dedicated with Kleber, to dance Bachata, to speak a good Spanish (or not), with the big (little) Javier, to prepare lemonade and understand carbon market with El Campesino, and to not give up on running with my friend Pedro.

I'm very thankful to all my friends of the research lab, in special to my great friend Oscar that for several times made me feel part of his family and that helped me a lot to achieve the results presented in this dissertation. Thanks to Marotta for the unforgettable moments of friendship we lived in travels, deadlines, and hard work times. With both I also shared great experiences in Japan, cycling around in Kyoto and eating (or trying to eat, isn't it Oscar?) using those "unhandleable" Japanese chopsticks. Thanks to my friends Cruz Alta and your stories, Juliano for your interest and patience in endless discussions about SDN, Felipe Carbone for your contagious calm, Angel and your inseparable "tucupi" that turned RU meals much more flavored, Purinho, Ricardo for your great help with programmable networks and friendship, Pedro Arthur, Jefferson, Weverton, Jair for your inspiring perseverance, Paolo for your sense of humor, Flávio "Cockroach", Lucas Bondan, Cristiano, Matheus Lehman, and everyone that I may have missed to mention but certainly were very important to me. Thank you all!

AGRADECIMENTOS

Agradeço primeiramente aos meus pais pela incrível perseverança demonstrada nos momentos difíceis e por serem meus exemplos de disciplina, coragem, humildade e honestidade. Por diversas vezes vocês me confortaram dizendo *vai dar tudo certo*, simples mas suficiente para me acalmar nas horas mais difíceis. Hoje dou mais um passo adiante graças a vocês. Também agradeço de coração a minha namorada Fernanda, que mesmo distante fisicamente sempre me compreendeu e apoiou. Estar ao seu lado me fez uma pessoa melhor, tanto pessoalmente quanto profissionalmente, porque aprendi a valorizar a comunicação e as relações humanas.

Agradeço ao meu orientador Prof. Lisandro por acreditar em minha capacidade, por ser um exemplo de dedicação e excelência, além de amigo, com o qual pude compartilhar devaneios sobre as decisões da vida. Ter feito parte do grupo de Redes da UFRGS me agregou qualificações que vão além do título de mestre. Aprendi que não existem barreiras e que dedicação e trabalho duro resultam em bons trabalhos de pesquisa. Agradeço também aos professores Luciano e Marinho por serem tão exigentes. Meu desempenho acadêmico certamente melhorou muito, principalmente em relação às apresentações de trabalhos.

Morar longe da família não é fácil, mas em Porto Alegre eu encontrei amigos que me apoiaram e fizeram me sentir em casa. Agradeço ao meu grande amigo Daniel pelo companheirismo durante longos dois anos, período em que compartilhamos não apenas a residência, mas também experiências inesquecíveis. Não poderia esquecer de agradecer aos demais companheiros de moradia Kleber (Catatau I), Javier (Catatau II), Juan (El Campesino) e o Pedro. Com vocês eu vivi ótimos momentos e aprendi a ser mais inquieto e dedicado com o Kleber, a dançar Bachata e falar espanhol (ou *intentar hablar*) com o grande (pequeno) Javier, a preparar limonada e entender o mercado de carbono com El Campesino, e a não desistir da corrida com meu amigo Pedro.

Sou muito agradecido aos amigos do laboratório, em especial ao grande amigo Oscar Caicedo que por diversas vezes me fez sentir parte da sua família e que me ajudou muito a alcançar os resultados apresentados nessa dissertação. Agradeço ao Marotta pelos incontáveis momentos de parceria que vivemos em viagens, *deadlines* e trabalho duro. Com ambos compartilhei também uma grande experiência no Japão, andando de bicicleta pelas ruas de Quioto, comendo (ou tentando comer né Oscar?) com *hashi*. Agradeço aos amigos Cruz Alta e suas estorias, Juliano pelo interesse e paciência nas discussões sem fim sobre SDN, Felipe por sua contagiosa calma, Angel e seu inseparável "tucupi" que deixava o almoço do RU mais gostoso, Purinho, Ricardo pela ajuda com Programabilidade de Redes e parceria, Jedi, Jeff, Weverton, Jair pela inspiradora perseverança, Paolo pelo senso de humor, Flávio "Barata", Lucas Bondan, Cristiano, Matheus e todos que eu não mencionei mas que certamente foram muito importantes pra mim. Obrigado a todos!

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 8

LIST OF FIGURES 10

LIST OF TABLES 11

ABSTRACT 12

RESUMO 13

1 INTRODUCTION 15

2 BACKGROUND AND RELATED WORK 18
2.1 Programmable Networks 18
2.2 Network Virtualization 23
2.3 Programmable Virtual Networks 25

3 CONCEPTUAL SOLUTION 27
3.1 Network Programming as a Service 27
3.2 ProViNet: Programmable Virtual Network Management Platform . . . 30
3.2.1 Managing Virtual Infrastructures 31
3.2.2 Network Applications Management 31
3.2.3 Network Applications Development 32
3.2.4 Network Applications Execution 34

4 PROTOTYPING PROVINET PLATFORM 37
4.1 Programmable Virtual Infrastructure Provisioning 37
4.1.1 Scalable Control Plane 37
4.1.2 Virtual Infrastructure Provider 40
4.2 Network Application Management 42
4.2.1 Developing 43
4.2.2 Executing 44

5 EXPERIMENTAL EVALUATION 46
5.1 Qualitative Comparison 46
5.2 Case Study and Performance Analysis 48

6 CONCLUSION 52
6.1 Main Contributions and Results Obtained 52
6.2 Final Remarks and Future Work 53

REFERENCES 55
ANEXO A **SUBMITTED PAPER – COMPSAC 2013** 58
ANEXO B **SUBMITTED PAPER – SBRC 2013** 68

LIST OF ABBREVIATIONS AND ACRONYMS

OF	*OpenFlow*
CTL	*Controller*
SBAPI	*Southbound API*
NBAPI	*Northbound API*
VM	*Virtual Machine*
VN	*Virtual Network*
SPs	*Service Providers*
QoS	*Quality-of-Service*
PN	*Programmable Networks*
SCP	*Scalable Control Plane*
VPN	*Virtual Private Network*
NOS	*Network Operating System*
SDK	*Software Development Kit*
InPs	*Infrastructure Providers*
NGI	*Next Generation Internet*
NMS	*Network Management System*
ONF	*Open Networking Foundation*
VLAN	*Virtual Local Area Network*
SDN	*Software Defined Networking*
PVN	*Programmable Virtual Network*
MPLS	*Multi Protocol Label Switching*
VIP	*Virtual Infrastructure Provider*
AON	*Application-Oriented Networking*
REST	*Representational State Transfer*
IETF	*Internet Engineering Task Force*
NPaaS	*Network Programming as a Service*

API	*Application Programming Interface*
NVE	*Network Virtualization Environment*
BPMN	*Business Process Management Notation*
DVSC	*Distributed Virtual Switch Controller*
ITU	*International Telecommunication Union*
CORBA	*Common Object Request Broker Architecture*
VIDL	*Virtual Infrastructure Description Language*
ISO	*International Organization for Standardization*
IEEE	*Institute of Electrical and Electronics Engineers*

LIST OF FIGURES

Figure 2.1: Four steps of flow handling in OpenFlow technology (YEGANEH; TOOTOONCHIAN; GANJALI, 2013). 20
Figure 2.2: Software-Defined Networking architecture (ONF White Paper). . . . 23
Figure 2.3: Network Virtualization Environment proposed by Boutaba (CHOWDHURY; BOUTABA, 2009). 24
Figure 2.4: Network Virtualization Business Model (CHOWDHURY; BOUTABA, 2008). 25
Figure 2.5: Network Virtualization Business Model proposed by 4WARD project. (ACHEMLAL, 2010). 26

Figure 3.1: Network Programming as a Service Architecture 28
Figure 3.2: ProViNet Conceptual Architecture. 30
Figure 3.3: *Workflow* implementation of a simple intrusion detector application. . 35
Figure 3.4: Network Application Execution approach. 36

Figure 4.1: Scalable Programmable Virtual Network Infrastructure 38
Figure 4.2: ProViNet Management Interface sreenshot. 39
Figure 4.3: Aurora: Resource access information screenshot. 42
Figure 4.4: Aurora Virtual Network topology screenshot. 43
Figure 4.5: ProViNet Graphical Development Interface. 44
Figure 4.6: Network Application that changes Firewall status. 45

Figure 5.1: Use case scenario of physical network migration 49
Figure 5.2: Sequence Diagram of resources provisioning. 50

LIST OF TABLES

Table 2.1: Open Networking Foundation Members in May 2013 22

Table 5.1: Comparison of Related Proposals 48
Table 5.2: Programmable Virtual Network Provisioning 50
Table 5.3: NBAPI Dispatcher performance 50

ABSTRACT

Computer networks have evolved to accommodate a wide variety of services, such as streaming of high quality videos and delay-sensitive content delivery. These services have increased the demand for features not originally considered in the Internet. Aiming to address novel network demands quickly, some researchers proposed Programmable Networks, in which network devices behavior can be changed using applications. Notwithstanding, such behavior might not be a consensus between computer network stakeholders. The emergence of Virtualized Networks overcame this issue by allowing the co-existence of multiple virtual networks on the same physical infrastructure. Finally, the convergence of programmability and virtualization techniques are explored within a third concept, the Programmable Virtual Networks (PVN). Faced with this new reality, network administrators are no longer just looking at network devices. They are looking at a system made of virtual devices and applications that define each virtual network behavior. This requires not just new tools and management approaches, over and above that, requires new thinking.

PVN deployments are found mostly in shared experimental facilities (also known as testbeds) and Cloud Computing environments. Testbeds are very innovation friendly, but with strong limitations in regards to taking experimental solutions to production. On the other hand, Cloud computing is a great production environment, but presents flexibility and innovation restrictions once network solutions adopted are usually proprietary. Therefore, in this dissertation it is introduced Network Programming as a Service (NPaaS), a new business model that aims to facilitate the conduct of innovative solutions for production environments. Different from traditional network business models, where end-users are just consumers of network services already available, in NPaaS, end-users are also able to develop and deploy novel network solutions. To support NPaaS, Programmable Virtual Network management platform is proposed. Such platform, named ProViNet, provides all architectural and technical features necessary to enable NPaaS deployment and management.

A qualitative evaluation of the NPaaS business model was performed, and the result was contrasted with some of the current models, thus, emphasizing the singularity of NPaaS. In the meanwhile, an experimental evaluation was conducted to demonstrate the feasibility of ProViNet platform. Results have shown that NPaaS represent a promising alternative for virtual network environments with public access such as public clouds. Moreover, a quantitative evaluation of the platform prototype demonstrated the technical feasibility and proved that network applications developed using BPMN are able to run with acceptable performance rates.

Keywords: Network Virtualization, Network Programmability, Software Defined Networking, Cloud Computing.

Programabilidade de Redes como Serviço: um modelo de negócios propício à inovação

RESUMO

As redes de computadores têm evoluído para acomodar uma grande variedade de serviços, tais como streaming de vídeos de alta qualidade e entrega de conteúdo sensível a atrasos. Estes serviços têm aumentado a demanda por recursos não originalmente considerados na Internet. Com a promessa de atender novas demandas de rede rapidamente, pesquisadores propuseram Redes Programáveis, nas quais o comportamento dos dispositivos de rede pode ser alterado utilizando aplicativos. Entretanto, tal comportamento pode não ser um consenso entre usuários da rede. O surgimento de Redes Virtualizadas superou tal questão, ao permitir a coexistência de múltiplas redes virtuais sobre a mesma infraestrutura física. A fim de se obter redes virtuais isoladas com comportamento programável, foram propostas as Redes Virtuais Programáveis (RVP). Diante dessa nova realidade, os administradores de rede não estão mais olhando unicamente para dispositivos de rede. Eles estão olhando para um sistema composto de dispositivos e aplicativos de rede que definem o comportamento individual de cada rede virtual. Isso requer não apenas novas ferramentas e abordagens de gerenciamento, além disso, exige a revisão de conceitos tradicionais sobre redes.

Implementações de RVP são encontradas principalmente em *testbeds* e ambientes de Computação em Nuvem. *Testbeds* são muito propícios à inovação, mas possuem fortes limitações no que diz respeito a migração de soluções experimentais para produção. Por outro lado, computação em nuvem é um ótimo ambiente de produção, mas possui restrições de flexibilidade e inovação, uma vez que as soluções de rede adotadas geralmente são proprietárias. Portanto, nesta dissertação introduz-se um novo modelo de negócio que permite a criação de soluções inovadoras em ambientes de produção, a Programabilidade de Redes como um Serviço (NPaaS). Diferente do modelo de negócio de redes tradicionais, onde os usuários finais são apenas consumidores dos serviços de rede já disponíveis, em NPaaS os usuários finais também são capazes de desenvolver e implantar novas soluções de rede. Para apoiar NPaaS, propõe-se uma plataforma de gerenciamento de rede virtual programável, chamada ProViNet. Essa plataforma fornece a arquitetura de software e estratégias necessárias para permitir a implantação e gestão NPaaS.

Uma avaliação qualitativa do modelo de negócio NPaaS foi realizada, o resultado foi contrastado com alguns dos modelos de negócio praticados atualmente. Assim, enfatizando a singularidade do NPaaS. Enquanto isso, uma avaliação experimental foi realizada para demonstrar a viabilidade da plataforma ProViNet. Os resultados mostraram que NPaaS representa uma alternativa promissora para ambientes de rede virtual com acesso público, como as nuvens públicas. Além disso, uma avaliação quantitativa do protótipo da plataforma demonstrou a viabilidade técnica e provou que aplicativos de rede desenvolvidos usando BPMN são capazes de executar com desempenho aceitáveis.

Palavras-chave: Network Programmability, Network Virtualization, Cloud Computing.

1 INTRODUCTION

Computer networks have evolved to accommodate a wide variety of services, such as streaming of high quality videos and delay-sensitive content delivery. These services have increased the demand for features not originally considered in the Internet, such as the assurance of very high bandwidth, location-awareness, and exponential growth of addressing demand. In order to address these emerging demands, the network community developed solutions such as IPv6 and IPSec. However, such solutions were proposed for over 10 years and still have been struggling to be barely adopted. The long delay from proposition to deployment of new features is related to the fact that the core of Internet 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 (HAUSHEER et al., 2011).

Among the factors that contribute to the Internet's hostile environment for innovation is the necessity of deep and global modifications in order to adopt new solutions. In the Internet, every device is somehow dependent on the other, so in order to keep interoperability among them, a novel communication protocol must be known by several devices. The high costs it may generate, including equipment replacement, ends up discouraging novel solutions adoption. Besides that, in an effort to keep legacy support, institutions like ISO, IEEE, ITU, and IETF have the role of discussing and standardizing new network protocols and communication approaches. The problem is that standardization process may take too long, becoming also an obstacle for the adoption of novel solutions. Another crucial factor is the dependency on the profit-oriented interest of network equipment manufacturers. They tend to develop solutions with higher potential of profit instead of most innovative. As a consequence, special needs are not met. On top of that, network companies usually develop proprietary software and hardware, giving no alternatives for customizations by end-users.

With the promise to reverse this state of ossification, some research works emerged in the late 90's introducing the concept of *Programmable Network* (CAMPBELL et al., 1999). This concept regards the fast, flexible, and dynamic deployment of novel in-network services in response to emerging demands (GALIS et al., 2004). Active Networks (PSOUNIS, 1999) and Open Signaling (CAMPBELL et al., 1999) are examples of programmable network technologies from the 90's. The lack of widespread adoption of such technologies by the industry was mainly because of security and isolation issues; network devices could be easily compromised with malicious code. Over time, network companies launched software modules that enabled certain level of customization in their devices. Cisco, for example, launched the Application-Oriented Networking (AON) (CISCO, 2013), while Juniper unveiled Junos Software Development Kit (SDK) (SUGIYAMA, 2012). In general, this kind of solution enable the development of appli-

cations that extend or add functionality to devices. However, there is no cross-vendor interoperability, *i.e.*, applications developed for one manufacturer's equipment are not compatible with another's.

Building programmable networks is about being able to address emerging demands quickly by aggregating novel services into the network. Considering that computer networks are usually a shared resource, such demands might not be a consensus within *stakeholders*, *i.e.* those interested in the network services, including end-users, network managers and administrators. In order to address such conflicting goals, the research community started investing on the concept of Network Virtualization (CHOWDHURY; BOUTABA, 2010). A virtual network environment allows the coexistence of multiples Virtual Networks (VN) on the same physical infrastructure (also called physical substrate). Thereby, each virtual network could be designed to address distinct *stakeholder*'s demands. Nevertheless, to enable such virtual networks to behave accordingly, the network devices must be programmable. When a network brings together the features of Virtualization and Programmability it shall be called Programmable Virtual Networks (PVN) (CHOWDHURY; BOUTABA, 2009).

Managing traditional computer networks presents several challenges, but managing virtualized and programmable networks might turn such task even more complex. A PVN is composed by arbitrary virtual network topologies independent from physical topology; moreover, each virtual network is able to run different and independent protocols and services. Given this new type of environment, network administrators are no longer just looking at network devices. They are looking at a system made of virtual devices and applications that define each virtual network behavior. This requires not just new management approaches and tools, but over and above that, requires new thinking. Designing solutions that enable the different network stakeholders to communicate, program, and manage PVN harmoniously represent an open research problem. Mainly because it requires rethinking the role of each stakeholder over the network, defining novel business models and architectures. In addition, novel abstraction models are required to enable stakeholders with different technical knowledge to program, monitor, and administrate programmable virtual networks.

The existing use cases of PVN follow different business models and implement varying approaches for interfacing PVN to its users. Some shared experimental facilities (testbeds), such as OFELIA, FIBRE and GENI, employ PVN to provide a set of isolated network resources with customizable protocols and services for experimental purposes. In the testbed business model, each member contributes with a set of resources (such as switches, routers, and servers), composing a federation system. Such model does not allow the employment of resources in production mode and pose strict rules for researchers access. Another typical PVN use case is found in Cloud Computing environments. Due to the fast growth of services provided by Cloud players, their data-center's network must be flexible and scalable. In doing so, PVN is seen as an alternative to enable Cloud network administrators to address such demands. Two different business models can be deduced from these use cases. The first one, testbeds, are very innovation friendly, but with strong limitations in regard to taking experimental solutions to production. On the other hand, Cloud computing is a real production environment, but has flexibility and innovation restrictions once the network solutions adopted by them are usually proprietary.

In face of the presented above, this dissertation introduces Network Programming as a Service (NPaaS), a novel business model that aims at allowing the conduction of innovative solutions over production environments. Different from traditional networks,

where end-users are just consumers of the network services already available, in NPaaS, end-users are also able to develop and deploy novel network solutions. By doing this, it is expected an increase in the amount of innovative solutions, once there would be a larger amount of capable innovators. As a side effect of such new point of view given to end-users, there is an increase in management complexity. So, in order to support NPaaS, it is proposed a **Pro**grammable **Vi**rtual **Net**work management platform, called ProViNet. Such a platform provides all architectural and technical requirements necessary to enable NPaaS deployment and management. Along with the proposed business model and the platform, this dissertation contributes with:

1. A novel network abstraction model based in Business Process Management Notation (BPMN). Which allows users with different levels of technical expertise to develop network applications. Such an approach encourages the development and sharing of novel network solutions even by users non-familiar with network technical details;

2. An architecture, developed to ProViNet platform, that leverages the benefits of recent programmable network technologies and provides for flexibility, scalability, and availability;

3. Additional grammar elements to complement the Virtual Infrastructure Description Language (VXDL) (KOSLOVSKI; PRIMET; CHARãO, 2008), enabling a proper representation of SDN control plane architectural elements in textual format.

A qualitative evaluation of the NPaaS business model was performed, and the result was contrasted with some of the current played models. Thus, emphasizing the singularity of NPaaS. Meanwhile, an experimental evaluation was conducted to demonstrate the feasibility of ProViNet platform and to support the performance analysis presented later. Results have shown that NPaaS represents a promising alternative for virtual network environments with public access such as public clouds. Moreover, the platform prototype demonstrated the technical feasibility and proved that network applications, developed as BPMN workflows, are able to run with acceptable performance rates. In addition, the platform reduces the complexity of functions such as developing, deploying, and managing network applications in PVN.

The remaining chapters of this dissertation are organized as follows. In Chapter 2, some background concepts considered fundamental for the understanding and motivation of the solution proposed are presented. In the same Chapter, the main works that discuss about Programmability in the context of Virtual Networks are outlined. Afterwards, in Chapter 3, NPaaS business model is discussed and the ProViNet platform is detailed. In order to demonstrate the feasibility of ProViNet, a prototype is introduced in Chapter 4. Then, a quantitative evaluation of such prototype and a qualitative evaluation of NPaaS are presented in Section 5. Finally, in Chapter 6 we close this dissertation with concluding remarks and future work.

2 BACKGROUND AND RELATED WORK

There is a growing consensus that computer networks must be open, extensible and programmable in order to follow the fast evolving and diversified demand (HAUSHEER et al., 2011). This topic has been well discussed in the literature, starting back in the 90's, when emerged the first proposals of Programmable Networks (PN) (CAMPBELL et al., 1999) (LIN et al., 2011). Since then, the popularity of such topic within the research community has been cyclic. In this chapter, a historical overview of this research area is presented, highlighting the major contributions and drawbacks of the approaches proposed over the time. It is also discussed how Network Virtualization (NV) (CHOWDHURY; BOUTABA, 2009) (CHOWDHURY; BOUTABA, 2010) has evolved in the same period and what is its role beyond programmability field.

From the joint application of PN and NV emerges Programmable Virtual Networks (PVN) (CHOWDHURY; BOUTABA, 2009). In PVN, virtual network functions, such as traffic isolation and resource sharing, are added with programmability. In this way it is possible to configure different network services and protocols to virtual networks dynamically, more precisely, creating and deploying novel in-network softwares. Different from network programming strategies from the 90's, such as Smart Packets and ANTS, in PVN, novel protocols can be deployed separately on distinct virtual networks, causing no impact on the whole network, but only on a set of isolated resources. As a consequence, emerging demands could be addressed quickly by novel network softwares.

In light of the vast existing theoretical foundations around the concepts of NV, PN, and PVN, in this Chapter it is discussed some of the mains works aiming to support discussions presented in later chapters. Therefore, in Section 2.1 Programmable Networks concept and a historical overview of the related works are given. Then, in Section 2.2 the concept of Network Virtualization is presented. Finally, in Section 2.3 it is dicussed about PVN, a concept that encompasses many of the recent proposals.

2.1 Programmable Networks

In general, administrators of network service providers cannot access the switch/router control environments (*e.g.*, IOS), algorithms (*e.g.*, routing protocols), and states (*e.g.*, routing tables, flows states). Such access limitation causes dependence on network equipment vendors, which decide whether to implement novel network solutions or not. As a consequence, creating new network service has become a hard task due to the closed nature of network nodes. To overcome this limitation, the network research community has proposed the concept of Programmable Networks (PN) (ROSS, 1989). The guiding concept of Programmable Networks defines that networks must enable the rapid deployment of novel network solutions and customization of the existing in response to emerging de-

mands, independent of network equipment vendors. One of the major challenges faced by PN proposals is finding a suitable trade off among network security, performance, flexibility, and manageability. With this in mind, computer networks must be carefully engineered turning networking infrastructures open, extensible and programmable (CAMPBELL et al., 1999).

Various research initiatives, such as Next Generation Internet (NGI), DARPA Active Networks, CANARIE, and Internet2, emerged in the 90's with the goal of providing network resources and conceptual basis for supporting network research, which ended up fostering proposals of programmable network models and architectures. Along with such initiatives some specialized conferences arose with aiming to promote network programmability proposals. Open signaling for ATM, Internet and Mobile Networks (OPENSIG) (CAMPBELL et al., 1999) and Conference on Open Architectures and Network Programs (OPENARCH) (IEEE Second Conference on Open Architectures and Network Programming Proceedings, 1999) are examples of these conferences, in which many important proposals were published, such as IEEE P1520 (BISWAS et al., 1998), SmartPackets (SCHWARTZ et al., 1999), and Netscript (FLORISSI, 1998).

There is no consensus when categorizing programmable network proposals. Papers describing five (LIN et al., 2011), three (GALIS et al., 2004), and two (CAMPBELL et al., 1999) broad areas are found in the literature. Following the categorization of Galis, which proved to be more recurrent in the works analyzed, three major areas are discussed below.

Open Signaling is a technique that models the hardware (switches/routers) communication by defining open programmable interfaces. Opening the control of networks through a set of well-defined network programming interfaces and distributed programming environments (*e.g.*, CORBA, DCOM, Java) enables the development of novel services and network architectures. By doing this, physical network nodes can be abstracted as a set of objects in a distributed system. Then, network operators can manipulate the state of such objects through programming interfaces creating novel services that ensure network requirements, such as quality of service and security. In the late 90s, there were several proposals considering Open Signaling, such as Xbind (LAZAR; LIM; MARCONCINI, 1996), Tempest (MERWE et al., 1998), and Mobiware (ANGIN et al., 1998).

Programmable Active Networks (PSOUNIS, 1999), also referred to just as Active Networks, regard the use of special devices capable to execute custom applications that modifies packets flowing through them, including packet payload. The core idea is allowing users (end-user, operators, and service providers) to develop applications and inject the code into network nodes to perform computation upon the packets along the way. In order to enable an application code to run in many devices, a common interface called NodeOS was defined. In a hybrid network, traditional nodes, *i.e.* not active, are still able to forward and route active packets, though cannot perform any special computation in them. Active Networks were not widely adopted for several reasons, most notably because of security (preventing malicious use of the network), isolation (protecting one application's behavior from another), and performance (programmable elements slow-down the forwarding path).

Node Operating Systems represent the opposite idea of having a global network operating system. In this case, instead of considering nodes as objects within a global view, each node represents a complete system. Each one has defined its own data space and variables. This represents an advantage in terms of deeper programmability, once it does not depend on abstractions or standard interfaces. This concept was more successful than the previous described, once it is possible to find recent proposals considering such ap-

proach (CHERTOV; HAVEY; ALMEROTH, 2010). Due to such success, this discussion goes further with some examples of implementations.

As a result of a MIT research in the 90's, Click Modular Router (MORRIS et al., 1999) was proposed. Similarly, Promile (RIO et al., 2001) was proposed by University College London. Both introduce a more intrusive and clean slate approach for programming network nodes. Such approach defines a set of software packets, called modules, which implement different functions and can be arranged in order to compose network services. Click proposals usually employ regular PCs to run Click modules, but present low performance if compared to production network devices. The last version of this technology was published in September 2011. Since its original publication, many works employed Click to develop their solutions (BAVIER et al., 2006) (ELMELEEGY; COX; NG, 2007) (CHERTOV; HAVEY; ALMEROTH, 2010).

Proposals from industry such as AON, 1000V and OnePK from CISCO and JunOS SDK (SUGIYAMA, 2012) from Juniper provide APIs to enable users to build patches that run on their proprietary node OS. Recently such approaches are being used as an answer from the industry to the emergent open solutions for programmable network deployment. However, considering the current heterogeneous network environment, in terms of equipment vendor, a solution developed to a specific vendor equipment will hardly scale, because it cannot interact with devices from other vendors.

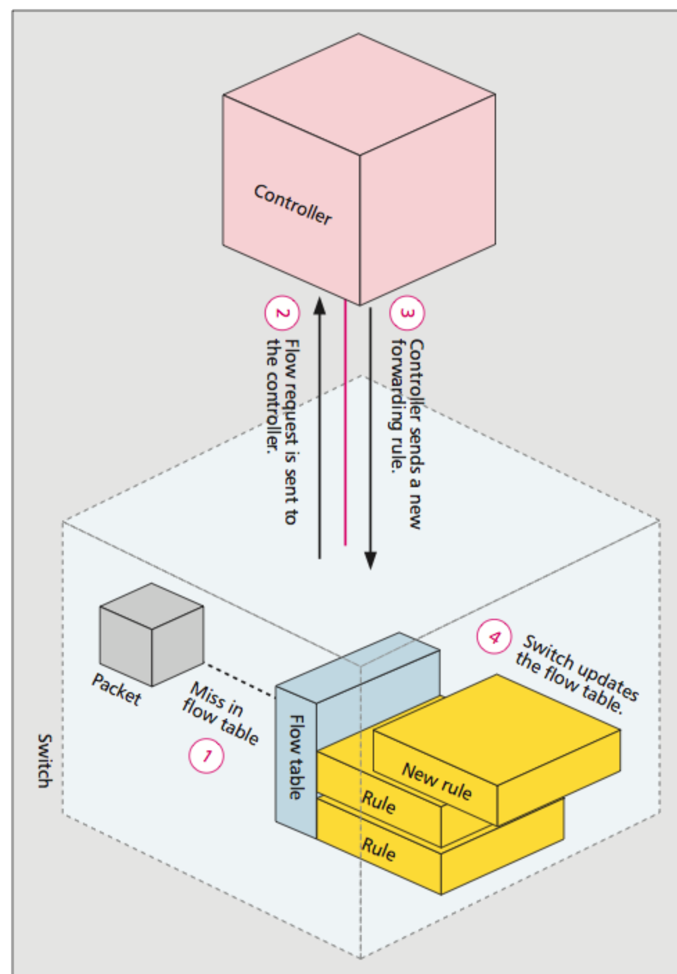

Figure 2.1: Four steps of flow handling in OpenFlow technology (YEGANEH; TOOTOONCHIAN; GANJALI, 2013).

After a period of numerous publications around 90's, PN research area became less popular; maybe the research community was disappointed by the lack of adoption by the industry. Network administrators did not feel confident with the idea of having application codes executing in their production networks, which could be easily compromised with malicious code. In March 2008, Stanford, in conjunction with Princeton, Berkeley, MIT, and others universities, proposed OpenFlow (OF) (MCKEOWN et al., 2008). OF is composed of three components, *(i)* OF Switches, produced by many vendors (such as HP, IBM, and Datacom), which are found in two versions: hybrid, with traditional network services/protocols, and pure OF with exclusive OF support; *(ii)* OF Controller, an external device (typically a regular x86 PC) responsible for running the network control logic and instructing OF Switch, and finally *(iii)*, OF Protocol, which enables the communication between Controller(s) and Switch(es). OF is flow-based and, as shown in Figure 2.1, every flow arriving in the Switch is matched against a flow table, when it miss (1), the packet data is sent, through an OF protocol request, to Controller (2), which in turn evaluates packet data and sends back a rule to match such kind of packets (3), lastly this rule is added to the flow table (4).

Before OpenFlow launch, a quite similar concept was proposed by IETF, called ForCES (KHOSRAVI; ANDERSON, 2003), which has the last version published in RFC 5810 (DORIA; HADI SALIM, 2010). The clear distinction between OpenFlow and ForCES can be found online in IETF drafts[1]. Briefly, OpenFlow defines the physical separation of the control and forwarding elements. Distinctly, ForCES abstracts forwarding devices as Network Elements and defines an internal separation of control and forwarding elements.

Shortly after OpenFlow publication, during a keynote presentation at INFOCOM conference, in April 2009, Nick McKeown announced Software-Defined Networking (SDN) (MCKEOWN, 2009) (LANTZ; HELLER; MCKEOWN, 2010). In order to guide SDN evolvement and standardize critical SDN architectural elements, such as OpenFlow, was created Open Networking Foundation (ONF). ONF is a non-profit industry consortium composed of many companies. The complete list of ONF members is presented in the Table 2.1. As defined by ONF, SDN is an emerging network architecture where network control is directly programmable and decoupled from forwarding. In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications. As a result, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build highly scalable, flexible networks that readily adapt to changing business needs.

There are distinct and clear definitions of OF and SDN, but confusions and misuses are found frequently in academia and industry publications. In order to make such distinction clear, the SDN architecture shown in the Figure 2.2 identifies a communication interface between the Control and Infrastructure Layers. OpenFlow is an alternative and currently the most adopted protocol to enable such communication. There are several advantages of using OpenFlow because it is open and has broad community support, but there are private alternatives employed in SDN solutions from the industry. The emergence of private SDN solutions reflects a current trend. The high expectation around SDN is inducing industries to employ SDN as a buzzword to attract customers and drive business growth. In a broader perspective, it shows that Programmable Network community is changing the focus from the basic mechanisms that dynamically install and move service code within the network, to more commercial and broad solution.

[1] http://tools.ietf.org/html/draft-wang-forces-compare-openflow-forces-01

6WIND	A10 Networks
Active Broadband Networks	ADVA Optical Networking
Alcatel-Lucent / Nuage Networks	Aricent Group
Arista	Big Switch Networks
Broadcom	Brocade
Centec Networks	Ceragon
China Mobile (US Research Center)	Ciena
Cisco	Citrix
CohesiveFT	Colt
Cyan	Dell/Force10
Deutsche Telekom	Ericsson
ETRI	Extreme Networks
F5 / LineRate Systems	Facebook
France Telecom Orange	Freescale
Fujitsu	Gigamon
Goldman Sachs	Google
Hitachi	HP
Huawei	IBM
Infinera	Infoblox / FlowForwarding
Intel	Intune Networks
IP Infusion	Ixia
Juniper Networks	KDDI
Korea Telecom	Lancope
Level 3 Communications	LSI
Luxoft	Marvell
MediaTek	Mellanox
Metaswitch Networks	Microsoft
Midokura	NCL Communications K.K.
NEC	Netgear
Netronome	NetScout Systems
Nokia Siemens Networks	NoviFlow
NTT Communications	Oracle
Overture Networks	PICA8
Plexxi Inc.	Qosmos
Rackspace	Radware
Riverbed Technology	Samsung
SK Telecom	Spirent
Sunbay	Swisscom
Tail-f Systems	Tallac
Tekelec	Telecom Italia
Telefónica	Tellabs
Tencent	Texas Instruments
Thales	Tilera
Transmode	Turk Telekom / Argela
TW Telecom	Vello Systems
Verisign	Verizon
VMware/Nicira	Xpliant
Yahoo	ZTE Corporation

Table 2.1: Open Networking Foundation Members in May 2013

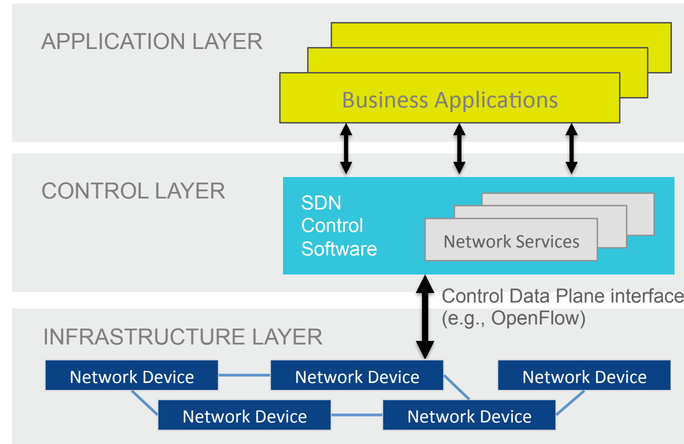

Figure 2.2: Software-Defined Networking architecture (ONF White Paper).

2.2 Network Virtualization

The literature point out Network Virtualization (NV) (CHOWDHURY; BOUTABA, 2008) as a solution for the Internet ossification problem. Purists see on NV a doorway for evaluating novel architectures while pluralists consider NV part of next generation architecture itself. The main characteristic of such networks is the sharing of a substrate by co-existing networks. Not too late, substrate is defined as a set of physical resources in the network infrastructure, such as PCs, switches, and routers. Chowdhury and Boutaba (CHOWDHURY; BOUTABA, 2009) proposed Network Virtualization Environment (NVE), in which the current ISPs is split into two independent entities: Infrastructure Providers (InPs) and Service Providers (SPs). The former is responsible for keeping physical infrastructure, while the last uses InPs resources to create virtual networks and provide network services for end-user.

NVE is properly depicted in Figure 2.3. There are, at the bottom, two InPs with different physical topologies and end hosts. Above them are two Service Providers, each one in control of a single Virtual Network. As can be noticed, these virtual networks have distinct topologies both in relation to each other and in relation to the physical infrastructure in the InPs. The white circles at the bottom represent physical nodes and the gray circles represent virtual nodes. Connecting the virtual nodes there are virtual links, and connecting physical nodes, physical links. Worth mentioning that even the links between the virtual nodes are distinct from the physical links.

Along with NVE, Chowdhury and Boutaba also pointed eight design goals that lead a successful Network Virtualization project. *Coexistence* is the first design and refers to the necessity to support many Virtual Networks (VN) over the same substrate. Furthermore, there should be *Flexibility* enough to accommodate arbitrary VN topologies, forwarding routines and management protocols. Probably numerous VN will be distributed in layers from end-to-end in the physical infrastructure, and keeping accountability of each layer requires *Manageability*. Considering that the total network resource demand becomes the sum of each VN layer demand, and that there is an expected increase in resource demand for next years, *Scalability* must be a permanent concern for InPs. Due to the conflicting goals among NVE stakeholders, keeping *Isolation* among VNs reveals crucial. Besides isolation among VN, it is important to ensure *Stability and Convergence* of underlying physical infrastructure in order to mitigate potential incidents.

Programmability is also pointed as a fundamental design goal to enable flexibility and

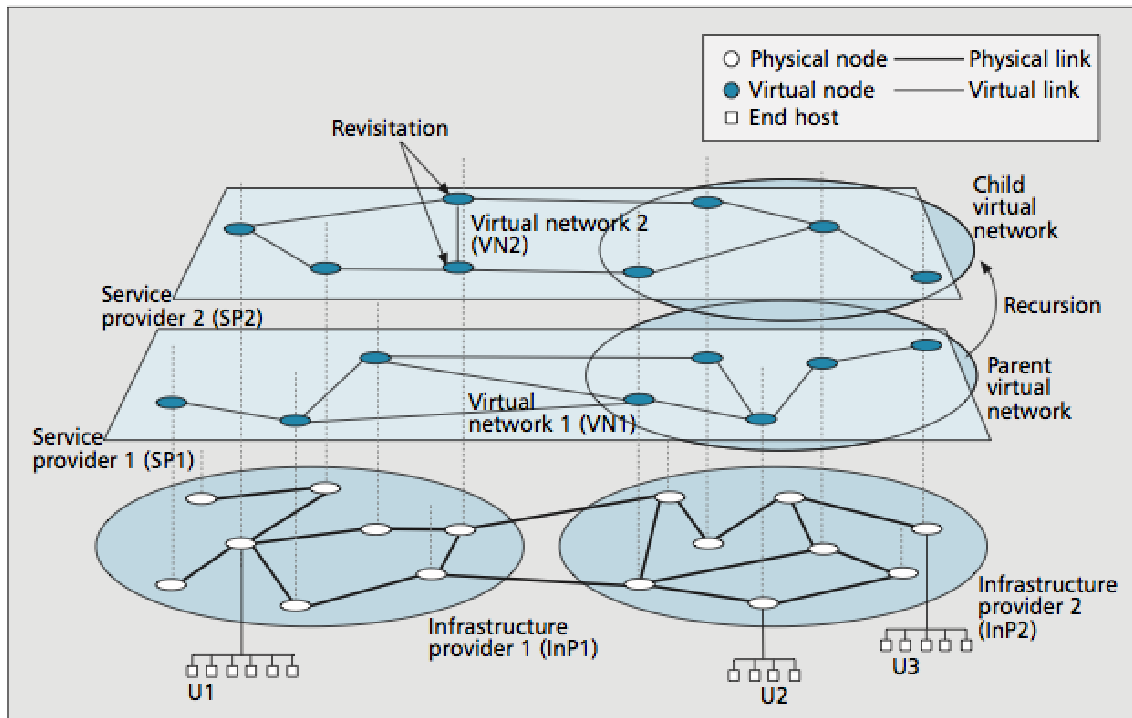

Figure 2.3: Network Virtualization Environment proposed by Boutaba (CHOWDHURY; BOUTABA, 2009).

manageability in NVE. Employing programming techniques, VN Service Providers will be able to provide different network services to attend end user specific demands. Moreover, novel architectures can be developed and deployed fostering innovation. Chowdhury and Boutaba also draw attention to the necessity to address *Heterogeneity* of underlying technologies and protocols, end-user access devices and InPs. At last, NVE should consider *Legacy Support*, a recurring challenge in the deployment of disruptive technologies.

Grant Traffic Differentiation (TD) is a primary function of NV and is straightly aligned with many of the design goals aforementioned, such as isolation, coexistence, flexibility, and heterogeneity. There are several TD approaches, including the addition of tags into packets header as distinguishing parameter (*e.g.* VLAN), enabling customized actions in intermediate nodes. It is also usual to encapsulate packets with additional header layers, allowing packets to be routed from end-to-end based in such outer layer (e.g. IP tunnels and MPLS). OpenFlow (see Figure 2.1) is also a TD enabler, but instead of adding tags, OF considers the flow characteristics itself, *i.e.* the existing information in TCP/IP headers, such as IP or MAC destination/source address.

Although network virtualization techniques exists since the 90's, such as VLANs, it has received special attention recently driven by the success of Cloud Computing. In this environment, a virtual network interconnects a set of virtual machines. Such virtual network can be technically deployed distinctly (such as VLAN, VPN, and application level overlay (CHOWDHURY; BOUTABA, 2009)). Whatever the approach taken is, the result is that users will have an isolated virtual network interconnecting their own virtual machines. It is convenient and usual in the academia, referring to the set of network, computing, and storage resources belonging to a user as *Slice*.

2.3 Programmable Virtual Networks

Programmable Virtual Networks (CHOWDHURY; BOUTABA, 2009) concept defines programmability as a special requirement of network virtualization, which enables properly configuration of services within virtual networks. As presented in the last two sections, there are several approaches to build virtual networks. One of them is by using OpenFlow. OF enables virtual networks creation by slicing network traffic according to specific flow characteristics. But, in addition to virtualization, OF also enables programmability, once it is part of a programmable network architecture called SDN. Such fact allows defining SDN as a PVN particular implementation. Worth noting that other technologies could be joined to implement PVN as well. For instance, a set of programmable software routers organized in a virtualized infrastructure, with network traffic sliced by VLAN tags, could also be considered a PVN.

Many environments could take advantage of PVN deployment. Data Centers, for example, would be able to attend different client demands by programming and providing different network solutions to them. Cloud Computing providers would be able to provide a larger set of virtual network services for customers. Meridian (BANIKAZEMI et al., 2013) is an effort toward this goal. It leverages SDN architecture to support Cloud Computing virtual network provisioning and propose abstractions. A real use case of PVN was developed by Google, which employed SDN in two worldwide backbones (FOUNDATION, 2013). Likewise, network facilities, also known as *testbeds*, has carried PVN projects in their environments. By doing this, researchers can require virtual networks and computer resources in order to experiment novel solutions.

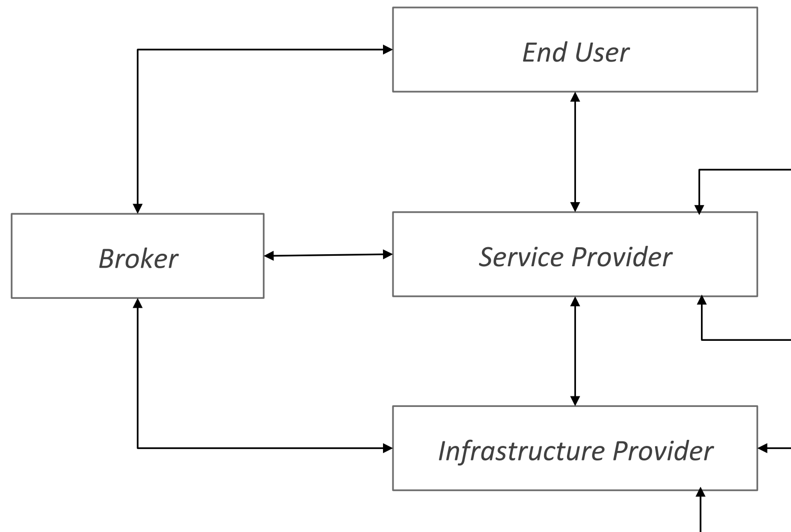

Figure 2.4: Network Virtualization Business Model (CHOWDHURY; BOUTABA, 2008).

Each one of the environments mentioned above was designed to address specific requirements. For instance, Google found out that SDN was the solution for its WAN network requirements. Although SDN enables network programmability, the access and programmability of Google's network are restricted to small group of employees. On the other hand, virtual network of *testbed* environments prove being more open for external deployment of novel services, which is performed by authorized researchers. Considering Public Cloud providers (*e.g.* Amazon AWS, Azure, and G Cloud), the access to the virtual network occurs, in general, through a few predefined services, such as creating subnetworks, multi-tenancy, DMZ, and configuring DNS servers. These services are developed,

configured, and deployed previously by Private Clouds providers, while consumers of such services are able just to configure some parameters of them. Even in Private Clouds that adopt programmable virtual networks, the services running in the virtual network are defined by network administrators and operators.

In summary, the access policies, roles and responsibilities are distinct over the several PVN deployments. These characteristics in conjunction with some others define a business model, which describe how the organization will create, delivery, and capture value (social, economical, cultural, or other kind of value). There are in the literature some business models related to PVN. For instance, Mosharaf *et.al.* (CHOWDHURY; BOUTABA, 2008) proposed in 2008 the Network Virtualization Business Model, which is depicted in the Figure 2.4. In such model, *Service Providers* lease resources from *Infrastructure Providers* and create services over that to negotiate with *end-user*. In turn, the *Broker* negotiate resources and services among *Infrastructure Providers*, *Service Providers*, and *end-user*. Finally, *end-user* are consumers of the services developed by *Service Providers*.

In another business model, proposed by Achemlal (ACHEMLAL, 2010), the *Service Provider* role is split in *Virtual Network Provider* (VNP) and *Virtual Network Operator* (VNO), as shown in the Figure 2.5. Whilst the former is responsible for constructing virtual networks with virtual resources from InPs or from another VNP, the VNO assume the responsibility for operating, controlling, and managing virtual networks. The existence of a VNP above the InP shows necessary to provide abstractions that let VNOs to consume resources from several InPs transparently, not worrying about resource convergence. The end user is not mentioned in the model, but as described in the article, its role is similar to the existing Internet users, with the addition of being able to attach their local network to the virtual network that provides the services according to their needs.

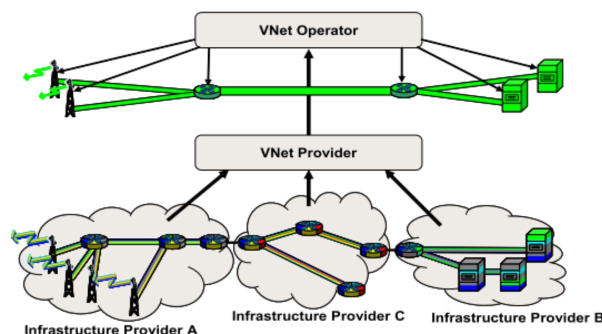

Figure 2.5: Network Virtualization Business Model proposed by 4WARD project. (ACHEMLAL, 2010).

From the *end-user* perspective, none of the presented models provides for the possibility to allow *end-user* themselves to develop, deploy, and manage novel network services in PVN environments. In *testbeds*, the use of resources is restricted for experimentation purposes and has exclusive access by researchers. Public Cloud environments usually offers network services ready to use, not customizable and neither programmable. The models found in the literature, proposed by Achemlal *et.al.* and Mosharaf *et.al.* follows the same trend when describe *end-user* just as service consumers. It is advocated in this dissertation that in order to really boost innovation in programmable virtual networks, there should be considered the participation of *end-user* also as a network service developer. Unlike current models, which neglect their potential and interest in developing innovative network services.

3 CONCEPTUAL SOLUTION

In previous chapters a historical overview of programmable network research has been presented. In addition, it was showed how this concept in conjunction with network virtualization culminated in Programmable Virtual Networking (PVN). Then, analyzing distinct PVN deployments, it was discussed about the role of end-users in current business models. In this opportunity was outlined the shortcomings of such business models, mainly in regards to the restrictive nature in the process of creation and deployment of novel network solutions. Such shortcomings have encouraged the proposals henceforth presented. At first, a novel business model is introduced. Then, in the following section, an open, scalable, and innovation friendly platform targeted at implementing such novel model is detailed.

3.1 Network Programming as a Service

Analyzing the network virtualization business model proposed by Boutaba, presented in section 2.2, it can be noticed a three-tier architecture. With Infrastructure Providers (InP) in the base, Service Providers (SP) in the middle, and End-Users on top. In order to improve end-users participation in the creation of network solutions, it is necessary to change their interaction with service providers. Toward this goal, it is proposed an integration of SDN architecture with the network virtualization business model defined by Boutaba. In such integration, shown in Figure 3.1, infrastructure providers take responsibility for the network infrastructure layer, also called forwarding plane or data plane. Service Providers in turn assume the other two layers of SDN, Control and Application.

At the Control Layer, Network Services are implementations of network functions with varying abstraction levels. Complex services, such as Firewall, Load Balancers, and QoS, as well as simple procedures for changing packet header parameters or dropping packets that match a specific pattern. Such services are available for end-users through special APIs that allows them to develop network applications, which are stored and executed in the Application Layer. Business Applications shown in Figure 3.1 are implementations of basic and recurrent services that can be available for applications of end-users. More details and examples of these applications and services will be given later.

In order to allow communication between SPs and InPs, two protocols are required, one for remote programming of network devices, and another for infrastructure management. The first allows customizing the behavior of each node in the network and hence can be used by network services to persist algorithm decisions. The latter allows managing virtual network infrastructures, which includes the creation, deletion and modification of links, nodes, and settings. It is also necessary a protocol to allow network applications to interact with network services. Whereas end-user applications depend upon such pro-

tocol, it should preferably be standardized, at least within the same service provider.

From the end-user point of view, the service provided by the service provider allows them to program networks. Due to this, the solution proposed here is called Network Programming as a Service (NPaaS). According to the network virtualization business model discussed, Service Providers are the responsible for developing and providing NPaaS directly or by means of Brokers to end-users, applying or not charges for that. In any case, services of network programming should be enough for end-users to develop network solutions in response to their own demands, and also enable innovation. Thus, in the remainder of this section is presented a set of requirements considered essential to a successful implementation of the NPaaS.

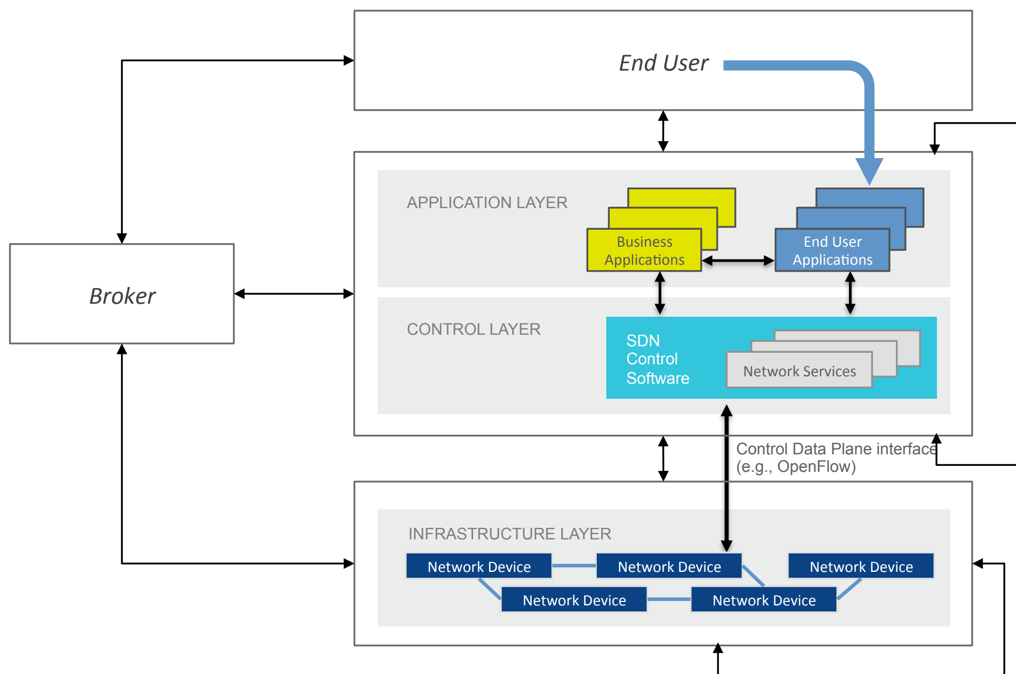

Figure 3.1: Network Programming as a Service Architecture

Network slicing: The communication protocol used between InPs and SPs should enable resources management of network, computing and storage. A subset of these resources is called ”slices“, a term generally used in the context of *testbeds*. Given a ”slice“ request, SPs are not responsible for the provisioning and configuration of resources. Nevertheless providers must translate requests from end-users according to protocols specification defined by each InP, which can differ from one to another. Web services can be used to promote this communication, due to its flexibility in respect to the connection of heterogeneous systems.

Abstraction models: The main goal of NPaaS is to encourage end-users to develop and deploy novel solutions in virtual networks. In doing so, one of the main challenges is finding a proper trade-off between programming granularity and simplicity. Network solutions that perform operations in low-level network parameters require more complex programming models, with less abstraction. On the other hand, simpler network solutions can be easily programmed with higher abstraction levels, *i.e.* less granular. Balancing such variables is important because complex models are not suitable for some end-users, especially those with less experience in computer networks. The opposite also applies; experienced users may require models with higher programming power. To illustrate, it

may be more complex to write an application using a procedural programming language than drawing a *workflow*.

Flexibility: Service providers are able to lease resources from many infrastructure providers. However, the communication protocol employed for resource requisitions and management may differ among providers. Due to this fact, service providers must employ a flexible approach in the communication with infrastructure providers. Supporting varying control plane technologies is also important, once each one provides a different set of network programming services.

Scalability: The amount of end-users may become large in some deployment environments. Therefore, the capabilities of service providers must be scalable in order to meet high demands. Service providers must maintain and manage control plane resources, which are used for storage and execution of network applications. It is worth noting that scalability described here is not related to the infrastructure of virtual network, which is related to the resources available in the infrastructure providers. It is related to the assets demanded by programming tools and techniques, which are related to expandible table rules, storage of network code, and scalable control plane. Another important requirement is to keep running state of network applications, regardless of infrastructure expansion or shrinkage.

Code Management: The life cycle of a network application in the context of Programmable Virtual Networking has four phases. *(i)* Development; *(ii)* Transference; *(iii)* Storage, and *(vi)* Execution/Management. In the initial phase, end-users should be able to develop the software – using abstraction models – as discussed in the second item. Once the application has been developed, its source code[1] must be in the right place for execution within the infrastructure of service provider. According to the approach either distributed or centralized, the source code must be transferred to one or several points. After that, service provider must store such code, ensuring availability and integrity. Finally, end-users must be able to manage such storage, as well as the execution of its source code.

Opt In/Out: There are two distinct computer networks involved in PVN scenarios. The virtual network itself, built by infrastructure providers, and the access network, used to connect end-users to the virtual networks. End-user interests in generating/receiving traffic data to/from virtual networks may vary. They can choose to offer novel virtual network services for consumers in the Internet, or deploy services in virtual network without any external connection. Anyway, Service Providers should support and be ready for creating such kind of connection, between internal and external networks points.

The set of requirements presented above guides the provision of NPaaS by Service Providers within the context of Boutaba's business model. However, different from the end-user role definition found in Boutaba's work, in NPaaS they are able to develop, deploy and manage their own network services in PVN. NPaaS promotes innovation in PVN by enlarging the set of people able to develop and deploy novel network service. As a side-effect, NPaaS introduces greater management complexity, once service providers must attend different and incremental network requirements.

In response to the increase in management complexity introduced by NPaaS business model, in the following section it is introduced a PVN management platform called ProViNet. It is targeted to sustain open PVN environments that implement NPaaS and follows the requirements discussed above. ProViNet allows service providers to consume

[1]The final product generated by the development process is called generically *source code*, which includes any variation.

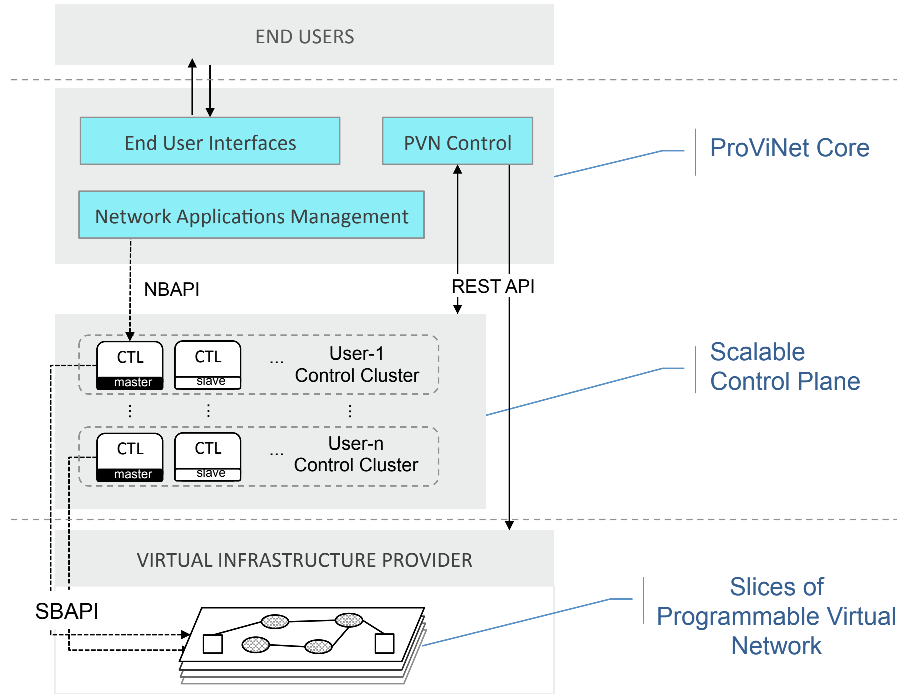

Figure 3.2: ProViNet Conceptual Architecture.

resources from different infrastructure providers and introduces a novel network programming strategy, more comprehensive and user friendly. To begin with, a general description of ProViNet architecture is given. Then, ProViNet modules are detailed in the remaining sections according to a bottom-up approach, in regards to network abstraction level, *i.e.*, from infrastructure issues until network programming strategies.

3.2 ProViNet: Programmable Virtual Network Management Platform

The architecture shown in Figure 3.2 illustrates the conceptual components of ProViNet platform. More precisely, it presents two major components *ProViNet Core* and *Scalable Control Plane* (SCP). There are three modules inside the *ProViNet Core*, which concentrate all business logic, access policy and configurations of the platform. SCP, in turn, provides the technology and resources necessary to host a scalable, robust, and high available virtualized environment, in which controllers can be quickly created, deleted, and migrated in response to the varying demand of end-users. From the top, end-users interact with ProViNet through *End User Interfaces* module. At the bottom, enabled by *PVN Control* module, ProViNet communicates with one or more Virtual Infrastructure Provider (VIP) in order to manage programmable virtual network infrastructures.

ProViNet was designed to enable NPaaS deployments, providing the necessary software functions to manage virtual network infrastructure, network applications development, and execution. In the following subsections, each ProViNet module is detailed, showing how it contributes to addressing such functions. On this occasion, technologies and implementation details are not important, but the approaches and strategies employed by each module to meet the requirements described in the previous section, such as flexibility, scalability, and code management. Technical details are covered in the next chapter, when a prototype of ProViNet is presented.

3.2.1 Managing Virtual Infrastructures

End-users interested in having a virtual network with custom services, developed by themselves, must have in mind that the first step is to build such network. *PVN Control* is the ProViNet module responsible for enabling the management of virtual network infrastructures, including the creation of virtual networks and configuration of control plane. Virtual links, hosts, and switches are technically provided by VIPs, but management of these components is performed by end-users, through the features available at the *PVN Control* module. From a virtual network topology previously described by end-users, this module send requests of creation and deletion of nodes, links, and hosts to VIPs. There are no restrictions in regards to topology structure, but different VIPs may define distinct policies. In a hypothetical example, a virtual network organized as a bus topology, with two hosts could be described in a special file and sent to ProViNet, which in turn, elaborate the necessary requests to the VIPs that will in fact build such network.

Virtual networks requested by ProViNet are kept in VIPs infrastructure, what makes them responsible for granting isolation between distinct virtual networks, availability of resources, and manageability. As mentioned, VIPs also provide virtual hosts. So, end-users must have access to such hosts in order to deploy services, softwares and configure it according to their necessities. ProViNet is compatible with multiple VIPs, allowing end-users to choose the VIP that provides better services, resources, and guarantees, such as reconfiguration of network links, resources expansion, and host migration. Furthermore, VIPs may vary in regards to the approach used to access the virtual network from outside their infrastructures, as discussed in *Opt In/Out* requirement in the last section. End-user may not be interested in having an isolated virtual network. External connectivity may enable their virtual network to communicate with the Internet or even private networks located abroad. This is also another important factor to be analyzed when choosing VIPs.

After virtual network provisioning, it is necessary to build a control plane and configure it to communicate with the virtual network. In response to end-users demands, *PVN Control* send requests to the *Scalable Control Plane*, ordering the creation of controllers. Then, information about the controllers created, such as IP address and location, are sent to VIPs, which must configure virtual network switches to report to the controllers created at the SCP. This approach provides scalability, once it is possible to create controllers according to SCP resource availability, which is supposed to be elastic. On top of that, controllers at the SCP are organized in clusters, one for each virtual network. So then, it is possible to set different control plane availability levels for distinct virtual networks. The High Availability (HA) strategy proposed here defines that each cluster must have at least one controller with the role *master*, which means, in production and active. In case of failure, another controller, with *slave* role must assume master's tasks.

3.2.2 Network Applications Management

In traditional network devices, applications that provide forwarding, routing or other functions are integrated in the devices, such as OSPF, IS-IS, and BGP protocols. As a distributed system, each device exchanges information in a switch-to-switch approach until the convergence or synchronization. In programmable networks that follow SDN architecture, network features are developed and deployed outside of network devices. In ProViNet, *Network Application Management* is the module responsible for storing, organizing, and running network applications. So, all innovation in terms of novel network features, developed by end-users, are deployed in this module.

Application and control planes, defined by SDN, are represented respectively by *Network Applications Management* and *Scalable Control Plane* modules. These planes run separately from each other. In doing so, control plane features are provided as Web Services through standard interfaces to external network applications. For example, if a network application needs to set a forwarding rule in a specific switch, it sends a remote call to the control plane, which in turn, interprets and executes the necessary communication with the virtual switches, finaly returning a response to the application. This approach differs from the one played in most SDN deployments, in which network applications are tightly coupled in control plane implementations (such as NOX (GUDE et al., 2008), Trema[2], and Beacon[3]). Avoiding such strong integration is important to prevent technology dependence, and foster the easy replacement of control plane technologies and network applications migration.

In regards to the standard interfaces mentioned, there is a common sense that establishes the use of "Southbound" and "Northbound" terms to describe API responsibility within the SDN architecture. In general, those APIs that enable interaction between control plane and forwarding plane are called "Southbound API" (SBAPI), and between control plane and network applications are called "Northbound API" (NBAPI). 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, NBAPI is employed in requests between network applications, at the *Network Application Management*, and master controllers (*CTL*) at the *Scalable Control Plane*. SBAPI in turn enables the communication between controllers and virtual switches in the virtual infrastructure provided by VIPs.

Remarkably, the connectivity between SCP – control plane – and virtual networks in VIPs infrastructure – forwarding plane – is critical (LEVIN et al., 2012). Virtual network behavior is defined by the control plane. So, failures in receiving or sending network control information may lead to inconsistencies in routing protocols and services. Several strategies can be used to overcome this situation. One of them is to previously configure virtual switches with a standard behavior, which is performed in case of control plane unavailability. Examples of standard behavior are to drop every arriving packet and to send back a special response to the packet origin. Certainly variations of these strategies are also possible. Failures could also be avoided if there is good network connectivity between such planes. With this in mind, using network links exclusive to transport control traffic is also valid.

3.2.3 Network Applications Development

Network services must be aligned with business demands, either from industry, academy or other organizational type. Attending such demands involve tuning and deploying network device and services. Historically, network services were bought altogether with devices, as a full package. In order to have a new feature in the network device, companies should either update software or upgrade hardware components of such device, which is usually compatible just with the same vendor product line. Such reality shows that for decades consumers of network services and devices were subject to vendors interests. As a consequence, specific demands are not met, or require great efforts to develop ad-hoc solutions, usually with low performance.

With programmable network devices, it is expected a rise in the amount of network

[2]http://trema.github.io/trema/

[3]http://openflow.stanford.edu/display/Beacon/

software solutions, enabled by the use of open and programmable devices. However, the development of such network softwares, performed in traditional networks by network device vendors, is now responsibility of someone else. ProViNet and NPaaS represent a line of thinking in which end-users must take part of such responsibility. But, considering their heterogeneous level of knowledge, in regards to network concepts and technical foundations, it may be initially hard to translate their business necessities into network softwares. It is strictly necessary provide a better and faster integration of end-users business demands and network applications.

Considering that applications can be seen as a sequence of processes, and that, similarly, a *workflow* defines a sequence of processes, it is proposed here the use of Business Process Modeling Notation (BPMN) as a language to develop network applications. In this way, end-users are able to develop network applications (*workflows*) by simply arranging graphical elements defined by BPMN. In other scenarios BPMN is used by executives, directors, and strategists to specify business process using Business Process Diagrams, which are based in flowcharting techniques. At a first glance, BPMN looks too abstract to represent network application routines, such as analyzing packet header values, or setting QoS parameters. However, the separation of network features in different abstraction levels, defined by SDN concept, have allowed the use of high level languages to manage and control network behavior. Using a bottom-up approach, the following topics give an overview of ProViNet strategy to enable the employment of *workflows* as network applications to control network devices behavior.

Forwarding Plane – In programmable devices defined by SDN, the action to be taken upon each packet or flow is defined as forwarding rules, which are manageable by external devices, called controllers, located in the control plane and deployed in the SCP of ProViNet. This way, through SBAPI, forwarding device behavior can be defined according to higher-level decisions taken at the control plane.

Control Plane – From one side, applications running in controllers at the SCP use SBAPI to communicate with forwarding devices at VIPs infrastructure, receiving reports about network traffic and sending forwarding rules in response. From the other side, the control plane provides the necessary abstractions to allow applications to use higher-level calls to program and configure the network. Due to this “translating” service performed by the control plane, it is common to call such plane as Network Operating System (NOS)[4]. Similarly, Operating Systems translate high level calls from applications to system calls. So, for instance, if an application intends to enable a point-to-point communication, instead of elaborating and setting forwarding rules individually to each network device along the way, it sends to the control plane just a simple request with two parameters, the initial and final point IP addresses. The control plane, in turn, elaborates and sends the necessary forwarding rules to devices. Similarly, the control plane could provide abstractions to other services such as Firewall, Load Balance, QoS, or simple change of packet parameter header.

Application Plane – At this level, end-users must be capable of developing network applications using a simple and easy understanding language. To accomplish this goal, *End-User Interfaces* module works in conjunction with *Network Application Management* to provide the necessary tools and graphical elements to support network programming. As explained in the last topic, NOS makes available a set of services for use in exter-

[4]In the past, the term network operating system referred to operating systems that incorporated networking (e.g., Novell NetWare), but this usage is now obsolete. Such term was resurrected to denote systems that provide an execution environment for programmatic control over the full network.

nal applications. This way, such services must be firstly described and then imported in ProViNet as graphical elements, becoming available into the network application interface. In most NOS implementations, these services are implemented as Web Services, so it is possible to use Web Services description files (such as WSDL and WADL) for the purpose here described. After the inclusion of service descriptions in ProViNet, the *End-User Interfaces* module builds a graphical programming environment, associating an object to each service, in addition to the regular elements defined in BPMN.

The graphical elements defined in Business Process Diagrams are sorted in *Events*, *Gateways*, *Activities*, *Connecting objects*, *Swimlanes*, and *Artefacts*. *Events* are used to represent situations that happens during the process, and usually have a cause (trigger) and an impact (result). In turn, *Gateways* are employed with the aim of either split or merge process flows, optionally based in conditions. When necessary, *Activities* are meant to represent works or tasks that must be performed during processes. In an effort to have these elements coherently connected in a logical sequence (essential to support the workflow execution), BPMN defines some *Connecting objects*, such as the ones used for representing normal, conditional and sequence flows, message flows, and associations. *Swimlanes* are used to represent the assignment of different responsibilities within or among organizations, or systems. At last, *Artefacts* serve to aggregate additional information about processes.

As discussed in *Abstraction models* requirement, in Section 3.1, NPaaS solutions must provide to end-users a programmatic approach to develop network applications in different granularities. Thus, end-users with varying experiences in computer networks and programming languages are able to develop novel network applications. The approach of using BPMN as a programming language to develop network applications is straightly aligned with this requirement. *Workflow* processes within ProViNet platform are represented by graphical elements that, in turn represent control plane services. In doing so, the granularity of network applications developed as *workflows* depends on control plane services available. For example, high-level network services, such as Firewalls, could be available as well as low level services, such as setting forwarding rule to a specific kind of network traffic. This flexibility results in a large range of services and possibilities of process combinations when creating *workflows*.

In addition to network services available in the control plane, it is possible to use other resources of Web Services. In the *workflow* depicted in Figure 3.3, there is a simple example of network application, in this case an intrusion detector that sends an email to network administrators in case of intrusions. Note at the top a very small set of services available. Conversely, in real deployments this amount may increase significantly, with the addition of control plane services and other Web Services.

3.2.4 Network Applications Execution

There are currently available several tools for *workflows* edition and orchestration. They are known as Enterprise Service Bus (ESB). Some of them are open source, such as Intalio, but the majority, such as Oracle ESB and IMB Websphere ESB are proprietary. Notwithstanding, none of them consider the peculiarities involved in executing *workflows* as network applications. For this reason, it is introduced within ProViNet a special *workflows* editor and orchestrator. As depicted in Figure 3.4, several services are listed in ProViNet GUI interface. Not only those available in the NOS but also the services provided by network Service Providers and by external Web Service providers, such as Google and Yahoo!. Interacting with such interface, end-users are able to manipulate

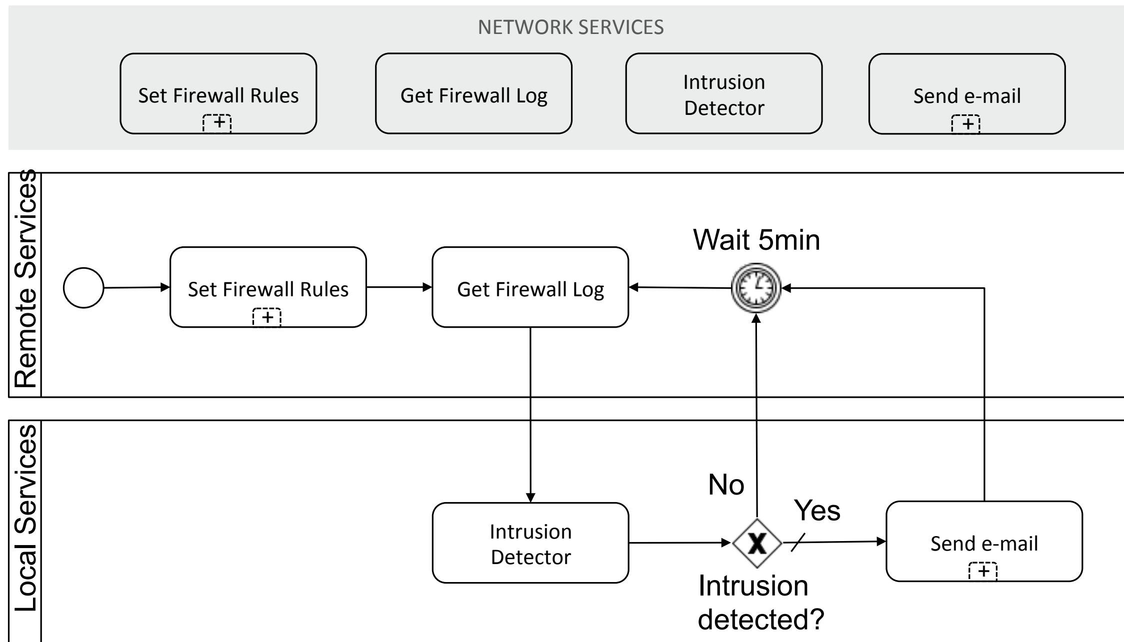

Figure 3.3: *Workflow* implementation of a simple intrusion detector application.

such services and BPMN graphical elements to draw *workflows*. In order to execute the sequence of services and logical operations described in *workflows*, *Network Application Management* module adds such services into a *Services Execution Queue*.

Services in the *Services Execution Queue* are forwarded to controllers by *NBAPI Dispatcher*, a component of *Network Application Management* module. One by one, services are firstly edited to have as destination address, a master controller inside the *Control Cluster* related to the virtual network that such network application is being developed for. As a result of the infrastructure provisioning process, the network address of such controllers (composed by IP and Web Service URL reference) are available in the ProViNet internal database. Controllers, in turn, process the requests of each service either by using SBAPI to communicate with the nodes that form the virtual network, or by consulting internal system parameters and statistics. Controllers have a global view of virtual network topology under its control. Because of this, some information requires just an internal look up, such as running state of links and nodes, as well as statistics about data traffic.

Some services require input parameters, such as point-to-point connectivity, which requires at least the specification of initial and final points. Such parameters can be either provided through GUI interface, or forwarded from the response of a previously executed process. Partial results can be displayed in *ProViNet Web GUI* after individual process execution, or after executing all processes in the Queue. An initial aproach of *workflows* validation were developed. Currently, it just check the multiple or absence of starting and end points.

Some BPMN logical elements require special computation, such as *Timers*, *Conditionals*, and *Sub-process*. They may define a nonlinear execution, what makes impossible using just an execution queue to control the sequence of processes. For such special cases, distinct algorithms are employed. Simple *Conditionals*, for instance, makes necessary to split the original queue into two other queues, one to track the processes after a true condition, and another after a false condition. *Workflow* execution algorithms are well discussed in the literature and several variations have already been proposed (SCHUSTER,

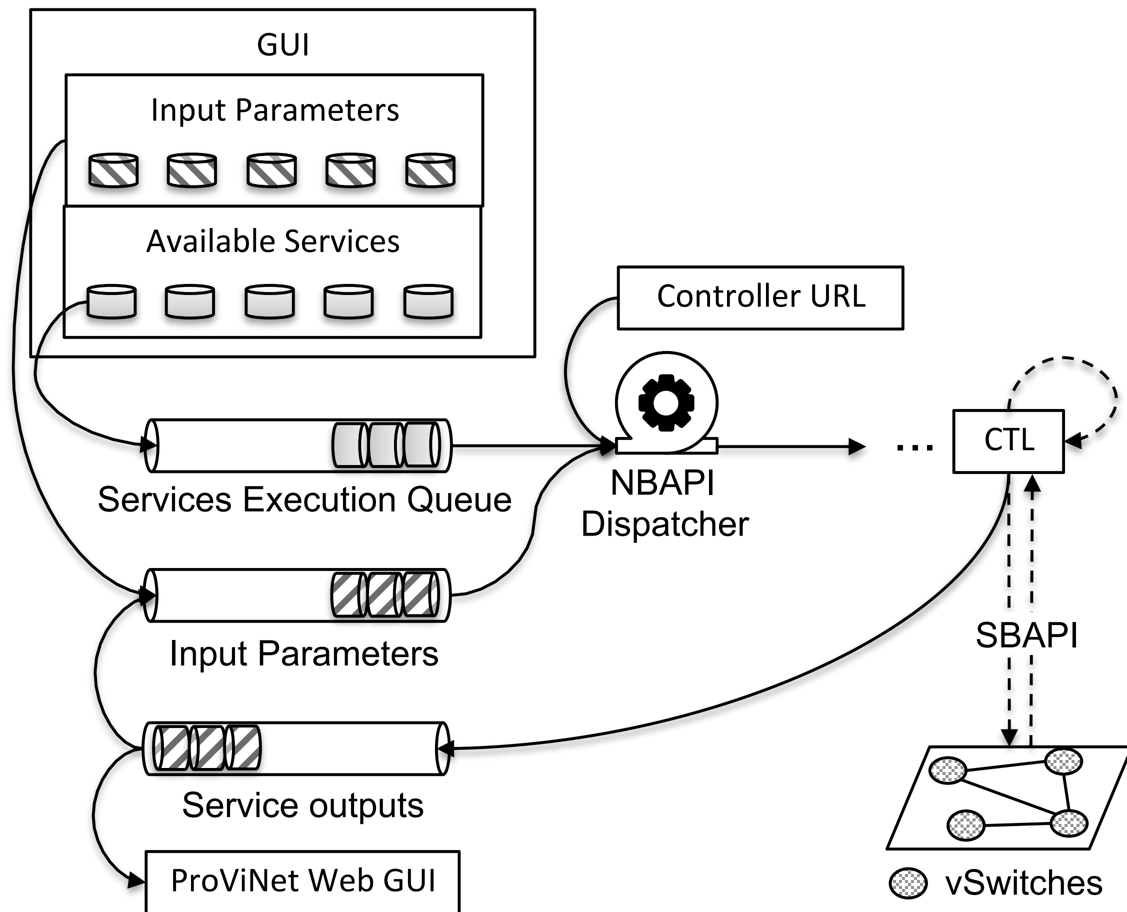

Figure 3.4: Network Application Execution approach.

2005). A discussion about these algorithms is not in the scope of this dissertation, once that irrespective to the one employed, final results are the same, differing in performance only.

Each *workflow* represents a network application, or part of one. Such representation can be exported to other file formats and used by other end-user as modules into their applications. Furthermore, *workflows* files can be negotiated as softwares, inducing the emergence of a new market. In which end-users and network companies are able to compete for better and innovative network solutions. With some adaptations, *workflows* developed within ProViNet platform can became compatible with already existing *workflow* Enterprise Service Bus. Such compatiblity would enable a faster adoption of NPaaS.

4 PROTOTYPING PROVINET PLATFORM

In the last chapter was presented NPaaS concept as well as ProViNet platform. Aiming to demonstrate the feasibility of the conceptual architecture detailed in Section 3.2, in this chapter a prototype of ProViNet is detailed. In contrast with the discussion presented before, details about implementation are given here. The prototype implementation described in this chapter regards ProViNet Core only. The remaining components depicted in the architecture of Figure 3.2, such as *Scalable Control Plane* and *Virtual Infrastructure Provider* are external components and just interact with the prototype. ProViNet system is Web based and is accessible through browsers in devices such as PC, Tablet, and Smart Phones. All modules of ProViNet were implemented using the framework Django 1.4.3, the programming language Python 2.7.3, and PostgreSQL 9.1.6 database management system.

Prototype components presented hereinafter follow the same order that an end-user would follow to use ProViNet features. The most initial step is to register and login into the platform authentication module. But, this step is very simple and self-evident. Then, the first step described is in regards to programmable virtual infrastructure provisioning process, which includes defining control plane configuration and virtual network requisition. After that Network Application management issues, such as developing and executing are detailed.

4.1 Programmable Virtual Infrastructure Provisioning

Infrastructure management is a critical feature in any network management software. Although ProViNet management platform does not interact directly with physical network infrastructures, it provides to end-users the necessary features to manage virtual networks through APIs. Such APIs allows ProViNet to communicate with external entities that indeed provide virtual infrastructures. Programmable Virtual Networks are composed by two layers of infrastructure: virtual networks and its control plane. As will be clarified in next subsections, it is proposed here to keep both layers in virtual environments. The first, named *Scalable Control Plane* is detailed firstly in the next subsection. Then, in the subsequent subsection, the second layer is presented, which is controlled by Virtual Infrastructure Providers (VIP) and managed by ProViNet.

4.1.1 Scalable Control Plane

In ProViNet, the control plane is implemented in a virtualization environment called *Scalable Control Plane* (SCP), as shown in Figure 4.1. Such environment provides features similar to the ones found in Cloud Computing environment. Such as virtual machine

creation, deletion, and migration. The SCP was deployed with XenServer (SYSTEMS, 2010), and the communication with this server were enabled by XenAPI, both provided by CITRIX. Using this API, *PVN Control* module is able to perform Remote Procedure Calls (RPC) to XenServer. This technology allows the configuration of a set of XenServers to work as a "pool" of resources. This way, a scalable environment is achieved, once the amount of servers is expandable. Within such scalable "pool", virtual machines are able to move among servers aiming to save energy, reduce delays, or even balance servers demand, moving VM to less busy servers.

Virtual machines created in the SCP runs an OpenFlow compatible controller technology, which, in current ProViNet implementation, is the FloodLight[1]. Another control plane software cold be used, such as Trema and Open DayLight. FloodLight was chosen because it support Representational State Transfer (REST) and provides plenty of services, such as Firewall, Point-to-Point connection factory, flow rule configuration, and provisioning of network state information. Besides that, it is possible to develop additional services and deploy it in FloodLight controllers.

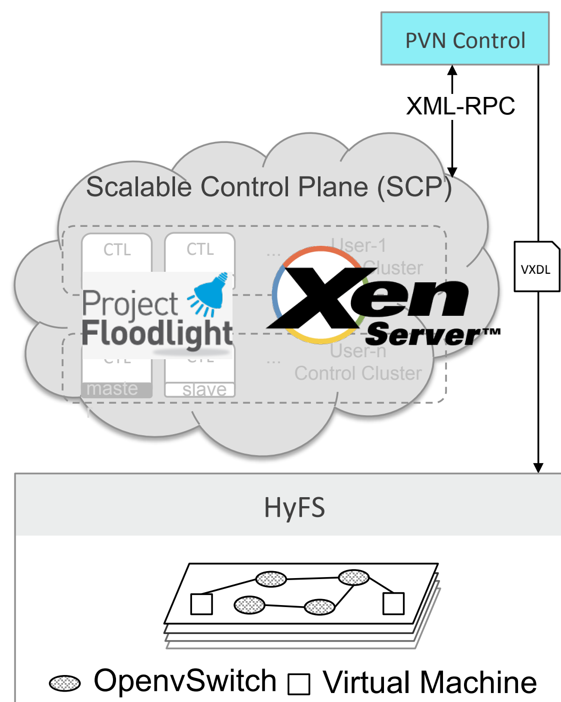

Figure 4.1: Scalable Programmable Virtual Network Infrastructure

Interacting with the ProViNet management interface, shown in Figure 4.2, end-users are able to add or delete controllers in the control plane. Each controller creation or deletion command is followed by a request from *PVN Control* to the SCP. For the sake of agility, a VM is previously configured with FloodLight and deployed in XenServer. This way, when end-users request a controller by clicking in the button *Add Controller*, *PVN Control* triggers a request[2] to XenServer Pool requesting to clone such VM. There is also in this VM a script responsible for starting FloodLight services shortly after booting the OS. Then, with the VM already running, *PVN Control* triggers another call to check VM information, such as IP address, and running-state, which are persisted in a previously

[1] http://www.projectfloodlight.org/floodlight/
[2] session.xenapi.VM.clone(vm_base_name, vm_cloned_name)

Figure 4.2: ProViNet Management Interface sreenshot.

configured database of ProViNet. In order to keep updated the *Status* column of ProViNet management interface, running-state is checked every minute.

Controllers requested by end-users take part of high available control plane, the *Control Cluster*. Considering that OpenFlow is the technology adopted in forwarding plane devices, it becomes able to communicate with multiple controllers, which in turn assume one of the following roles: *master*, *slave*, or *equal*. These roles permeate two kind of policies. *(i)* One defines that switches must report events to the *master* properly configured and active, which in turn reply with the appropriate flow rules. In case of failures in the *master* controller, another controller previously configured as *slave* may assume *master*'s role. While in *slave* state, controllers are able to read reports from switches, but unable to reply. *(ii)* Conversely, if all controllers assume *equal* role, all of them are able to read devices' report and write flow rules in response. In this case, a stateful synchronization between controllers is required to ensure coherence of decisions.

If ProViNet is configured with the policy (*i*), the first controller requested assume *master* role, and the following, *slave* role. This way, if *master* controller fail, its status will change from ”running“ to ”turned off“, and a randomly chosen *slave* controller will take responsibility of *master* role. However, if policy (*ii*) is set, every controller will be created with the role *equal*. In this case, requests from applications to the control plane will follow a Round Robin algorithm to choose the targeted controller. Each *Control Cluster* can be associated with at most one virtual network slice. The creation of such slice is discussed in the next subsection.

4.1.2 Virtual Infrastructure Provider

A complete PVN environment is composed by a virtual programmable infrastructure, network control software, and network applications. In this section it is discussed about the former, which is comparable to SDN forwarding plane. *PVN Control* module is the ProViNet component that provides the necessary features to manage such virtual programmable infrastructure. As depicted in Figure 4.1, *PVN Control* communicates with Virtual Infrastructure Providers (VIP) through HTTP calls following REST architectural style.

ProViNet prototype was initially configured to work with a platform called Aurora (WICKBOLDT et al., 2012), which is actually the only VIP, based in private Cloud, that supports custom programmable virtual network topology requests, and is compatible with ProViNet. In addition to virtual hosts, this platform creates virtual network topologies using virtual instances of switches, actually OpenvSwitch. In addition to that, Aurora is able to configure such virtual instances to receive control information from external controllers. In Programming Network as a Service business model, VIP role could be assumed by existing infrastructure providers.

In order to enable resource allocation from multiple providers, the communication between ProViNet and VIPs is implemented as plug-ins, one for each provider. But, as an effort to keep interoperability and compatibility with ProViNet services, VIPs must comply with some requirements, such as: *(i)* support a communication protocol that grant authenticity, integrity, and reliability; *(ii)* be able to technically manage programmable switches, thus enabling the building of Programmable Virtual Network (PVN) topologies, finally, *(iii)* VIPs must be able to attend requests of arbitrary network topology, regardless of the physical one.

Currently, ProViNet and Aurora exchange topology information, such as links, hosts, and virtual switches, by means of an XML file. These three elements are specified along with specific parameters, such as link bandwidth and latency, host capacity in regards to memory, storage, and operating system, and virtual switch ports. There are several description models available to build such file, such as Rspec (GENI), NDL-OWL (RENCI), NMC (OGF) and Virtual Resources and Interconnection Networks Description Language (VXDL) (KOSLOVSKI; PRIMET; CHARãO, 2008). VXDL is employed in ProViNet because such language is able to represent in simple and clean XML all details of a virtual infrastructure, including virtual machines, switches, and links. In order to better understand VXDL, the following sample is highlighted, which describes a simple topology with 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>
```

VXDL specification does not define tags to represent controllers associated to the virtual switches described (which are programmable according to SDN architecture, *i.e.*, with external control plane). Defining controllers is critical to enable high-availability approaches described in Subsection 4.1.1. In order to overcome such VXDL limitation, it is proposed some additional XML tags. `controllerList` represents a list of controllers and is identified by `id` attribute. Within this list, multiple `controller` tags can be nested, and each one has the attribute `type=""`, which is used to inform whether the controller in the list is *master*, *slave*, or *equal*. In addition, configuration parameters such as port, protocol, and IP address required by forwarding devices are represented by tags hierarchically positioned bellow `controller`. Taking the same network topology example given before in VXDL, the representation of two controllers would be added as follows:

```
...
<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>
...
```

In the current version of ProViNet such VXDL file must be elaborated manually by end-user, though, without information about control plane. With such file ready, they upload it through a form in ProViNet interface. Then, *PVN Control* use the controller information previously obtained after control plane provisioning, described in Subsection 4.1.1, to create control plane related tags and add it to the VXDL file. Only after that the file is sent to VIP in an HTTP request.

After properly instantiation and configuration of the virtual infrastructure, the VIP sends back to ProViNet an HTTP answer with request results. In case of failure, the cause is specified. In case of success, *PVN Control* redirects the end-user navigation to the Aurora management interface shown in the screenshot depicted in Figure 4.3. In which end-users can visualize information and control the virtual resources. Hosts, for example, are listed below *VMs* label, altogether with control commands to change host running state, migrate hosts to different locations as well as access host console.

VMs

Name	Host	Memory	VCPU	State	Action
Node0-provinet-10	QEMU/KVM at hyfs3	128.0 MB	1	running	XML Shutdown Migrate Console
Node1-provinet-10	QEMU/KVM at hyfs3	128.0 MB	1	running	XML Shutdown Migrate Console
Node2-provinet-10	QEMU/KVM at hyfs3	128.0 MB	1	running	XML Shutdown Migrate Console
Node3-provinet-10	QEMU/KVM at hyfs3	128.0 MB	1	running	XML Shutdown Migrate Console

Virtual Routers

Name	Host	State	Action
Switch1-provinet-10	QEMU/KVM at hyfs3	Active	Delete
Switch2-provinet-10	QEMU/KVM at hyfs3	Active	Delete

Virtual Links

Start	End	State	Action
Node0-provinet-10 - net0 (00:00:00:00:00:99)	Switch1-provinet-10 - port0 (00:00:00:00:01:03)	Created	Delete
Node1-provinet-10 - net0 (00:00:00:00:01:00)	Switch1-provinet-10 - port1 (00:00:00:00:01:04)	Created	Delete
Switch1-provinet-10 - port2 (00:00:00:00:01:05)	Switch2-provinet-10 - port0 (00:00:00:00:01:06)	Created	Delete
Node2-provinet-10 - net0 (00:00:00:00:01:01)	Switch2-provinet-10 - port1 (00:00:00:00:01:07)	Created	Delete

Figure 4.3: Aurora: Resource access information screenshot.

Virtual Routers and *Virtual Links* are also shown to end-users. From this interface they can delete virtual links and routers. Conversely, any addition of resources or topology change requires a new VXDL elaboration and submission to ProViNet. In further versions of ProViNet and Aurora, such changes will be available in a graphical mode. In addition to the deleting option, in current versions end-user have a general overview of virtual infrastructure topology, as shown in Figure 4.4. Moving the objects, end-user can organize the topology in order to have a better understanding.

4.2 Network Application Management

In Section 3.2.3 of the last chapter, it was discussed about reaching a broad range of end-users, with heterogeneous experience in computer network and software development. To this end, it was proposed the use of BPMN to develop *workflows* as network applications. The ProViNet prototype part that implements such functionality is presented in this section. Prototype details are split into two subsections. In the first one, an interface, based in flowcharting techniques, created to allow end-users to develop *workflows* graphically is presented. Then, in the following subsection, details about how ProViNet enable the management of *workflow* execution and present outputs are given.

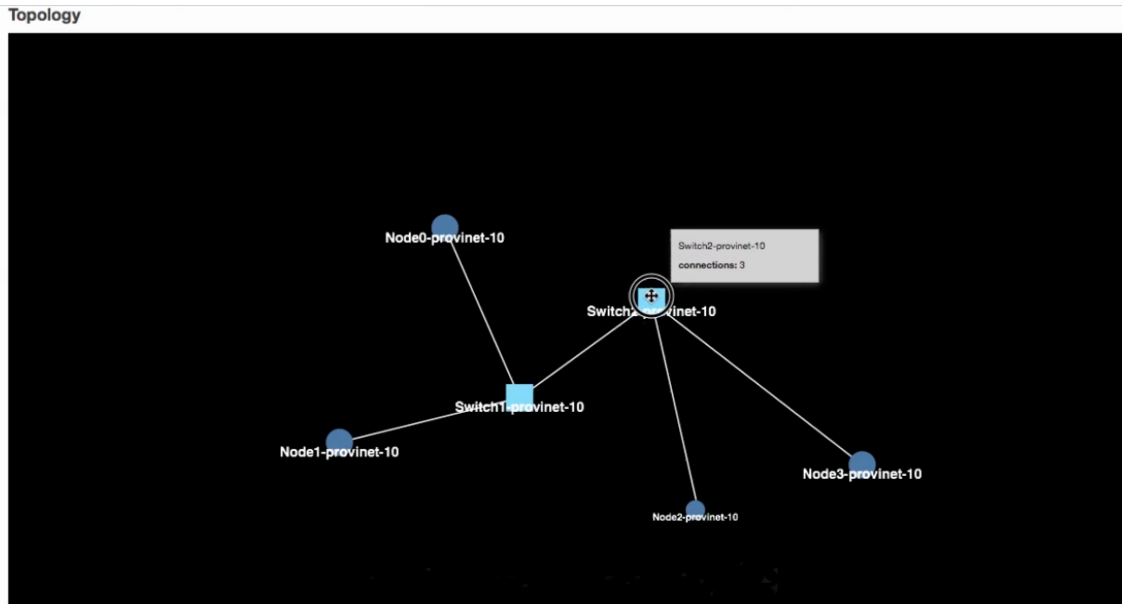

Figure 4.4: Aurora Virtual Network topology screenshot.

4.2.1 Developing

Before describing the network application development, it is important to clarity the procedure of bootstrap and configuration required by ProViNet. According to the *workflow* development approach discussed in Section 3.2.3, a set of BPMN graphical objects are presented in the interface, such as *Events*, *Gateways*, *Activities*, *Connecting objects*, *Swimlanes*, and *Artefacts*. In face of such graphical objects, end-users are able to create *workflows* that represent complete network applications. However, in ProViNet platform, *Activities* represent external Web Services, either those provided by control plane technology or other providers. So, in order to show to end-users a list of services, it is necessary to register them into ProViNet.

To configure services in ProViNet, the Web Services description file is necessary. Such file is either a Web Services Description Language (WSDL), for SOAP servers or Web Application Description Language (WADL) for REST servers. Service providers usually keep such file publicly available. ProViNet platform support multiple service description files, either from control plane technologies or public Web Service providers. As an example, it is possible to register FloodLight, Yahoo!, and Google mail service. Then, based in the services registered, ProViNet mount the graphical interface shown in Figure 4.5. Interacting with such an interface, end-users can drag and drop the graphical representation of services and BPMN components from the list in the left side to the center, composing *workflows*. The development of such interface was possible with the use of WireIt javascript library.

To better understand ProViNet network application, it is presented also in Figure 4.5 a simple network application. In such application the first object is the starting point, defined by BPMN, then three control plane services appears. The first check Firewall status, the second change it to ”Enable“, and the last check it again. The output of each service is shown at the right side in the ”Output Terminal“ box. Certainly more complex services could be developed, using different services and achieving deeper network details, such as packet header analysis. But the example presented is enough to visualize a network application developed in ProViNet graphical network application developer.

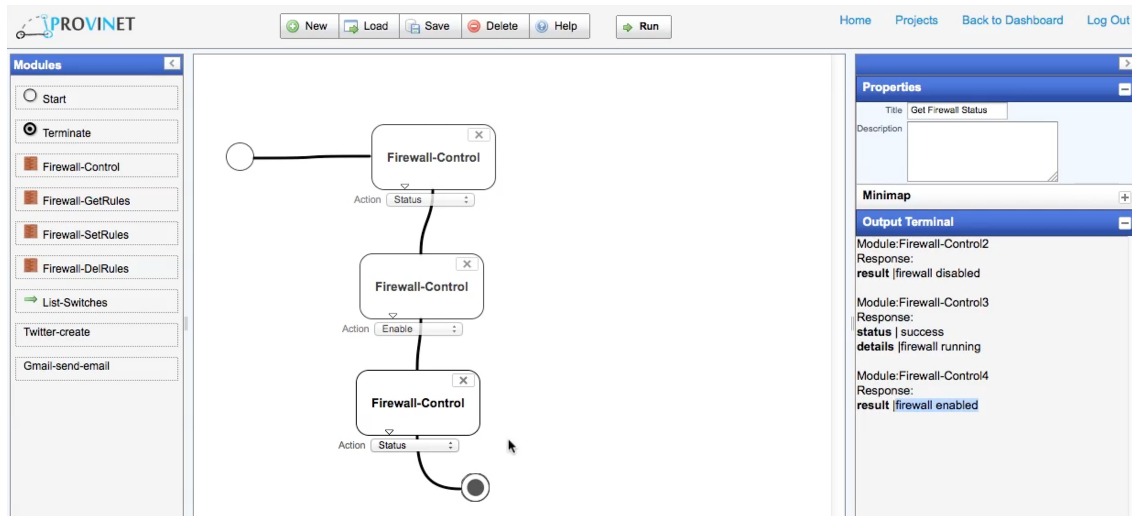

Figure 4.5: ProViNet Graphical Development Interface.

4.2.2 Executing

Regular programming languages are usually interpreted or compiled and executed. During network application development in ProViNet, WireIt library keep network application graphical representation also in XML or JSON format. Enabling end-users to save and load applications. Although neither XML nor JSON are programming languages, ProViNet interpret it as such, because it represent *workflows*, which are developed using BPMN visual programming language. Ideally, ProViNet should have a converter of WireIt output formats to Business Process Execution Language (BPEL), which can be executed directly.

Once BPEL converter is not yet created, ProViNet has its own XML/JSON interpreter. Discussed in Section 3.2.4 and illustrated in Figure 3.4, it is composed by execution queues and an HTTP dispatcher. Parsing the xml/json file, ProViNet follows the path defined in the network application, adding the *Activities* and *Gateways* in arrays. Each service is represented by an ID that was given in the moment of registration in ProViNet. So there is in the database associations of IDs and service references. After parsing the whole file, an executor function is started, which consumes the array elements in the same order that it was added. The service associated to each ID is retrieved from database and forwarded to a HTTP request dispatcher. If it is a control plane service, HTTP dispatcher will also retrieve the targeted controller address. Then the request is send to the destination address that is composed by the controller address concatenated with the service reference. Otherwise, the full service reference is used as destination.

For each service dispatched, a reply is expected. It could be just an acknowledge or an result of the request service. If it is a result, ProViNet adds such response in another array, keeping track of the requester service ID to which the reply was given. Finally, all results are presented in the output area located at the right side of ProViNet network application graphical developer. In the example shown in Figure 4.6, an application check Firewall status, then set a new Firewall rule, and finally check status again. In this example all services are targeted to the control plane, which in this experiment was the FloodLight controller.

As defined in FloodLight Firewall REST API[3], ALLOW and DENY examples of

[3]http://www.openflowhub.org/display/floodlightcontroller/Firewall+REST+API

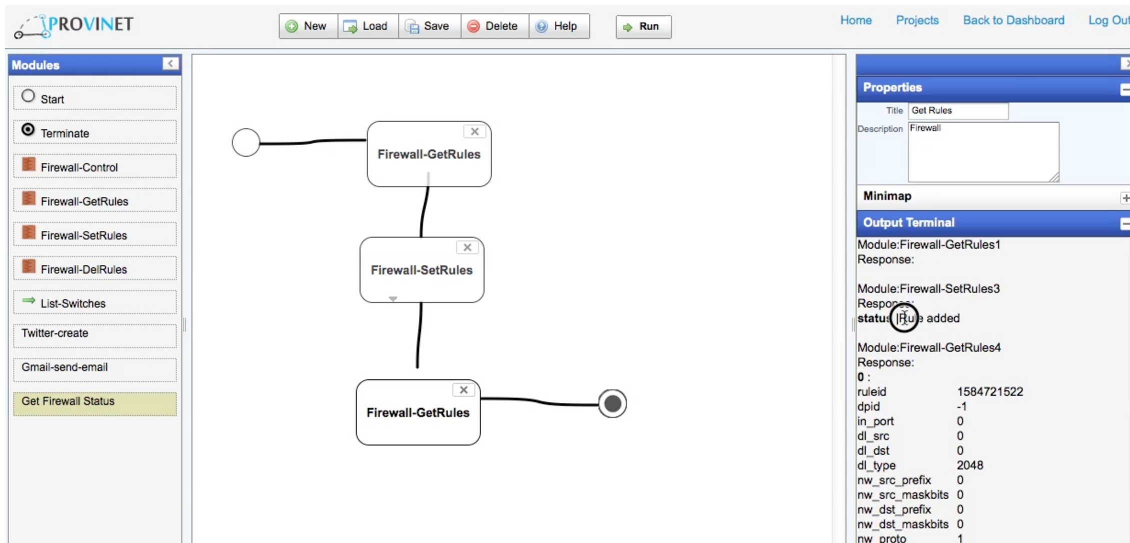

Figure 4.6: Network Application that changes Firewall status.

”Firewall Set“ requests are given respectively below:

```
var ALLOW = '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32"}';
curl -X POST -d $ALLOW http://localhost:8080/wm/firewall/rules/json

var DENY = '{"src-ip": "10.0.0.4/32", "action":"DENY" }';
curl -X POST -d $DENY http://localhost:8080/wm/firewall/rules/json
```

In the first example, all traffic that the IP source address match ”10.0.0.3/32“ and destination IP address match ”10.0.0.7/32“ is allowed. In the second, all traffic with destination IP address matching ”10.0.0.4/32“ is blocked.

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. Drawing *workflows* is a simple task and may not represent much complexity to end-users. Regarding the granularity, it is directly related to the services available by controller technologies. For instance, with the current technology deployed, which is Floodlight, available services allow the development of applications to control network from OSI Physical Layer (L1) to Transport Layer (L4). In order to give a better overview of ProViNet prototype it is available at YouTube a demo video[4].

[4]http://www.youtube.com/watch?v=IAoTHAQmFnQ

5 EXPERIMENTAL EVALUATION

In Chapter 3 it was presented NPaaS business model and ProViNet platform. At that moment it was mentioned that ProViNet is a platform designed and developed to support NPaaS business model. In this chapter, qualitative and quantitative evaluations are presented in regards to ProViNet. Firstly, after the definition of some criteria the platform is qualitatively evaluated and compared with some of the network programming solutions described in Section 2. Then, the platform is evaluated based in performance analysis of ProViNet prototype, which was employed in an use case of physical infrastructure migration to Cloud Computing, a recurring situation in recent times. The qualitative comparison demonstrates the uniqueness of ProViNet, while the performance analysis illustrates that such platform and the prototype developed shows acceptable performance results.

5.1 Qualitative Comparison

Considering end-users point of view, some features considered essential to make network programming simpler, manageable, accountable, and user friendly are characterized. After that, in Table 5.1 remarks about the network programming solutions presented in Chapter 2 are given in contrast to NPaaS business model. Worth mentioning that NPaaS is not evaluated, once that business models are in general too abstract. However, once ProViNet implements all NPaaS requirements, validating such platform it is expected that the business model is also valid.

End-user Access Policy: Regarding end-user access limitations, network programming solutions are classified into: *Inaccessible*, for the cases when end-users are not considered in the procedures of network application creation, deployment, and management. *Restricted* means that end-users are able to create and deploy their own network solutions, though they must request special permissions in advance. In *testbed* environments, for example, users are usually authorized to use their resources after justification analysis, and during predefined periods. And, finally, network programming solutions could define end-user policy as *Accessible*, *i.e.*, they are able to freely require virtual resources, develop, and deploy custom network solutions.

Resource Organization Method: Different infrastructures could be used to instantiate virtual networks and virtual machines. Generalizing, such resources could come from Data Centers, Universities, or Research Laboratories. Testbeds, for instance, follows a Federation model, in which Research Laboratories and Universities can register in the federation as long as they contribute with resources to the global network deployment. Proposals are then generalized into two categories, *Federation* or *Data Center* organization method.

Network Topology: Some network programming solutions enable end-users to re-

quest virtual network and virtual machines disposed in custom topology, totally independent from the physical substrate present in Infrastructure Providers. In other solutions, end-users are able to request virtual network topologies limited to the physical distribution of nodes and links. Finally, it may be of interest that network topologies do not differ from a predefined set of topologies, such as Star or Bus. Concerning this topic solutions are classified into *Physically Dependent*, *Physically Independent*, or *Virtually Limited*.

Resource Description: Network programming solutions may vary in regards to the approach used to describe virtual infrastructure. It could be employed Virtual Infrastructure Description Languages (VIDL) to represent the network topology and be sent to virtual infrastructure providers. Alternatively, solutions could use a graphical interface to enable end users virtual network requests. The proposals are classified whether *VIDL* or *Graphical*.

Resource Request Method: Graphically drawing Slices has the drawback of one-by-one definition, what means that end-users must choose individual topology components and configurations one at a time. Such approach may be a problem in larger slices. Conversely, if the user can just send the whole slice definition at once, using a VIDL file, it may be faster. Certainly, there are a previous moment when end-user prepare the file, but such process could be easily automated. So the proposals may vary in *One-by-one Requests*, *At Once Request Submission* and *Both* support.

Granularity or Programming 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. Solutions are able to provide granularity levels from Physical Layer (*L1*) to Application Layer (*L7*) of OSI reference model.

Control Plane Management: In the SDN architecture the control plane runs out of network devices, more precisely in an external machine (called controller). This controller could be placed in a physical or virtual environment, since it has network access to the forwarding network device under its control. Furthermore, most forwarding devices support High Availability (HA) feature, accepting the configuration of more than one controller. Many network programming solutions assigns to users the responsibility to manage the control plane. However, if such control is assumed by the network programming solution, users can concentrate on network applications development. Considering the exposed, solutions are classified by its level of management. If the solution supports HA, creation of controller instances on demand and automatized forwarding plane configuration it is classified as *High Level*. If it support just two of the mentioned management features, it has *Middle Level* and if just one is supported it is considered *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 customer. And finally business models such as NPaaS focus in end-users.

The proposals presented in the Table 5.1, OFELIA Control Framework, CITRIX DVS and ProtoGENI were omited from Section 2 because it helps understand our solution, while the papers described there helps understand the problem. In addition to them, ProViNet is compared to a generic solution that is implemented with the integration of OpenStack and Quantum technologies. OpenStack is a platform for managing Cloud en-

Feature	**ProViNet**	**OFELIA Control Framework**	**ProtoGENI**	**CITRIX DVS**	**OpenStack & Quantum**
End-user Access Policy	Accessible	Restricted	Restricted	Restricted	Accessible
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 Programming 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 5.1: Comparison of Related Proposals

vironments, and Quantum is a module that enable the installation of third party plug-ins to control OpenStack virtual network specifically. These proposals represent the state-of-art in the matter of programing virtual networks solutions.

Analyzing the comparison presented, it can notice that only ProViNet and OpenStack plus Quantum has no restrictions to end-user access. It can also be noticed that ProViNet and CITRIX DVS are the only ones to provide the three functionalities described earlier and so remarked as High Level on Control Plane Management. Considering the features altogether and the alignment to the proposal of enabling end-users to develop network applications, ProViNet reveals to be a promising alternative.

5.2 Case Study and Performance Analysis

In order to evaluate the prototype described in Chapter 4, a case study was elaborated. In such case study a typical situation will be analyzed, in which, due to economic factors, a network administrator was requested to migrate a physical network infrastructure to the Cloud. Considering migrate all the functionalities together, *i.e.* Firewalls, Load

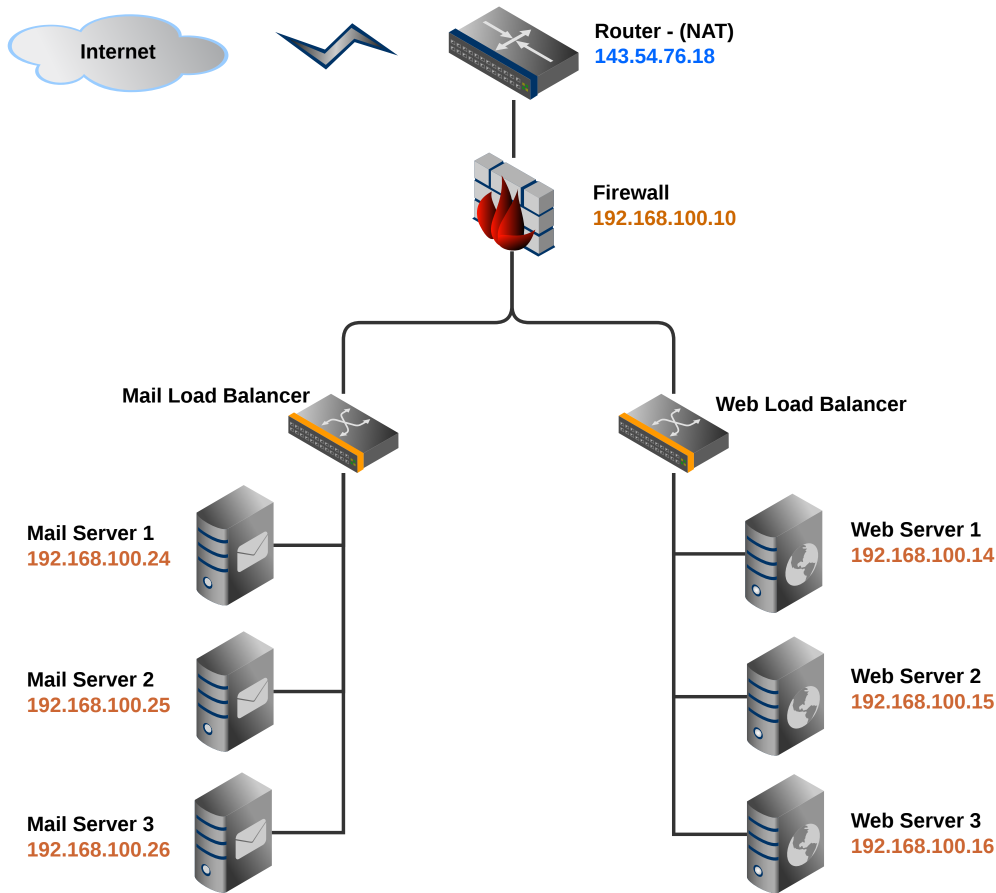

Figure 5.1: Use case scenario of physical network migration

Balancers, NAT and so on. The physical infrastructure mentioned is depicted in Figure 5.1. It is composed by two load balancer appliances, one Firewall, one border switch with NAT, three Web servers, three Mail servers and ten links. Using such case study it was evaluated the 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, the 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 the XenServer. To run ProViNet Core it was used a laptop Intel Core i7 2.8GHz and 4GB RAM.

The process of PVN Provisioning starts by translating the physical topology shown in Figure 5.1 into a virtual topology description file using the VXDL grammar. The resulting file is similar to the example already given in the Subsection 4.1.2. The appliances will become common forwarding devices in the virtual topology, once in a programmable virtual network the applications are located outside the switches. The following steps are represented in the sequence diagram of the Figure 5.2.

Considering the sequence diagram mentioned, lets represent the time taken to upload the VXDL as T_{upload}, and for creating the controller instances as $T_{ctl-request}$. Meanwhile, the time for adding the controller information in the VXDL file is represented as $T_{edit-vxdl}$,

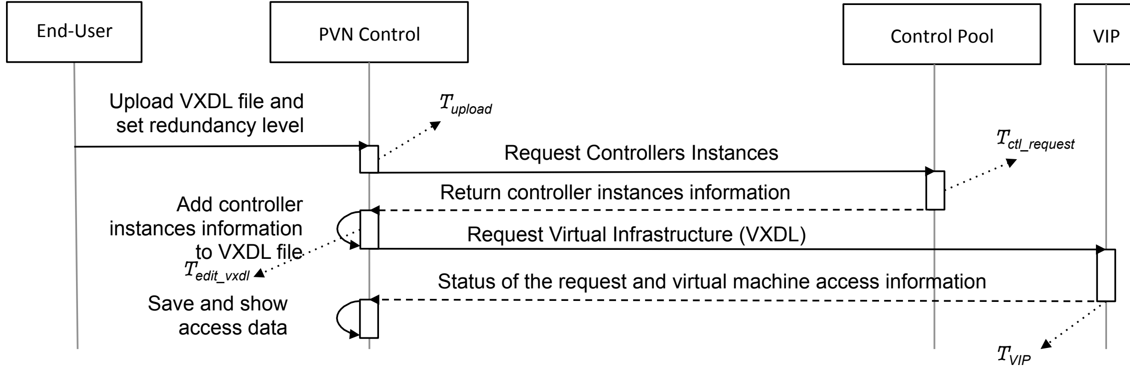

Figure 5.2: Sequence Diagram of resources provisioning.

T_{upload}	$T_{ctl-request}$	$T_{edit-vxdl}$	T_{VIP}	T_{total}
0.0293s	49.6581s	0.0388s	43.0345s	92,7608s

Table 5.2: Programmable Virtual Network Provisioning

and finally the time taken by the VIP to create and configure the virtual resources as T_{VIP}. In doing so, the total time for the whole process of virtual infrastructure provisioning, including control plane configuration, is referred to as T_{total} and is the sum of the others, so $T_{total} = T_{upload} + T_{ctl-request} + T_{edit-vxdl} + T_{VIP}$. All the values presented in the Table 5.2 regards the average time of 30 executions, in seconds, taken to conclude each procedure. It was considered that the end-user required two controllers to compose the control plane.

It is noteworthy that the time taken by the VIP to provide the virtual infrastructure is not under the control of ProViNet, but it is still considered to give an idea about the overall performance. The time taken by VIP includes the tasks of instantiating virtual machines, creating vSwitches, creating the links and connecting the virtual resources. Considering that the infrastructure required, has six virtual machines, four vSwitches and ten links, the time of 43.03 seconds is a good remark.

Request	Average Time
Add Flow	0.1480s
List Flow	0.0619s
Delete Flow	0.1246s

Table 5.3: NBAPI Dispatcher performance

The most significant time presented in the Table is the $T_{ctl-request}$, which is directly dependent on the hypervisor performance to clone and start two virtual machines. From the 49.65 seconds average, 23.05 seconds were taken just for cloning and 26.42 seconds for starting. ProViNet needs to wait the virtual machine booting, once just after that a valid IP is given to the machine and can be added in the VXDL file.

The next step in the migration use case is to develop the network applications that will control the virtual switches. Accessing ProViNet Web GUI, end-users are able to develop the *workflows* representing each network feature, such as Firewall, Load Balancer and NAT. Each feature is composed by a set of services that when executing will trigger such service toward the control plane. The component under study in this evaluation is the *NBAPI Dispatcher*. To measure the performance of this component it was run three types

of services. One for adding a flow in a switch, another to read the flow, and the last to delete the flow. After 30 executions, the average times between sending the request (HTTP GET or POST) and receiving the answer are presented in the Table 5.3.

6 CONCLUSION

In this dissertation, the main impediments and barriers that made computer networks innovation process seem frozen for decades have been discussed. Some occasional attempts to change network closed nature emerged along the 90's have been also presented, such as Active Networks and OpenSig. Recent proposals introduced the employment of Network Virtualization approach together with Programmable Network principles. As a result emerged the Programmable Virtual Network (PVN) concept, which enables distinct virtual networks to provide different services by means of network programming approaches.

Among PVN proposals, Software-Defined Networking (SDN) plays the lead role and has inspired several proposals and research projects along the last years. SDN has quickly gained momentum and its advantages and constraints have been exhaustively explored by recent researches and network industry. Despite several aspects covered in past research, managing programmable virtual network environments is still a great challenge. Issues regarding management of network applications deployment, virtual network provisioning, and access policies still require solutions. Network programming solutions without management becomes useless and discourage its adoption in real deployments.

In this Master's dissertation it was introduced a new business model named NPaaS, which is characterized by an unprecedented perspective given to end-users. Such model was designed having in mind that Service Providers must share their responsibility of creating network solutions with end-users. This way, a larger amount of people would be involved and the amount of innovation tends to grow. However, managing virtualized and programmable networks considering end-users as network application developers might become even more complex. In order to support NPaaS and provide the necessary management features it was proposed ProViNet management platform.

NPaaS business model was qualitatively compared against other models in order to highlight its importance and uniqueness. In summary, NPaaS reveal to be an innovation friendly solution due to its open nature and strong focus in end-users. In turn, the ProViNet platform was evaluated in a use case of infrastructure migration, from physical to virtual, action that reflects a tendency in recent times. Although not exhaustive, performance results have shown that the platform is feasible and flexible enough to work with different Infrastructure Providers and control plane technologies.

6.1 Main Contributions and Results Obtained

The main contributions of this research are the proposed Network Programming as a Service business model and the programmable virtual network management platform. As discussed before, NPaaS model was designed to turn network environments more open,

programmable, and accessible for end-users. In doing so, specific requirements not met by current network services could be designed, developed and deployed by end-users themselves. Such freedom induce the emergence of a new market, in which network solutions can be shared or commercialized among users. Infrastructure Providers defined in NPaaS does suffer great impact, once there are currently available from traditional network industry several models of programmable network hardwares.

ProViNet contributes with a feasible architecture engineered to support NPaaS requirements. In doing so, all components of the platform are strategically scalable in order to attend the large amount of end-users foreseen by NPaaS. Scalable Control Plane for example is able to grow as much as necessary. In addition to that, the platform contributes with an approach to grant isolation among controllers of different *slices*, grouping them in clusters. Moreover, in order to enable interoperability with a wide range of Infrastructure Providers, the platform is configurable with plug-ins.

Furthermore, there are some other important contributions that are worth mentioning. The employment of BPMN to enable the development of network applications by end-users with varying experiences in programming languages and networking is one of them. Without a doubt, such approach is a great contribution not just in NPaaS context, but also in industry, where network administrators could integrate business processes with network programming services, giving them unprecedented controlling power. Beyond that, it is also a contribution the set of new tags proposed to enable the description of control plane details in the virtual infrastructures description language VXDL.

The scenario described during ProViNet evaluation, presented in Section 5.2, has the objective of representing a real demand. Currently, Cloud Computing consumers have not much flexibility in regards to virtual network customization. Through the use of the platform it was shown that such consumers are able to migrate the whole topology while keeping its organization identical, including links, hosts, nodes, and network services. Although the response time were not initially a concern, results have shown that the time spent by the processes required to perform infrastructure migration and to execute network applications within this scenario are acceptable.

6.2 Final Remarks and Future Work

In Section 3.1 it was described a Network Programming as a Service architecture obtained from the integration of SDN and a network virtualization business model. In this integration, SDN Infrastructure Layer becomes part of Infrastructure Provider, and Control and Application Layers goes within Service Provider boundaries. Another possible organization would be to transfer Control Layer to Infrastructure Providers. In future work the consequences of this change will be analyzed. But prematurely, it is possible to infer that new high availability strategies could be provided by InPs, once they would have total control over both infrastructure and control layers. It may be an advantage because the most critical part of SDN architecture is the connection between such layer. Keeping just one responsible to it may induce less failures and more effective recovering process.

Developing network applications using BPMN met NPaaS target public requirements. However, many users would prefer developing network applications using regular programming languages, such as Java, C, or C++, due to its incontestable programming flexibility. A strategy to enable the use of such languages and still keep all the platform advantages was proposed in a previous deliverable of this research, found in the Appendix B. In such strategy, instead of accessing the current graphical development interface, end-users

had access to virtual machine consoles. This way, control plane services are provided as software APIs and can be employed in network applications. Future investigations could explore variations of these strategies, for instance, providing an integration of them. Then experienced developers would use regular programming language strategy and network administrators, managers, and business executives BPMN strategy.

In relation to the current version of ProViNet platform, there are some technical details that could be improved. Such as the creation of an interface in which end-users would be able to graphically define virtual network topologies, leaving the error prone strategy of manually editing an XML file.

REFERENCES

ACHEMLAL, M. **Virtualisation approach**. Technical Report D- 3.1.1, 4WARD - Architecture and Design for the Future Internet, 2010.

ANGIN, O.; CAMPBELL, A.; KOUNAVIS, M.; LIAO, R.-F. The mobiware toolkit: programmable support for adaptive mobile networking. **IEEE Personal Communications**, [S.l.], v.5, n.4, p.32–43, Dec. 1998.

BANIKAZEMI, M.; OLSHEFSKI, D.; SHAIKH, A.; TRACEY, J.; WANG, G. Meridian: an sdn platform for cloud network services. **IEEE Communications Magazine**, New York, NY, USA, v.51, n.2, p.120–127, Nov. 2013.

BAVIER, A.; FEAMSTER, N.; HUANG, M.; PETERSON, L.; REXFORD, J. In VINI veritas: realistic and controlled network experimentation. **SIGCOMM Computer Communication Review**, New York, NY, USA, v.36, n.4, p.3–14, Aug. 2006.

BISWAS, J.; LAZAR, A.; HUARD, J.-F.; LIM, K.; MAHJOUB, S.; PAU, L.-f.; SUZUKI, M.; TORSTENSSON, S.; WANG, W.; WEINSTEIN, S. The IEEE P1520 standards initiative for programmable network interfaces. **IEEE Communications Magazine**, [S.l.], v.36, n.10, p.64–70, 1998.

CAMPBELL, A. T.; DE MEER, H. G.; KOUNAVIS, M. E.; MIKI, K.; VICENTE, J. B.; VILLELA, D. A Survey of Programmable Networks. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.29, n.2, p.7–23, Apr. 1999.

CAMPBELL, A. T.; KATZELA, I.; MIKI, K.; VICENTE, J. Open signaling for ATM, internet and mobile networks. **SIGCOMM Computer Communication Review**, New York, NY, USA, v.29, n.1, p.97–108, Jan. 1999.

CHERTOV, R.; HAVEY, D.; ALMEROTH, K. MSET: a mobility satellite emulation testbed. In: CONFERENCE ON INFORMATION COMMUNICATIONS, 29., 2010, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2010. p.2276–2284. (INFOCOM'10).

CHOWDHURY, N.; BOUTABA, R. Network virtualization: state of the art and research challenges. **IEEE Communications Magazine**, [S.l.], v.47, n.7, p.20 –26, july 2009.

CHOWDHURY, N. M. K.; BOUTABA, R. A Survey of Network Virtualization. **Technical Report: CS-2008-25**, Waterloo, Ontario, Canada, v.25, p.1–29, Oct. 2008.

CHOWDHURY, N. M. K.; BOUTABA, R. A survey of network virtualization. **Computer Networks**, New York, NY, USA, v.54, n.5, p.862–876, Apr. 2010.

CISCO. **Application-Oriented Networking**. Available at: http://www.cisco.com/go/aon. Visited at: mar 2013.

DORIA, A.; HADI SALIM, J. **Forwarding and Control Element Separation (ForCES) (RFC 5810)**. [S.l.]: RFC Editor, 2010.

ELMELEEGY, K.; COX, A. L.; NG, T. S. E. Etherfuse: an ethernet watchdog. In: CONFERENCE ON APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.253–264. (SIGCOMM '07).

FLORISSI, D. The NetScript Approach to Programmable Networks. **OPENSIG'98 Workshop on Open Signaling for ATM, Internet and Mobile Networks**, [S.l.], Oct 1998.

FOUNDATION, O. N. **The Google SDN WAN**. An Open Networking Foundation Interview.

GALIS, A.; DENAZIS, S.; BROU, C.; KLEIN, C. **Programmable Networks for IP Service Deployment**. Norwood, MA, USA: Artech House, Inc., 2004.

GUDE, N.; KOPONEN, T.; PETTIT, J.; PFAFF, B.; CASADO, M.; MCKEOWN, N.; SHENKER, S. NOX: towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, New York, NY, USA, v.38, n.3, p.105–110, July 2008.

HAUSHEER, D.; PAREKH, A.; WALRAND, J.; SCHWARTZ, G. Towards a compelling new Internet platform. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT (IM), 2011, Dublin, Ireland. **Proceedings...** IEEE, 2011. p.1224–1227.

IEEE Second Conference on Open Architectures and Network Programming Proceedings. **OPENARCH**. 1999.

KHOSRAVI, H.; ANDERSON, T. **Requirements for Separation of IP Control and Forwarding (RFC 3654)**. [S.l.]: RFC Editor, 2003.

KOSLOVSKI, G. P.; PRIMET, P. V.-B.; CHARãO, A. S. VXDL: virtual resources and interconnection networks description language. **Springer**, [S.l.], v.2, p.138–154, 2008.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: ACM SIGCOMM WORKSHOP ON HOT TOPICS IN NETWORKS, 9., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.19:1–19:6.

LAZAR, A.; LIM, K.-S.; MARCONCINI, F. Realizing a foundation for programmability of ATM networks with the binding architecture. **IEEE Journal on Selected Areas in Communications**, [S.l.], v.14, n.7, p.1214–1227, 1996.

LEVIN, D.; WUNDSAM, A.; HELLER, B.; HANDIGOL, N.; FELDMANN, A. Logically centralized?: state distribution trade-offs in software defined networks. In: HOT TOPICS IN SOFTWARE DEFINED NETWORKS, 2012, New York, NY, USA. **Proceedings...** ACM, 2012. p.1–6. (HotSDN '12).

LIN, P.; BI, J.; HU, H.; FENG, T.; JIANG, X. A quick survey on selected approaches for preparing programmable networks. In: ASIAN INTERNET ENGINEERING CONFERENCE, 7., 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p.160–163. (AINTEC '11).

MCKEOWN, N. Keynote talk: software-defined networking. In: IEEE INFOCOM09, 2009, Rio de Janeiro, RJ, Brazil. **Proceedings...** IEEE, 2009.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. OpenFlow: enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, New York, NY, USA, v.38, p.69–74, March 2008.

MERWE, J. Van der; ROONEY, S.; LESLIE, I.; CROSBY, S. The Tempest-a practical framework for network programmability. **IEEE Network**, [S.l.], v.12, n.3, p.20–28, 1998.

MORRIS, R.; KOHLER, E.; JANNOTTI, J.; KAASHOEK, M. F. The Click modular router. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1999, New York, NY, USA. **Proceedings...** ACM, 1999. p.217–231. (SOSP '99).

PSOUNIS, K. Active Networks: applications, security, safety, and architectures. **IEEE Communications Surveys Tutorials**, [S.l.], v.2, n.1, p.2–16, Set 1999.

RIO, M.; PEZZI, N.; ZANOLIN, L.; MEER, H. D.; EMMERICH, W.; MASCOLO, C. Promile: a management architecture for programmable modular routers. In: OPENSIG, 2001, Imperial College, London, UK. **Proceedings...** ACM, 2001.

ROSS, F. An architecture for programmable network elements. In: IEEE GLOBAL TELECOMMUNICATIONS CONFERENCE AND EXHIBITION, 1989, Dallas, Texas, USA. **Proceedings...** IEEE, 1989. p.122–126. (GLOBECOM '89).

SCHUSTER, H. Pros and Cons of Distributed Workflow Execution Algorithms. In: **Data Management in a Connected World**. Berlin, Heidelberg, Germany: Springer, 2005. p.215–234. (Lecture Notes in Computer Science, v.3551).

SCHWARTZ, B.; JACKSON, A.; STRAYER, W.; ZHOU, W.; ROCKWELL, R.; PARTRIDGE, C. Smart Packets for Active Networks. In: OPEN ARCHITECTURES AND NETWORK PROGRAMMING PROCEEDINGS, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.90–97. (OPENARCH '99).

SUGIYAMA, H. Programmable Network Systems: through the junos sdk and junos space sdk. In: WORLD TELECOMMUNICATIONS CONGRESS (WTC), 2012, Miyazaki, Japan. **Proceedings...** IEICE-CS, 2012. p.1–6.

SYSTEMS, C. **XenServer Administrator's Guide 5.5.0**. 2010.

WICKBOLDT, J. A.; GRANVILLE, L. Z.; SCHNEIDER, F.; DUDKOWSKI, D.; BRUNNER, M. A New Approach to the Design of Flexible Cloud Management Platforms. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM), 8., 2012, Las Vegas, USA. **Proceedings...** ACM, 2012. p.155–158.

YEGANEH, S.; TOOTOONCHIAN, A.; GANJALI, Y. On scalability of software-defined networking. **IEEE Communications Magazine**, [S.l.], v.51, n.2, p.136–141, 2013.

ANEXO A PUBLISHED PAPER – COMPSAC 2013

In this paper it was published advances in the approach of programming network applications, which mainly introduced the strategy of using BPMN to define network applications. To support such novel strategy, it was discussed about an executor of *workflows* based in execution queues. Among the contributions of this paper is the use of VXDL files, and the definition of new tags to VXDL. Such language enables virtual infrastructure representations, and with the novel tags it becomes able to represent also control plane information of programmable virtual networks.

- **Title:**
  ProViNet - A Programmable Virtual Network Management Platform
- **Conference:**
  Conference of 37th COMPSAC/IEEE Annual International Computer Software & Applications (COMPSAC 2011)
- **URL:**
  `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6649845&queryText%3DProViNet`
- **Date:**
  July 22-26, 2013
- **Held at:**
  Kyoto Terrsa, Kyoto, Japan
- **Digital Object Identifier (DOI):**
  `http://dx.doi.org/10.1109/COMPSAC.2013.58`

ProViNet - An Open Platform for Programmable Virtual Network Management

Abstract—**The disheartening and thorny path followed while deploying new solutions in the core of current computer networks, culminates in a low rate 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 composition, in which applications are built simply by queuing control plane services. Such approach encourages End Users to develop and share novel network solutions, inducing higher rates of 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 composition, in which applications are built simply by queuing control plane services. Such approach encourages End Users to develop and share novel network solutions, inducing higher rates of 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 our proposed approach.

The rest of the paper is organized as follows. In Section II, we outline the main papers that discuss about the management of PVN. 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 platform proposed by Stanford University and assists researchers in creating slices using resources from numerous 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 [12] 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 [13].

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.* [14], 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 management platform. 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*.

Nowadays, Cloud providers do not allow *End-Users* to define a 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.

We argue that higher innovation rates comes with 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. Our target public has no special permissions or privileges other than the inherent to its own interests. We call them End Users. A network manager, a software developer, or network engineering could deploy ProViNet

to manage virtual network solutions of slices in a virtual infrastructure. Its important to note that ProViNet End Users may not be the same that will consume the network solution developed, called Service Consumer in this paper.

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 the End User. Nothing prevents other types of users from adopting ProViNet, but there are already a numerous solutions focused on attending their specific demands. Differently, ProViNet has as main requirement the easy of use, once the objective is to encourage a large number of users to create their own network solution. Our challenge is to find a good trade off among simplicity and granularity, or programmability level, so the End User can easily create and deploy powerful network solutions.

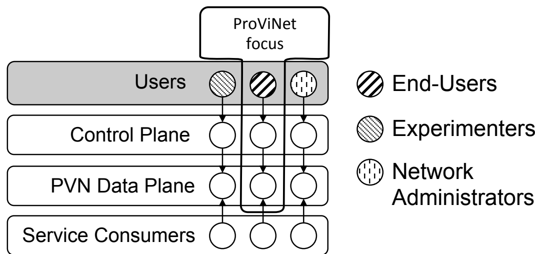

Figure 1. ProViNet provides End Users access to the Control Plane

The four layers depicted in Figure 1 represent the 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 the network, because the control plane becomes closer to the network edge. Exploring this characteristic ProViNet is able to manage network applications of End User and their relationship with PVN slices. It is worth noticing that the End User is the one who will create the applications, but the consumer of the application functionality may be someone else. In this paper they are referred as Service Consumers. Occasionally they both can be the same person, but in case they are not, it can represent the starting of a novel network application market.

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

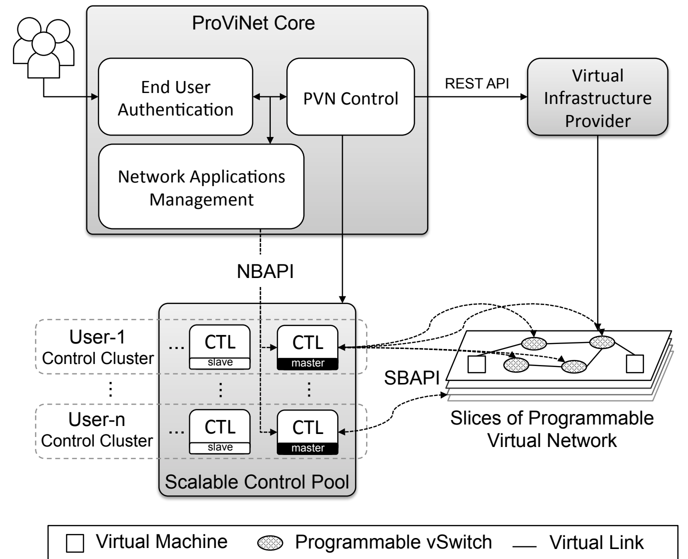

Figure 2. Conceptual Architecture

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, 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) [15]. 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 the *End User* have 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) all the controller information is added automatically by ProViNet.

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 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 informations related to accessing resources in the *Slice*. In order to diminish complexity, such information is graphically represented and allow the End User to interact via browser with the virtual machine instances of the Slice.

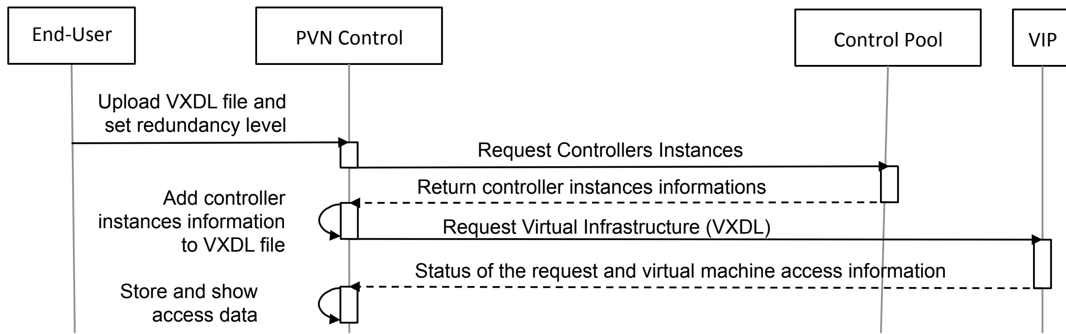

Figure 3. Sequence Diagram of Slices Provisioning Procedure

D. Network Applications Management

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 services in the queue are consumed by the *NBAPI Dispatcher*, an element of *Network Application Managment* module. The *Dispatcher* role is to send HTTP requests to the master controller of the users's *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 attend each service request exchanging SBAPI calls with the vSwitches in the Slice, or consulting its internal system.

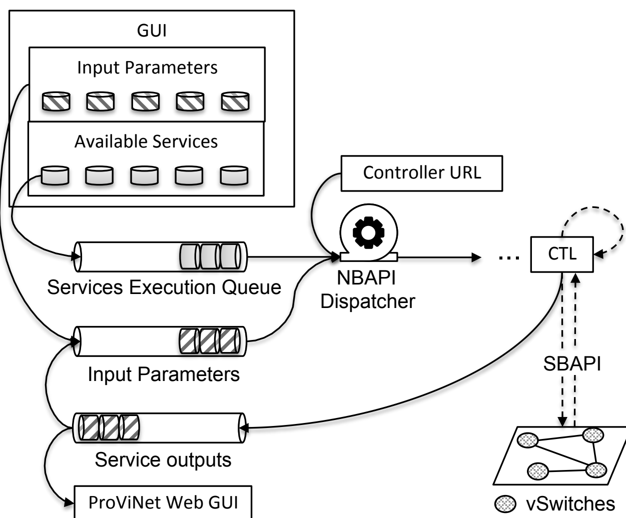

Figure 4. Queuing approach for a user-friendly Network Programming

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 the programming procedure is done through the Web GUI. The example is present using XML format because it may represent Web GUI states.

The objective of the Firewall example is to make specific switch to drop all the packets having the port 22 as destination. To that end, the services *getTopology* and *setFlow* must me queued. The first will retrieve the switches ids, then the first switch is chosen as input parameter to the *setFlow* service.

```
<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 may be 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 the framework Django 1.4.3, Python 2.7.3 and PostgreSQL 9.1.6 database.

The End User Authentication handles the login and registration features of ProViNet. The PVN Control component has all the routines and APIs necessary to communicate with the Virtual Infrastructure Provider and with the Scalable Control Pool (SCP). The SCP was deployed with XenServer [16], then the communication with this server were implemented with the XenAPI, provided by CITRIX. Using this API, remote systems is able to control a Pool of XenServers.

For instance, when the user request the virtual infrastructure and sets the redundancy, the redundancy is used to iterate calls of clone function (session.xenapi.VM.clone(vm_base_name, vm_cloned_name)). When XenServer receives the clone request, which has the name of a base virtual machine as parameter, it creates a copy of the virtual machine with matching the name given. In our implementation, the virtual machine used as base for the copy were already prepared with an installation of FloodLight, which was configured to start automatically after the OS boot. When the VM at the XenServer is running, PVN Control does another call of reading IP address. Which will return the IP address of the VM just cloned. The IP address is used to edit the VXDL file, which is accomplished using the Python library xml.etree.ElementTree.

The second communication of the PVN Control component is performed with the Virtual Infrastructure Provider. After the user has uploaded the VXDL, the VM cloning procedure were done, as well as editing, PVN Control reads the edited VXDL and store in a url encoded string. Then a HTTP POST request is sent to the VIP, with the encoded VXDL string and some authentication data. After the properly instantiation and configuration of the virtual infrastructure, the VIP sends back a HTTP answer, with the result of the request, whether success of failure. In case of failure, the cause of the failure is specified.

Finally, the Network Application Management (NAM) component was basically implemented with python routines, which, in its turn integrates with the templates of Django framework. The Django view that handles the GUI input events, interacts with the NAM python class, that has permissions to manage network application queues. Such queues are simple python arrays of service objects, which represents generically any service, such as its inputs, outputs and URI.

We choose the FloodLight as the controller implementation, because it provides an API allowing restfull consumption of services available to 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 some use cases. The videos are available at the address <to appear>. Our prototype source code and documentation are found also in the same web site.

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 programming simpler and manageable. In what follows we describe such features and then, in the Table V-A we present the remarks of each solution. The analysis of the proposals considered the on-line available material, such as 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 *No Restrictions*, *i.e.*, the user is able to freely require virtual resources, similar to the approach performed in Cloud Computing deployments. Note that paying or not for the resources is not considered a restriction.

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 the SDN architecture the control plane runs out of the 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, mot switches support High Availability (HA) by accepting the configuration of more than one controller. Then in case of ones failure the other can assume the control master role. Most of the proposals leaves the control plane management to the users. However, if such control is done in 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 was 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*.

Among the proposals presented in the Table V-A, 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 no restrictions to End Users access. 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

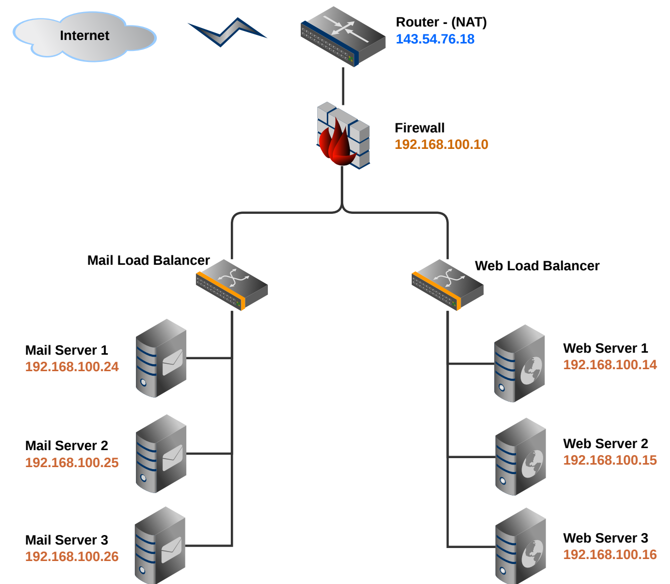

Figure 5. Use case scenario of physical network migration

In order to evaluate the prototype described in the section IV, we elaborate a case study. In this case study we will analyze a typical situation, in which, due to economic factors,

Feature	**ProViNet**	**OFELIA Control Framework**	**ProtoGENI**	**CITRIX DVS**	**OpenStack+Quantum**
End-User Access Policy	No Restrictions	Restricted Access	Restricted Access	Restricted Access	No Restriction
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

the network administrator was requested to migrate a physical network set up to the Cloud. Considering migrate all the functionalities together, *i.e.* Firewalls, Load Balancers, NAT and so on. The physical infrastructure mentioned is depicted in the 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 the 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, the 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 the XenServer. To run the ProViNet Core we used a laptop Intel Core i7 2.8GHz and 4GB RAM.

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

Considering the sequence diagram mentioned, lets represent the time taken for upload the VXDL as T_{upload}, and for creating the controller instances as $T_{ctl-request}$. Meanwhile, the time for adding the controller information in the VXDL file is represented as $T_{edit-vxdl}$, and finally the time taken by the VIP to instantiate the virtual resources as T_{VIP}. In doing so, the total time for the whole process of provisioning the virtual infrastructure, including the configuration of the control plane, is referred to as T_{total} and is the sum of the others, so $T_{total} = T_{upload} + T_{ctl-request} + T_{edit-vxdl} + T_{VIP}$. All the values presented in the Table V-B regards the average time of 30 executions, in seconds, taken to conclude each procedure. We considered that the End User required a redundancy of two controllers.

It is noteworthy that the time taken by the PIV to provide the virtual infrastructure is not under the control of ProViNet, but we still consider it to give an idea about the overall performance or our solution. The time taken by VIP includes the tasks of instantiating virtual machines, creating vSwitches, creating the links and connecting the virtual resources. Considering that the infrastructure required, has six virtual machines, four vSwitches and ten links, the time of 43.034518532 seconds is a good remark.

The most significant time presented in the table is the $T_{ctl-request}$, which is directly dependent of the hypervisor performance to clone and start two virtual machines. From the 49.658138036 seconds average, 23.055444002 seconds were taken just for cloning and 26.425736904 seconds for starting. ProViNet needs to wait the virtual machine booting, once just after that a valid IP is given to the machine and can be added in the VXDL file.

The next step in the migration use case is to develop the 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 the Section III-D. Once the queues of Firewall, Load Balancer and NAT is ready he 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 the flow, and the last to delete the flow. After 30 executions, the average times between sending the request (HTTP GET or POST) and receiving the answer are presented in the Table V-B.

T_{upload}	$T_{ctl-request}$	$T_{edit-vxdl}$	T_{VIP}	T_{total}
0.029394308	49.658138036	0.038839101	43.034518532	92,760889977

Table II
PROGRAMMABLE VIRTUAL NETWORK PROVISIONING

Request	Average Time
Add Flow	0.14807082812
List Flow	0.0619584878286
Delete Flow	0.124699409803

Table III
NBAPI DISPATCHER PERFORMANCE

VI. CONCLUSIONS AND FUTURE WORK

With the evolution of virtual networks, a diversity of computing environments has emerged. The network solutions developed in the past, and implemented in accordance with the wishes of the manufacturers of networking equipment, may no longer be sufficient. To overcome such limitation novel solution may be proposed. However the process of developing and deploying novel solutions in virtual network environments is still immature.

In order to speed up the process of proposing and deploying novel solutions in virtual network, the Software Defined Networking is a good ally. SDN propose 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. However, 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.

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. As shown in this work, we propose an architecture with the necessary components for managing virtual network infrastructures, network applications and controlling End User access. We also propose in this work, a user friendly approach for programing virtual networks, which employs concepts of Web service composition.

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 our proposed approach. We also presented an 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 programing approach with techniques of services composition. This would allow different kinds of services aggregation, not just the sequential organization provided by the queue approach.

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

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

[14] 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.

[15] G. P. Koslovski, P. V.-B. Primet, and A. S. CharÃ£o, "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.

[16] "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

ANEXO B PUBLISHED PAPER – SBRC 2013

In this appendix the paper entitled "*ProViNet - Uma Plataforma para Gerenciamento de Redes Virtuais Programáveis*" is presented (in Portuguese). This was the first deliverable of this research and focused in programmable virtual network infrastructure provisioning. The strategy of application plane proposed in this paper considered the use of virtualized resource pools. After resource provisioning, ProViNet configure a application plane virtual machine. So, accessing such machine from ProViNet interface End-users were able to develop and deploy its own network application. There was introduced in this paper an approach or control plane high-availability, which used a DNS server to enable end-users application to dispatch Web Service calls using as destination an general address of control plane, such as http://controlplane.local/GetTopologyInformation.

- **Title:**
  ProViNet - Uma Plataforma para Gerenciamento de Redes Virtuais Programáveis
- **Conference:**
  XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2013)
- **URL:**
  `http://sbrc2013.unb.br/`
- **Date:**
  May 6-10, 2013
- **Held at:**
  Royal Tulip Brasilia Alvorada, Brasilia, Distrito Federal, Brazil
- **Digital Library of SBC:**
  `http://www.lbd.dcc.ufmg.br/colecoes/sbrc/2013/xxxx.pdf`

ProViNet: Uma Plataforma para Gerenciamento de Redes Virtuais Programáveis

Wanderson Paim de Jesus[1], Ricardo Luis dos Santos[1], Oscar Maurício Caicedo Rendón[1] Lisandro Zambenedetti Granville[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{wpjesus, rlsantos, omrendom, granville}@inf.ufrgs.br

Abstract. *With the evolvement of virtualization and network programming techniques, high-level programs can be used to define the behavior of network traffic while keeping isolation. However, to ensure a harmonious relationship between users, network programs and virtual networks, considerable management efforts are needed. In this paper we propose the ProViNet platform, a solution for managing the deployment of network programs in programmable virtual networks. ProViNet contributes with an architecture that allows sharing of the control plane of these networks in a scalable way. During the development of this work we identified the need for a standard representation of programmable virtual infrastructures, so it is also proposed a programmable virtual networks description language. In order to verify the feasibility of the proposed platform, we implemented a prototype, which is analyzed and evaluated in this work.*

Resumo. *Ao passo que evoluem as técnicas de virtualização e programação de redes, programas de alto nível podem ser utilizados para definir o comportamento de tráfegos de rede isoladamente. Entretanto, garantir um relacionamento harmônico entre usuários, programas de rede e redes virtuais exige grandes esforços de gerenciamento. Neste trabalho propomos a plataforma ProViNet, uma solução para o gerenciamento da implantação de programas de rede em redes virtuais programáveis. ProViNet contribui com uma arquitetura que permite o compartilhamento do plano de controle dessas redes de forma escalável. Durante o desenvolvimento do trabalho foi identificada a necessidade de uma representação padrão da infraestrutura virtual programável, para tanto propõe-se também, uma linguagem de definição de redes virtuais programáveis. A fim de verificar a viabilidade da plataforma proposta, foi implementado um protótipo, o qual é analisado e avaliado neste trabalho.*

1. Introdução

Historicamente, o núcleo das redes de computadores, quando comparado com os servidores, *desktops* e dispositivos móveis da borda das redes, é um ambiente hostil à inovação. No contexto específico da Internet, esse fato é geralmente referenciado como ossificação [Hausheer *et al.* 2011]. Para exemplificar, soluções propostas a mais de dez anos, como IPv6 e IPSec, ainda não estão amplamente em uso. São apontadas como possíveis causas: i) A necessidade de modificações globais, ocasionalmente exigindo a substituição de equipamentos; ii) A lentidão no processo de padronização que trata da interoperabilidade

com serviços legados; iii) e a abordagem adotada pelas fabricantes de equipamentos, de implementar e implantar soluções baseadas em seu retorno financeiro.

Na intersecção entre o conceito de Virtualização de Redes [Chowdhury e Boutaba 2010] e o de Programabilidade de Redes [Kanaumi *et al.* 2010], emergem as Redes Virtuais Programáveis (RVP), as quais promissoramente prometem reverter o cenário de lentidão testemunhado nas redes de computadores. Uma das abordagens adotadas pelas RVP segue o formato das Redes Definidas por Software (SDN - *Software-Defined Networks*) [Lantz *et al.* 2010], que define o desacoplamento dos planos de controle e de dados. Algumas implementações do plano de controle se baseiam na Arquitetura Orientada a Serviços (SOA - Service Oriented Architecture) para prover a comunicação com aplicativos de rede. Dessa forma, utilizando uma interface padronizada de definição de serviços, os aplicativos de rede podem ser programados em linguagens distintas, se tornam menos dependentes da tecnologia utilizada no plano de controle.

Por um lado a arquitetura SDN, com o desacoplamento entre aplicativos de rede, plano de controle e infraestrutura programável, se mostra flexível e escalável. Por outro lado, induz uma grande complexidade no gerenciamento. Modelos de gerenciamento utilizados nas redes comuns não são adequados às redes programáveis, uma vez que não tratam da implantação dinâmica de novos serviços. Além disso, dependendo das políticas de acesso às infraestruturas programáveis, um grande número de usuários poderão propor e implantar seus próprios aplicativos de rede. Nesse cenário, harmonizar os aplicativos, usuários e redes virtuais, mantendo a confiabilidade e escalabilidade dos serviços implantados é um problema em aberto.

As propostas para o gerenciamento de RVP variam de acordo com o ambiente de implantação e com os requisitos dos usuários. Nos ambientes de *testbeds*, as propostas focam em prover aos experimentadores e cientistas soluções de controle de *Slices*, enquanto o gerenciamento da programabilidade que os usuários possuem sobre os *Slices* ainda é incipiente. São exemplos desse tipo de proposta o ProtoGENI [pro 2012] e o OFÉLIA *Control Framework* [Kopsel 2011]. Nos ambientes de *Cloud*, mais direcionados ao mercado, implementam-se soluções para gerenciar os serviços que os provedores de *Cloud* oferecerão aos seus clientes. Ou seja, o foco maior está em prover controle aos gerentes e administradores da *Cloud*, deixando o usuário final sem chance de propor novos aplicativos de rede. As propostas da CITRIX, XenServer *Distributed vSwitch Controller* e da CISCO, OnePK, são exemplos.

O problema de pesquisa deste trabalho está em como prover acesso de múltiplos usuários finais a uma infraestrutura de RVP, agregando facilidades no gerenciamento e implantação de aplicativos de forma a encorajar o desenvolvimento de novas soluções de rede. Propõe-se para isso a plataforma de gerenciamento ProViNet (***Pro**grammable **Vi**rtual **Net**work Managemet Platform*). ProViNet contribui para o estado-da-arte em quatro pontos: *(i)* na elaboração de uma arquitetura para gerenciamento de RVP que provê escalabilidade e alta disponibilidade de serviços; *(ii)* em uma abordagem para implantação dinâmica de novos serviços no plano de controle; *(iii)* na proposta de uma linguagem de descrição de rede virtual programável, chamada *Programmable Virtual Network Description Language* (PVNDL), adaptada da VXDL [Koslovski *et al.* 2008]; *(iv)* e por fim, no desenvolvimento, como parte da plataforma, de um sistema com interface de acesso Web

que facilita a compreensão e interação dos usuários finais com ambientes de RVP.

O restante do artigo está organizado conforme segue. Na Seção 2, são descritos os principais trabalhos que envolvem o gerenciamento de RVP. Na Seção 3 é apresentada a plataforma ProViNet, discutindo a arquitetura conceitual e conceitos empregados. Em seguida, na Seção 4 é detalhado o protótipo utilizado como base para a avaliação e análise apresentada na Seção 5. Por fim a Seção 6 conclui o artigo com as considerações finais e perspectivas para trabalhos futuros.

2. Trabalhos Relacionados e Contextualização

Os conceitos de Virtualização de Redes [Chowdhury e Boutaba 2009] [Chowdhury e Boutaba 2010] e a Programabilidade de Redes [Campbell *et al.* 1999] [Lin *et al.* 2011] são bem discutidos na literatura. A junção desses conceitos formam as Redes Virtuais Programáveis (RVP), as quais são mais comumente aplicadas em dois ambientes, nos projetos de plataformas de testes, também conhecidos como *testbeds* e em Nuvens privadas (*Private Clouds*). Os trabalhos relacionados apresentados neste capítulo são organizados conforme esses dois ambientes, buscando evidenciar em cada um deles o nível de abstração da infraestrutura virtual, a abordagem utilizada e a natureza da licença.

Dentre as propostas no contexto dos *testbeds*, as quais focam em auxiliar os pesquisadores em seus experimentos, destaca-se o *framework* de controle OFELIA (OCF) [Kopsel 2011], o qual é uma derivação da plataforma Expedient proposta pela Universidade de Stanford. Esse *framework* auxilia os pesquisadores na criação de *Slices* utilizando recursos de várias federações. Além disso lhes oferece também a capacidade de associar esses *Slices* a controladores previamente configurados. Apesar dessa funcionalidade de associação através da interface gráfica, o OCF não provê o gerenciamento da implantação de aplicativos de rede, considerada uma das maiores preocupações da virtualização de redes [Chowdhury e Boutaba 2010].

Outra proposta, ainda no contexto dos *testbeds*, foi criada no projeto GENI [GENI 2011] e é chamada ProtoGENI [pro 2012]. Tal proposta também implementa uma interface de acesso via Web que assiste aos pesquisadores na criação de *Slices* com recursos oriundos de diversas federações. Ao interagir com a interface do ProtoGENI, pesquisadores podem instanciar nós virtuais e conecta-los dinamicamente. Assim como a maioria das propostas desenvolvidas nesse mesmo contexto, o foco está no provimento do *Slice* e não na programabilidade do mesmo. Em geral essas soluções são livres de licença, entretanto existe um conjunto burocrático de regras para utilização e acesso aos recursos geridos pelas ferramentas citadas.

As soluções no contexto de *Cloud Computing* se diferem das propostas de *testbeds*, principalmente pelo foco comercial. Os ambientes de *Cloud* em geral demandam uma grade quantidade de recursos de rede, pois comercializam serviços com alta taxa de disponibilidade e qualidade de serviço. Portanto, a criação de serviços de rede customizados para atender a demandas especiais é visto como um grande atrativo para os provedores de *Cloud*. Para atender esses provedores, surgiram soluções que aumentam o poder de customização dos serviços providos nas redes virtuais de seus *datacenters*. Algumas dessas soluções são comercializadas, tais como Nexus 1000v e OnePK da CISCO e DVSC (*Distributed Virtual Switch Controller*) da CITRIX. Outras são de código aberto, como os *plugins open-vSwitch*, Ryu Plugin e o *restproxy*, para a plataforma OpenStack

[OpenStack 2011].

Tanto em *Cloud* quanto em *testbeds*, é crescente o número de propostas que empregam os conceitos da arquitetura SDN em suas soluções de programabilidade em redes virtuais. Rubio *et al.* [Rubio-Loyola *et al.* 2011], por exemplo, propôs um plano de orquestração, o qual é complementar aos planos originalmente definidos na arquitetura SDN (controle e dados). O propósito desse novo plano é controlar dinamicamente o comportamento das redes virtuais em resposta às constantes variações, gerando assim um sistema de gerenciamento autonômico. Embora esse sistema tenha vantagens, como resposta rápida às falhas geradas por variações no comportamento da rede, requer soluções padronizadas e bem testadas. Em consequência, o usuário final continua distante da criação e implantação de aplicativos na rede virtual programável. Tal fato, contribui para baixa taxa de inovação nas redes, pois mantem a natureza restritiva e a pequena quantidade de pessoas aptas a desenvolver e implantar novas soluções.

Em resumo, apesar de existirem propostas recentes envolvendo redes virtuais programáveis, nenhuma das pesquisadas promove o gerenciamento da programabilidade em uma infraestrutura de RVP, considerando o acesso de múltiplos usuários finais. Além disso, maior parte das propostas analisadas se limitam ao provisionamento da rede virtual (controle de máquinas virtuais e configuração da rede virtual), não oferecendo funcionalidades para o gerenciamento dos aplicativos de rede que podem ser instalados dinamicamente nas redes virtuais programáveis.

3. ProViNet

Uma das abordagens para as Redes Virtuais Programáveis é a utilização da arquitetura de Redes Definidas por Software. Tal arquitetura define que os planos de controle e de dados sejam desacoplados. Sendo assim, eles necessitam de um protocolo para comunicação. Recentemente tem se empregado o termo *Southbound API* (SBAPI) para se referir aos protocolos que provejam essa comunicação entre o plano de controle e de dados. Conforme ilustrado na Figura 1 a SBAPI **(2)** é utilizada para comunicação entre o *Pool* de Controle e os *Slices*, que serão descritos mais diante.

Ao passo que surgiram diversas soluções para a camada de controle, e cada implementação adota um padrão de linguagem, os aplicativos de rede criados seguiam tal heterogeneidade, ficando assim dependentes das tecnologias dos controladores e isoladas entre si. Atualmente a tendência é que as diferentes implementações do plano de controle ofereçam uma interface padrão de comunicação com aplicativos de rede externos. É comum se referir a esse tipo de interface como *Northbound API* (NBAPI) **(1)**. Em geral elas independem de linguagem, baseando-se na arquitetura *Representational State Transfer* (REST), por exemplo. Desse modo, novos aplicativos que venham a interagir com a camada de controle necessitam apenas das especificações de serviços providas por cada implementação de controlador.

Conforme ilustrado na Figura 1, a plataforma ProViNet auxilia os usuários finais no gerenciamento e implantação de aplicativos de rede em *Slices* de Redes Virtuais Programáveis. Para isso, se apoia no conceito de separação de planos, sejam eles: o plano de dados, representado pelos elementos que formam a rede virtual nos *Slices* e utilizam a SBAPI para se comunicarem com o plano de controle; o plano de controle, formado pelos controladores, que são agrupados no *Pool* de Controle para prover alta disponibili-

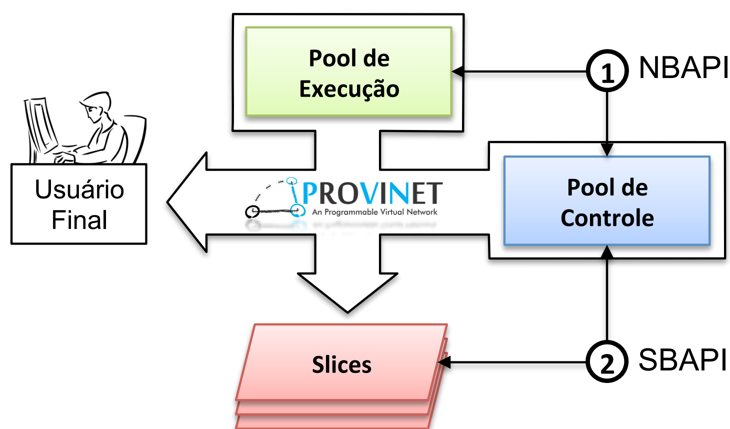

Figura 1. Módulos e relacionamentos

dade, conforme será apresentado na Subseção 3.1; e pelo plano de aplicativos, presente no *Pool* de Execução, o qual organiza, armazena e executa os aplicativos de rede que se aproveitam do conceito de NBAPI para comunicação com o plano de controle.

A plataforma ProViNet se aplica a qualquer ambiente que utilize infraestrutura compatível com o conceito de RVP. Em geral a aplicação se dá em dois níveis, um no nível de rede virtual, no qual o plano de dados seriam representado pelos *vSwitches*, e outro no nível físico, utilizando os *switches* físicos compatíveis como plano de dados. Ou até mesmo em um modelo híbrido, com controle em ambos os níveis.

3.1. Arquitetura Conceitual

A arquitetura apresentada na Figura 2 ilustra os componentes da plataforma ProViNet assim como suas relações em alto nível. O usuário interage com a plataforma através de uma interface Web que expõe graficamente as funcionalidades providas por seus módulos. Conforme ilustrado, essas funcionalidades se subdividem em quatro interações. Na interação **(a)**, o módulo Controle de Usuários atende a requisições de autenticação e cadastro. O processo de gerenciamento de implantação e execução de aplicativos, executado na interação **(b)**, é tratado pelo módulo Controle de Aplicativos. As solicitações de infraestrutura virtual são enviadas na interação **(c)** e tratadas pelo módulo de Controle de *Slices* e Referências. Por fim, as configurações inerentes aos três módulos citados são apresentadas em uma interface de administração **(d)** para o Administrador do ambiente de implantação do ProViNet.

ProViNet fornece escalabilidade utilizando uma arquitetura baseada no conceito chamado de *Resource Pool*. Cada *Pool* representa um conjunto de servidores de virtualização (*hypervisors*) interligados e controlados por uma plataforma de gerenciamento única. O *Pool* de Controle é utilizado para execução de máquinas virtuais com sistemas idênticos, que executam uma implementação de controlador pré-definida e compatível com o conceito de NBAPI. Os aplicativos escritos pelos usuários são executados em máquinas virtuais individuais alocadas no *Pool* de Execução. A comunicação entre os controladores do *Pool* de Controle e as VMs do *Pool* de Execução é provida pela NBAPI. Já a comunicação entre os controladores e os elementos do plano de dados nos *Slices* é provida pela SBAPI. As requisições realizadas pelos módulos do ProViNet aos *Pools* utilizam as interfaces de comunicação providas pela plataforma de virtualização adotada.

Finalmente, a requisição ao Provedor de Infraestrutura Virtual ocorre por meio de uma requisição HTTP, enviando um documento de descrição de infraestrutura virtual, a ser comentado mais adiante.

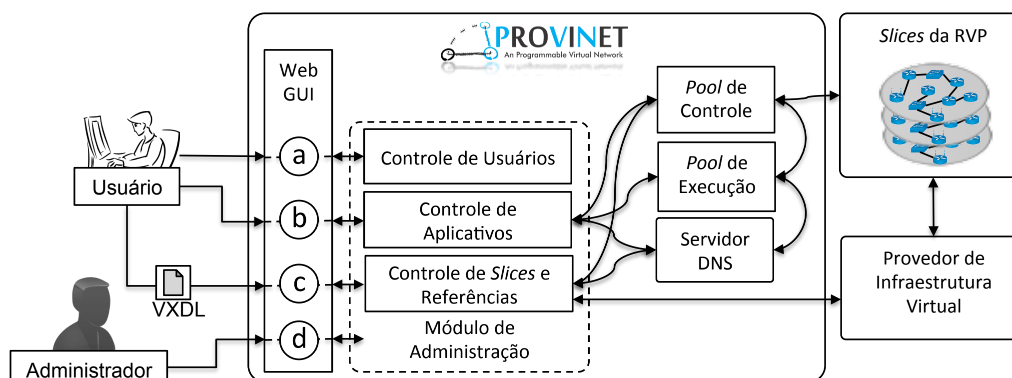

Figura 2. Arquitetura conceitual

Analisando novamente a Figura 2, percebe-se a esquerda o Usuário Final e a direita a Rede Virtual Programável. Entre esses elementos está a plataforma ProViNet, provendo a ligação entre eles. Nessa posição, o ProViNet oferece o gerenciamento da implantação de aplicativos, o controle de acesso para ambientes com múltiplos usuários e os requisitos não funcionais de disponibilidade, confiabilidade e escalabilidade. Com exceção do módulo de Controle de Usuários e de Administração, devido a simplicidade, os outros são detalhados nas subseções seguintes.

3.2. Controle de *Slices* e Referências

Um ambiente virtualizado envolve uma diversidade de máquinas virtuais as quais se inteligam por meio de uma rede virtual. Tal rede virtual pode ser provida de diversas formas dependendo das tecnologias suportadas pela plataforma de gerenciamento do ambiente virtual (são exemplos de plataformas o Eucalyptus, OpenNebula e OpenStack). Seja qual for a abordagem de provimento, o resultado é que o usuário terá uma rede virtual que interliga exclusivamente suas máquinas virtuais em um domínio único de *broadcast*. As redes virtuais, assim como as máquinas virtuais dos usuários compartilham os recursos físicos do provedor (ou *datacenter*). Desse modo, é conveniente e usual se referir ao conjunto de recursos de rede, computacionais e de armazenamento pertencentes a um usuário de *Slice*.

Não é função da plataforma ProViNet prover a formação do *Slice*, ou seja, iniciar as máquinas virtuais e configurar a rede virtual que interliga as mesmas. Portanto, o módulo Controle de *Slices* e Referências tem o papel de receber e encaminhar um documento de descrição de infraestrutura virtual especificando a topologia e os recursos a serem alocados pelo Provedor de Infraestrutura Virtual (PIV). Existem diversas propostas para esse tipo de documento, tais como Rspec (GENI), NDL-OWL(RENCI), NMC (OGF) e *Virtual Resources and Interconnection Networks Description Language* (VXDL) do projeto INRIA [Koslovski *et al.* 2008]. Neste trabalho é proposta a linguagem Programmable Virtual Network Description Language (PVNDL), uma adaptação da VXDL que contempla infraestruturas de rede virtual programável que possua plano de controle desacoplado do plano de dados.

Originalmente a linguagem VXDL define que os recursos possuem um nome e podem ter uma lista de funções, parâmetros, softwares e uma localização conforme se nota no trecho destacado de sua definição e apresentado abaixo:

```
<resource> ::= "resource" "(" <name> ")" "{"
               ["function" <elementary-functions>]
               ["parameters" <resource-parameters>]
               ["software" <software-list>]
               ["anchor" <location>] "}"
```

O atributo *function* pode assumir diversos valores, tais como *endpoints*, *aquisition* e *router*. Propõe-se a adição de um novo atributo inerente ao valor *router*, o qual nomeia-se *<controller-list>*. Na definição original, apenas o atributo *<ports>* era definido. Conforme apresentado abaixo, na linguagem PVNDL o novo atributo *<controller-list>* é adicionado como possível atributo de *router*, representando uma lista de controladores. Foi adotado uma lista pois alguns roteadores ou *switches* compatíveis com SDN aceitam redundância de controladores [1]. Cada elemento *<controller>* deve receber atributos definindo o tipo de conexão (ftp, ptcp e ssl são exemplos), o endereço IP e a porta em que o controlador remoto está configurado.

```
<elementary-functions> ::= <function> ("," <function> )*
<function> ::= "endpoint" | "aquisition" | "storage"
    | "computing" | "visualization" | "network_sensor"
    | "router" "(" "ports" <ports> [<controller-list>] ")"
<ports> ::= <number>
<controller-list> ::= <controller> ("," <controller> )*
<controller> ::= "(" <connection-type> <ip-address> <port> ")"
```

O usuário inicialmente deve elaborar este arquivo sem as informações sobre o controlador, pois estas serão adicionadas automaticamente pelo ProViNet durante o processo de requisição de infraestrutura virtual. Isso se justifica pelo fato do usuário não ter informações sobre os endereços de IP dos controladores, até mesmo porque eles são dinâmicos, conforme será apresentado adiante.

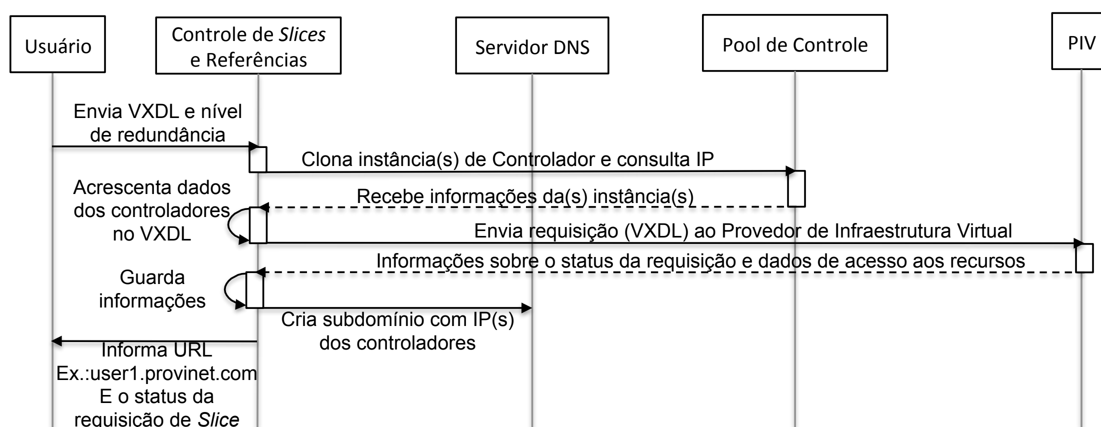

Figura 3. Diagrama de sequência da abordagem de criação de *Slices*

[1] O *switch* virtual openvSwitch por exemplo aceita a configuração de um *master* e diversos *slaves*

De acordo com o digrama da Figura 3, quando o usuário já elaborou o documento PVNDL, ele o envia ao módulo Controle de *Slices* e Referências via formulário de *upload*. No mesmo formulário, o usuário pode definir um nível de redundância de controladores no *Pool* de Controle que pretende ser associado ao *Slice* requerido. Para agilizar esse processo de requisição, mantêm-se no *Pool* de Controle uma instância de VM configurada com uma implementação de controlador compatível. Tal instância está fora de execução e serve apenas como base para clonagens (um serviço de cópia rápida provido pela plataforma de virtualização).

Quando o módulo em questão recebe de volta do *Pool* as informações da(s) instancia(s) clonadas, inicia-se o processo de adição no PVNDL das entradas necessárias, com os endereços IPs das máquinas, tipo de conexão e porta. Nesse momento o documento é então enviado ao PIV por uma requisição Web, gerada automaticamente pela plataforma. Vale ressaltar que para este trabalho, utilizamos o sistema HyFs [Wickboldt *et al.* 2012] como representante do PIV. Ao receber a resposta sobre o status da requisição e com informações sobre o acesso aos recursos do *Slice*, a plataforma registra no Servidor DNS os IPs dos controladores (os mesmos adicionados no documento PVNDL). Esse registro é a adição do nome do usuário como subdomínio do domínio *www.provinet.local*. Ou seja, cada usuário tem um subdomínio no qual são associados os IPs dos controladores criados. Caso o usuário tenha definido grau e redundância maior que um, o Servidor DNS aplicará o algoritmo Round Robin entre as referências. Abaixo segue um exemplo de configuração de uma domínio em que o ”user1”possui dois controladores e o ”user2”tem um.

```
...
                  NS       servidor.provinet.local.
servidor          A        xxx.xxx.xxx.xxx
user1             A        <IP-controlador-1>
user1             A        <IP-controlador-2>
user2             A        <IP-controlador-3>
...
```

Por fim, o módulo fornece ao usuário, a informação sobre a URL a ser utilizada em seus aplicativos para fazer chamadas aos serviços (Ex.: http://user1.provinet.local). São fornecidos, também, os dados de acesso aos recursos virtuais recebidos do PIV. Dessa forma, o aplicativo do usuário poderá enviar requisições aos serviços providos pelos controladores instanciados no *Pool* de Controle. Como ilustrado na Figura 4, o aplicativo do **user1** rodando no *Pool* de Execução poderia fazer a chamada *http://user1.provinet.local/getTopology* para consultar a topologia da rede virtual disponibilizada em seu *Slice*. Os elementos da rede do *Slice*, já foram configurados com os endereços IP dos controladores pelo PIV. Se o usuário tem mais de um controlador, as requisições dos aplicativos serão direcionadas para os controladores conforme o algoritmo Round Robin, implementado pelo Servidor DNS. Assim existirá um balanceamento de carga entre os controladores. Além disso, em caso de falha de um controlador, terá outro para atender as requisições.

3.3. Controle de Aplicativos

Uma vez que a estrutura de programabilidade já está estabelecida, ou seja, o usuário possui um *Slice* e um plano de controle devidamente configurado, resta ao usuário desenvolver e executar os aplicativos de rede. Para que o desenvolvimento seja possível,

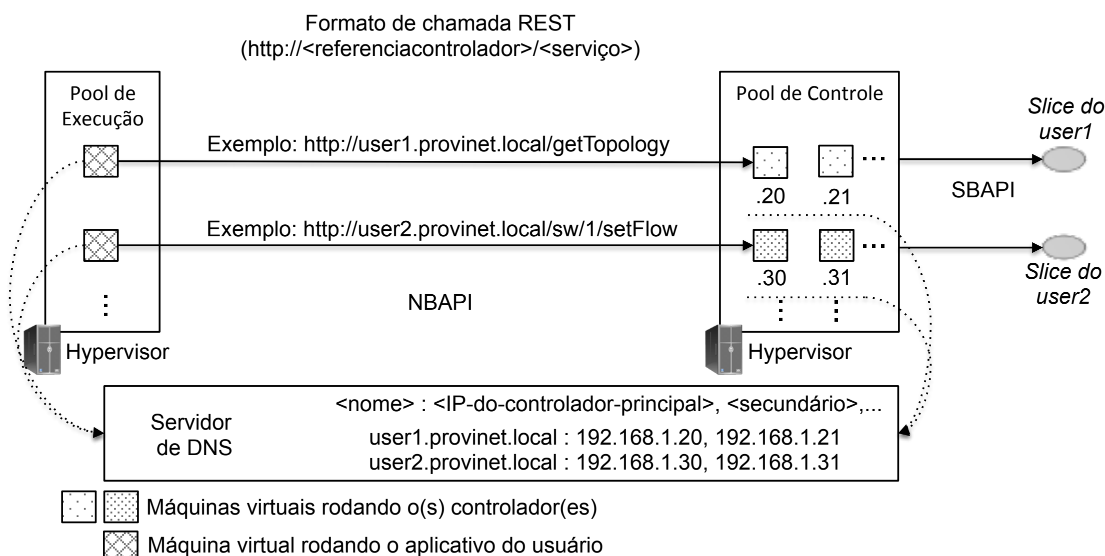

Figura 4. Abordagem de referências para provimento de escalabilidade e confiabilidade

o usuário precisa saber quais são os serviços a sua disposição. Por isso está acessível através da interface Web do ProViNet uma documentação completa sobre tais serviços, os quais dependem da implementação utilizada no *Pool* de Controle.

De acordo com o diagrama apesentado na Figura 5, no primeiro passo o usuário acessa a plataforma e requere a VM que rodará seus aplicativos. Nesta requisição são apresentados perfis de VM, variando o Sistema Operacional e quantidade de recursos, tais como memória, processamento e armazenamento. Após a escolha de um perfil, a plataforma, por meio de uma interface de comunicação com o *Pool* de Execução requere uma VM com tais características. Para evitar qualquer tipo de bloqueio por *firewall*, ao receber os dados da VM criada, o módulo Controle de Aplicativos configura um acesso VNC via *Browser*. Ou seja, o usuário é capaz de visualizar e interagir com a VM criada por meio de um terminal apresentado em seu *Browser*.

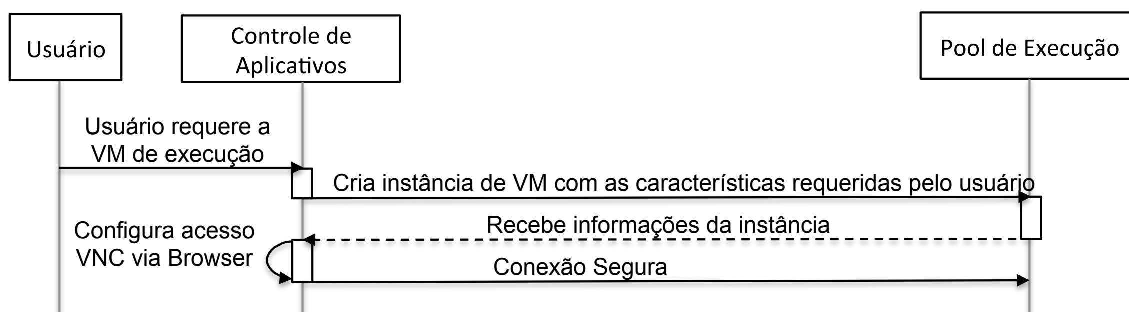

Figura 5. Abordagem de referências para provimento de escalabilidade e disponibilidade

Para controlar o aplicativo de rede, o usuário deve interagir com a VM do *Pool* de Execução através da interface gráfica do ProViNet. Assim ele é capaz configurar tal VM com todos os requisitos necessários para execução de seu aplicativo, tais como bibliotecas e pacotes. Os aplicativos podem ser escritos em qualquer linguagem, basta que seja

capaz de enviar e receber chamadas HTTP segundo a arquitetura REST. Um exemplo de chamada é demonstrado na Figura 6. Quando o aplicativo do usuário faz uma chamada de serviço, a URL de destino é traduzida pelo Servidor DNS pra um IP válido, o qual pertence a lista de IPs de controladores cadastrados no subdomínio daquele usuário. Então a chamada do aplicativo é de fato enviada ao IP retornado do Servidor DNS. O controlador que recebe a requisição está no *Pool* de Controle, e de lá faz as chamadas via SBAPI de acordo com o serviço requerido. Ao receber as respostas dos dispositivos de rede presentes no *Slice*, elabora uma resposta em formato JSON e retorna ao aplicativo do usuário que fez a requisição.

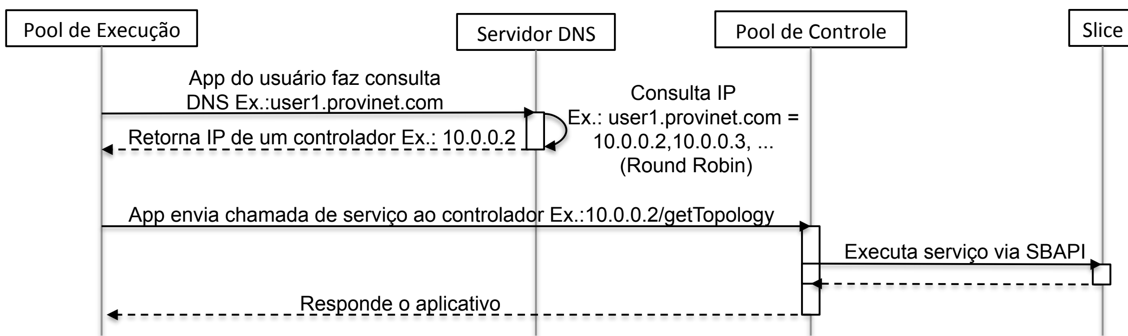

Figura 6. Diagrama de sequência ilustrando requisição de serviços

Existe um conjunto de serviços implementados pelo controlador que roda no plano de controle. Todavia, diante da necessidade de um novo serviço, o qual não está disponibilizado no conjunto padrão de serviços do controlador, o mesmo pode ser desenvolvido e instalado. Em geral os controladores seguem uma arquitetura que permite o acoplamento de módulos com a implementação de novos serviços. Baseado nas informações de quais controladores pertencem a quais usuário, a plataforma ProViNet auxilia o usuário na instalação dos módulos.

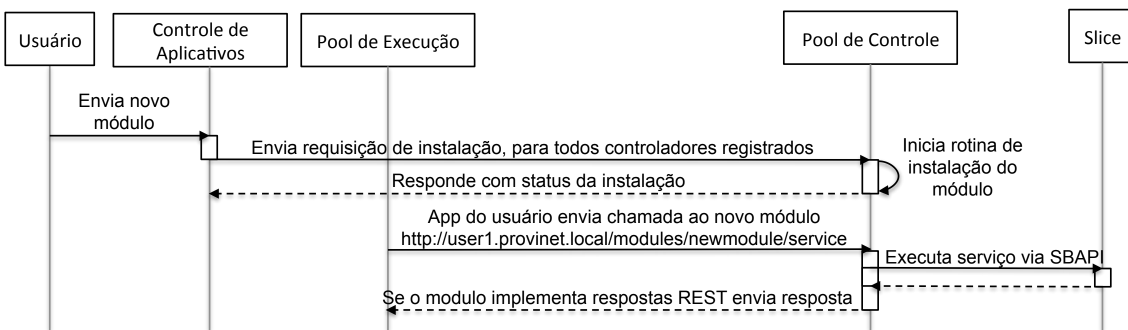

Figura 7. Diagrama de sequência ilustrando a implantação de um novo módulo

Para implementar o módulo a ser instalado, o usuário deve seguir os tutoriais[2] disponibilizados pelo órgão desenvolvedor do controlador adotado no plano de controle. Conforme ilustrado na Figura 7, o processo de instalação de um novo

[2]Tomando como exemplo o controlador Floodlight, estão disponíveis no endereço *http://www.openflowhub.org/display/floodlightcontroller/How+to+Write+a+Module*, um conjunto de instruções para o desenvolvimento de novos módulos.

módulo se inicia pelo envio do módulo compactado para a plataforma. O módulo responsável então o encaminha a todos os controladores do usuário que estão associados a um determinado *Slice*. Em cada controlador, um *deamon* configurado para fazer a implantação de módulos executa uma rotina de instalação. Uma vez que os módulos estão instalados, os aplicativos dos usuários podem enviar requisições a ele (Ex.: *http://user1.provinet.local/modules/newModule/service*).

4. Protótipo

Com o objetivo de demonstrar a viabilidade da arquitetura conceitual detalhada na Figura 2 do Capítulo 3, foi desenvolvido um protótipo. Sua implementação baseia-se no desenvolvimento dos cinco principais módulos, a Interface Web, o Controle de Usuários, o Controle de Aplicativos, o Controle de Slices e Referências e a Administração do ProViNet. Tais módulos foram implementados utilizando o *framework* Django 1.4.3, a linguagem Python 2.7.3 e o sistema gerenciador de banco de dados PostgreSQL 9.1.6. A fim de fornecer maior compatibilidade do sistema, foi utilizado o servidor Web Apache 2.2.23.

O módulo Controle de Usuários fornece na interface do ProViNet, formulários de registro e login. O Controle de Aplicativos apresenta na interface uma área especial para apresentação dos serviços disponibilizados no plano de controle e uma área para requisição, controle e interação com as VMs no *Pool* de Execução. É disponibilizado pelo módulo de Controle de *Slices* e Referências um formulário para *upload* do documento PVNDL e definição do nível de redundância. Finalmente, a área de Administração apresenta um conjunto de funcionalidades de configuração, que incluem o endereço de IP e dados de acesso do *hypervisor master* dos *Pools*, do Servidor DNS, do Provedor de Infraestrutura Virtual e ainda a definição dos perfis de máquinas virtuais que serão disponibilizadas aos usuários.

Os *Pools* de Execução e de Controle são, na prática, servidores com alguma plataforma de virtualização instalada. No protótipo desenvolvido foi utilizado o *hypervisor* XenServer da CITRIX. Sendo assim, a comunicação entre os módulos da plataforma ProViNet e os *Pools* se dá pela utilização do XenServer SDK. Com o SDK é possível controlar e monitorar o *hypervisor* através de chamadas XML-RPC. O último módulo da plataforma é o Servidor DNS, o qual foi instalado em uma máquina com os sistemas Bind9 e Apache2. Para o controle dinâmico de configuração do DNS, um serviço web foi implementado para que, a partir de chamadas HTTP a uma URL específica, seja feita a adição e remoção de entradas no arquivo de configuração do bind9.

A implementação de controlador utilizada foi o Floodlight, por prover uma API RESTfull que possibilita o consumo dos serviços disponibilizados no controlador por meio de chamadas HTTP. Outras implementações poderiam ser utilizadas, desde que seja possível a instalação de módulos para provimento de serviços customizados e a disponibilização de serviços utilizando a arquitetura REST.

5. Caso de Estudo e Análise de Resultados

Para avaliar a solução apresentada, foi elaborado um caso de estudo que aborda cada um dos diagramas de sequência apresentados no Capítulo 3. O cenário de execução é composto por servidores Intel Xeon CPU E3-1220 3.1GHz, 4GB RAM, com XenServer

Clonar e iniciar duas instâncias	Adicionar referência ao PVNDL	Criação da rede virtual pelo PIV	Configuração de subdomínio	TOTAL
12,824s	0,003s	57,81s	0,04s	75,677s

Tabela 1. Tempos para requisição de *Slice* e configuração de referências

6.1 representando o *Pool* Controle e de Execução. O controlador OpenFlow utilizado foi o Floodlight v0.90, e é iniciado, por *script* durante a inicialização da VM (Ubuntu 12.04 com 1 vCPU e 384MB RAM) no *hypervisor*. Para executar o *framework* Django com o ProViNet foi utilizado um *laptop* Intel Core i7 2.8GHz e 4GB RAM. Por fim, o Servidor DNS foi instalado e configurado em um terceiro PC (Intel Core 2 Duo 2.33GHz e 4GB RAM) na mesma rede local que os outros PCs.

O caso de estudo se inicia com o registro do usuário, porém tal processo não é avaliado devido a sua simplicidade. Portanto, é apresentada na Tabela 5 a avaliação da sequência ilustrada na Figura 3. Considera-se nessa avaliação que os usuário escolheu nível de redundância 2, ou seja seu *Slice* terá dois controladores associados. Considera-se também que a infraestrutura descrita no PVNDL foi uma topologia em arvore com 7 *switches* e 4 *hosts*.

Vale ressaltar que o tempo que o PIV leva para prover a infraestrutura deve variar de acordo com a abordagem de mapeamento do PVNDL para a infraestrutura física. O tempo apresentado nesse quesito considerou o sistema HyFs [Wickboldt *et al.* 2012] como provedora da infraestrutura, portanto esse mapeamento é feito pela criação de um *overlay*, em que *switches* e *hosts* sao máquinas virtuais com *software switches* e instâncias Ubuntu respectivamente. Os tempos parciais e totais ilustrados representam a média de tempo gasta na execução das ações a que se referem.

O próximo passo é a requisição da VM de execução, ou seja a VM em que o usuário vai instalar o aplicativo de rede e que tem acesso via NVC no Browser. Para simplificar o caso de estudo, se o usuário requisitar uma VM com as mesmas configurações daquela que roda o controlador, o tempo gasto será de 6,890 segundos. Até mesmo porque a abordagem de cópia com o conceito de clone também é utilizada nesse caso.

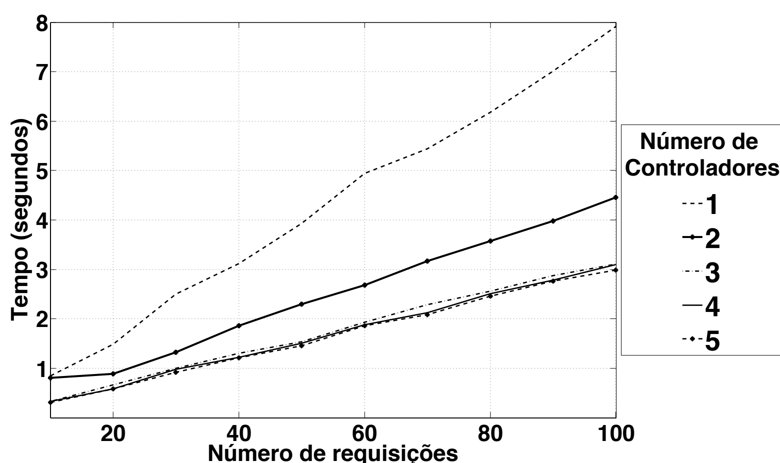

Figura 8. Performance do plano de controle com balanceamento de carga

Para medir desempenho das requisições do aplicativo do usuário, foram realiza-

dos experimentos com varições no nível de redundância anteriormente citado (2). Dessa forma, é possível acompanhar a variação no desempenho das chamadas, pois ao utilizar uma maior quantidade de controladores, o balanceamento de carga realizado pelo Servidor DNS torna o processo mais rápido. Tal fato pode ser acompanhado no gráfico da Figura 8, que mostra o tempo gasto para concluir X requisições, sendo X, valores entre 10 e 100 com intervalos de 10 unidades. A requisição executada para obtenção dos valores apresentados foi *http://www.provinet.local/wm/core/controller/switches/json*, a qual retorna a lista de *switches* presentes no *Slice*.

Analisando os valores apresentados no gráfico 8, percebe-se que o balanceamento de carga provido pela abordagem proposta e gerenciado pelo ProViNet é efetivo e implica em uma redução do tempo gasto para execução das requisições. Entretanto o ganho se torna menos significativo para um número de controladores maior que 3. Ou seja, utilizando apenas 1 controlador, foram gastos 7,915 segundos para concluir 100 requisições, ao passo que com 2 controladores esse valor caiu para 4,452. O ganho ao aumentar o número de controladores de 4 para 5, não é tão expressivo quanto de 1 para 2, saindo de 3,098 para 2,989 segundos nesse caso.

Para avaliar a funcionalidade do ProViNet de instalar módulos sob demanda, desinstalou-se do Floodlight um módulo que originalmente já vem instalado, o módulo de *firewall*. Compactou-se tal módulo em um arquivo e, através da interface do ProViNet, foi feita a requisição de instalação. Uma vez que o módulo Controle de Aplicativos possui cadastrado o endereço IP de todos os controladores e seus respectivos usuários, após receber o arquivo por *upload*, um *script* de envio é disparado. O papel desse *script* é acessar via ssh a VM de cada controlador, fazer a cópia do novo módulo para uma pasta específica na VM e ativar um segundo *script* na VM que faz a instalação do mesmo. Esse segundo *script* segue as informações disponibilizadas no site do Floodlight para instalação de módulos. O tempo médio gasto nesse processo foi de 23,435 segundos.

6. Conclusões e Trabalhos Futuros

A diversidade de ambientes computacionais requerem distintos serviços de comunicação em redes. As soluções desenvolvidas no passado, e implementadas de acordo com as vontades das fabricantes de equipamentos de redes, podem não ser mais suficientes. Entretanto, com o surgimento de propostas abertas de virtualização e programabilidade, como as Redes Definidas por Software, a criação de novas soluções se torna, de certa forma, mais democrática. Ou seja, depende menos dos anseios financeiros das grandes fabricantes.

Todavia, os desafios agregados ao gerenciamento de ambientes de Rede Virtual Programável (RVP) ainda representam um grande desafio. Neste trabalho propomos a plataforma ProViNet para o gerenciamento da implantação de aplicativos de rede em ambiente de RVP. A plataforma ProViNet contribui com uma arquitetura escalável, utilizando o conceito de *Resource Pool*, com uma abordagem para a instalação dinâmica de novos módulos no plano de controle, com a elaboração de uma linguagem de definição de infraestrutura virtual de rede virtual programável, chamada VXDL2, e por fim, com o desenvolvimento de um sistema com interface de acesso Web, facilitando a compreensão e interação com usuários finais.

Como trabalhos futuros pretende-se investigar mais precisamente, como os am-

bientes de *Cloud* poderiam prover pratilheiras de serviços de rede dinâmicas. As quais seriam ocupadas por soluções dos próprios usuários.

Referências

(2012). ProtoGENI. Disponível em: http://www.protogeni.net/trac/protogeni. Acessado em: Dezembro 2012.

Campbell, A. T., Meer, H. G. D., Kounavis, M. E., Miki, K., Vicente, J. B., e Villela, D. (1999). A survey of programmable networks. *COMPUTER COMMUNICATION REVIEW*, 29:7–23.

Chowdhury, N. e Boutaba, R. (2009). Network virtualization: state of the art and research challenges. *Communications Magazine, IEEE*, 47(7):20 –26.

Chowdhury, N. M. K. e Boutaba, R. (2010). A survey of network virtualization. *Comput. Netw.*, 54(5):862–876.

GENI (2011). Global Environment for Network Innovations. Disponível em: <http://www.geni.net/>. Acessado em: Julho 2012.

Hausheer, D., Parekh, A., Walrand, J., e Schwartz, G. (2011). Towards a compelling new internet platform. Em *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, páginas 1224 –1227.

Kanaumi, Y., Saito, S., e Kawai, E. (2010). Deployment of a programmable network for a nation wide randd network. Em *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, páginas 233 –238.

Kopsel, W. (2011). Ofelia - pan-european test facility for openflow experimentation.

Koslovski, G. P., Primet, P. V.-B., e Charão, A. S. (2008). Vxdl: Virtual resources and interconnection networks description language. volume 2 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, páginas 138–154. Springer.

Lantz, B., Heller, B., e McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. Em *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, páginas 19:1–19:6, New York, NY, USA. ACM.

Lin, P., Bi, J., Hu, H., Feng, T., e Jiang, X. (2011). A quick survey on selected approaches for preparing programmable networks. Em *Proceedings of the 7th Asian Internet Engineering Conference*, AINTEC '11, páginas 160–163, New York, NY, USA. ACM.

OpenStack (2011). Open source software for building private and public clouds. Disponível em: http://www.openstack.org/. Acessado em: Julho 2012.

Rubio-Loyola, J., Galis, A., Astorga, A., Serrat, J., Lefevre, L., Fischer, A., Paler, A., e Meer, H. (2011). Scalable service deployment on software-defined networks. *Communications Magazine, IEEE*, 49(12):84 –93.

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