VÍTOR ANTÓNIO GONÇALVES RIBEIRO DA CUNHA

SERVICE FUNCTION CHAINING PARA NFV EM AMBIENTES CLOUD

SERVICE FUNCTION CHAINING FOR NFV IN CLOUD ENVIRONMENTS


Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

VÍTOR ANTÓNIO GONÇALVES RIBEIRO DA CUNHA

SERVICE FUNCTION CHAINING PARA NFV EM AMBIENTES CLOUD

SERVICE FUNCTION CHAINING FOR NFV IN CLOUD ENVIRONMENTS

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor João Paulo Silva Barraca, Professor Assistente Convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui Luís Andrade Aguiar, Professor Catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor André Ventura da Cruz Marnoto Zúquete
professor auxiliar da Universidade de Aveiro

vogais / examiners committee

Doutor Francisco Manuel Marques Fontes
consultor sénior da PT Inovação

Prof. Doutor João Paulo Silva Barraca
professor assistente convidado da Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço toda a ajuda e tempo disponibilizado pelos orientadores desta dissertação, assim como pelo Igor Cardoso, pelo Prof. Doutor Diogo Gomes e pelo Prof. Doutor Daniel Corujo. Aproveito também para agradecer ao lado da PT Inovação, Pedro Neves, Rui Calé, Mário Rui Costa e muitos outros (demasiados para nomear, mas igualmente importantes) pelo ambiente fantástico no qual se pôde desenvolver esta dissertação. Por fim agradeço aos meus pais por todo o apoio que sempre deram ao longo dos anos.

Palavras Chave

vDHCP, vDPI, shaper, SDN, NFV, VNF, cloud, openstack, homegateway, virtualização, neutron, opendaylight, SFC, chaining, traffic steering, RADIUS, AAA

Resumo

Service Function Chaining, Virtual Network Functions e Cloud Computing são os conceitos chave para resolver (em "grande plano") uma necessidade actual dos operadores de telecomunicações: a virtualização dos equipamentos na casa dos consumidores, particularmente o Home Gateway. Dentro deste contexto, o objetivo desta dissertação será providenciar as Funções Virtuais de Rede (tais como um vDHCP, Classificador de Tráfego e Shaper) assim como respectivas APIs necessárias para se atingir essa solução de "grande plano". A solução utilizará tecnologias Open Source como OpenStack, OpenVSwitch e OpenDaylight (assim como contribuições anteriores do Instituto de Telecomunicações) para concretizar uma Prova-de-Conceito do Home Gateway virtual. Após o sucesso da primeira PdC iniciar-se-á a construção da próxima prova, delineando um caminho claro para trabalho futuro.

Keywords

vDHCP, vDPI, shaper, SDN, NFV, VNF, cloud, openstack, homegateway, virtualization, neutron, opendaylight, SFC, chaining, traffic steering, RADIUS, AAA

Abstract

Service Function Chaining, Network Function Virtualization and Cloud Computing are the key concepts to solve (in “big-picture”) one of today’s operator’s needs: virtual Customer Premises Equipments, namely the virtualization of the Home Gateway. Within this realm, it will be the purpose of this dissertation to provide the required Virtual Network Functions (such as a vDHCP, Traffic Classifier and Traffic Shaper) as well as their respective APIs to build that “big-picture” solution. Open Source technologies such as OpenStack, OpenVSwitch and OpenDaylight (along with prior work from Instituto de Telecomunicações) will be used to make a working Proof-of-Concept of the Virtual Home Gateway. After the success of the first PoC, starts the construction of the next PoC and a path for future work is laid-down.

Contents

List of Figures

List of Tables

Glossary

ACL	Access Control List
API	Application Programming Interface
CAPEX	Capital Expenditures
CCAP	Converged Cable Access Platform
COTS	Commercial off-the-shelf
CPE	Costumer Premises Equipment
DPI	Deep Packet Inspection
ETSI	European Telecommunications Standards Institute
HGW	Home Gateway
IaaS	Infrastructure as-a-Service
IETF	Internet Engineering Task Force
IPAM	IP Address Management
NIST	National Institute of Standards and Technology
NFV	Network Function Virtualization
NSH	Network Service Header
ONF	Open Networking Foundation
OPEX	Operating Expenses
OS	Operating System
ODL	OpenDaylight
OVS	Open vSwitch
PCEF	Policy and Charging Enforcement Function
PoP	Point-of-Presence
PaaS	Platform as-a-Service
pHGW	Physical Home Gateway
PoC	Proof-of-Concept
QoS	Quality of Service
RGU	Revenue Generation Unit
RSP	Rendered Service Path
SaaS	Software as-a-Service
SBR	Service Based Routing
SDN	Software-Defined Networking
SF	Service Function
SFC	Service Function Chaining
SFF	Service Function Forwarder
SFP	Service Function Path
SoC	System-on-Chip
TDF	Traffic Detection Function
ToS	Type of Service
vCPE	Virtual Costumer Premises Equipment
vDHCP	Virtual DHCP
vHGW	Virtual Home Gateway
VM	Virtual Machine
VNF	Virtual Network Function

CHAPTER 1

Introduction

The adoption of Cloud Computing (subsection 2.4) allows for computational resources to be subscribed like utilities and used over a network connection which, versus having to own those resources in physical hardware, not only gives an economical advantage but also better agility and faster time-to-market.

As the services provided by the cloud are only accessible through the network that connects the end-points, networking is therefore a critical aspect of any cloud. Still, networking is not restricted to play only the supporting role of the Cloud Computing infrastructure. Virtualization allows the shifting of dedicated hardware functions to Virtual Network Functions (VNFs), which can be run using the cloud itself to extend, implement and improve networking capabilities, allowing for on-demand capacity adjustment, topology adjustments, new functions or even instantiate new networks altogether.

This opens way to further explore Network Function Virtualization (NFV) within Cloud Computing environments, harnessing the advantages of the cloud and incorporating them into the realm of other network-related problems, such as Content Caching, Virtual Evolved Packet Core or Costumer Premises Equipment (CPE) virtualization for instance.

I will begin by describing the context of this dissertation's work, then moving into the objectives set for the dissertation, the contributions made and lastly will describe the document's structure.

1.1 Motivation / "big picture"

This dissertation will be a part of the CPEs virtualization, particularly the Virtual Home Gateway (vHGW). The vHGW is the "big-picture" and motivation for this dissertation's work (the "small-picture"), but ultimately its specification and requirements should be seen as assumptions and limitations. I will start by presenting the Home Gateway (HGW) problem statement, followed by an inside look of the HGW, ending with a solution path for the vHGW.

1.1.1 THE HGW PROBLEM STATEMENT

The end-customers' main drive to subscribe services is usually their usefulness (value-added/perceived value of the service itself), not the leased hardware features just for the sake of it. Nevertheless, the level of service and the kind of services that are available to them are directly related to the capabilities of their CPEs. [1]
This requires operators to make difficult compromises. By one hand, they must not overspend in equipments, as these are not the main drive for costumers and they account negatively on their Capital Expenditures (CAPEX). By the other, telecoms must also ensure their equipments will keep-up with changing customer demands and the introduction of new services (new Revenue Generation Units (RGUs) to increase profitability).

With an increasingly "over IP" world of services, the HGW ends-up assuming the role of an enabler of other services and the gateway for more RGUs per customer. However, today's HGWs pose a number of challenges to the agility of Telecoms' core business:

- Telecoms often have in operation multiple models of gateways (from different eras and/or different access technologies), each requiring their own unique firmware.
- Deploying a new feature to all variants of HGWs is a virtually impossible task.
- Should an emergency update be required (a security patch) it will be very hard to patch all models timely.
- CPEs are usually sourced to (third-party) partners, which adds a whole layer of contractual red-tape, dependence on a third-party to take action, negotiations and there may be even competing interests on the table.
- HGWs are embedded systems, not computational power-houses. This will impose a strong limit as to which new features can even be considered and what will be their capabilities.

So, in order to find a solution to this "big-picture" problem, one must first take a closer look inside the Home Gateway to draw an action plan.

1.1.2 LOOKING INSIDE THE HGW

The HGW is an embedded device whose functionality is usually provided by a System-on-Chip (SoC) plus some peripheral electric/radio-electric/optical hardware (transceivers for ports for instance). Regardless of the level of integration the SoC/HGW may have, its functionality is achieved by two major categories of logical parts:

1. Network-related hardware (such as Ethernet ports, built-in Ethernet switch and WiFi module).
2. Computational resources (such as a built-in CPU, RAM and flash storage).

The computational resources of the HGW are used to run an Operating System (OS) (often times Linux) which will be in charge of running functions belonging to the Host Layer (OSI Layer-3 and above), for instance DHCP, DNS Relay, SPI Firewall, configuration interfaces (Web-Interface, Telnet, SSH) and other value-added services (such as NAS, Printing Server, DDNS updater, etc).
The network-related hardware is in charge of handling network interactions that belong to the Media Layer (for instance Ethernet, PPP, IPv4, etc), interfacing directly with the electric, radio-electric

and even optical signals. However, this does not mean that all Media Layer protocols are handled strictly in dedicated hardware units. There are various levels of Hardware Offloading/OS dependence for different protocols (therefore reliance on computational resources), which vary according to the grade and age of the equipment.

Given that HGWs are located in the customer's premises, the cost of replacement is higher than just the equipment itself, as that action must be performed in-location which incurs in added transit costs (hefty when compared to the cost of the equipment itself). This means that, from an economical point of view, the HGW should have a usable life-cycle as long as possible, without having to pay much of a premium for it. Because the HGW is in the networking business (not computation), this ideally translates to an equipment whose life-cycle closely matches the effectiveness of its network-related hardware and underlying networking technologies.

However, because software functions run locally, there is a computational requirements side to this equation and (given this is an embedded system) a significant contribution made by the software development costs for that particular equipment. Because of the later, the HGW is likely to face early-obsolescence (with the cost penalty this brings).

1.1.3 SOLUTION PATH

Ideally we want a system with the best future-proofing in the network-related technologies and as little computational dependencies as necessary (to prevent early-obsolescence due to the cost of developing for that embedded system).

This means transforming the CPE into a Physical Home Gateway (pHGW) with all the necessary network interfaces but connected to an operator's Cloud via a tunnel, being the software functions (and their computational requirements) decoupled from the equipment itself and offloaded to that Cloud – this while transparently providing the end-user with the full functionality like it was run locally. The pHGW only needs to handle the tunnel to the operators' Cloud and the functions of the Media Layer (OSI Layer-2 and bellow), being the tunnel an extension of the virtual network (subnet) at a switch level (often called a "Layer-2 tunnel"), thus the customers' devices are effectively connected to the virtual network as if it were any other local network handled by the switch of a typical HGW.

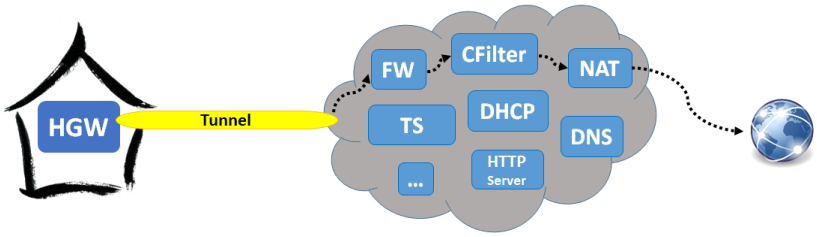

Figure 1.1: High-level vision of a cloud-powered HGW

Service Function Chaining (SFC) would aid replicating existing behavior of the HGW (such a service chain of *Routing + NAT + Firewall*) and also enable the faster creation of new (and more complex) services out of existing network functions.

However, in order to build those chains and put this concept in motion, we first need the required VNFs (this is where this dissertation will contribute) and proper support of the cloud platform to

allow for SFC and transparently connect external devices within a virtual network assigned for the HGW of the subscriber (through some form of tunnel).

As it happened, there was prior work within *Instituto de Telecomunicações* which gave OpenStack those capabilities in a neatly integrated fashion (with the APIs and semantics of the platform). Igor Cardoso delivered a Masters Dissertation on the topic of connecting external devices in an OpenStack Neutron network (via a GRE tunnel) [2] and Carlos Gonçalves along with João Soares delivered a way of performing SFC within an OpenStack Neutron network (a work which would be later maintained also by Igor Cardoso). More details to follow in the *State of Art* (chapter 3).

These are the broad-strokes that set the "big-picture" in motion, next I will move to the "smaller-picture" and focus on what are the actual dissertation's goals.

1.2 GOALS

The dissertation has its own clear set of goals, which were defined in two phases of work (that follow the needs of each phase of the vHGW). The context about each phase of the "big-picture"/vHGW will be given in the respective sections of this dissertation's work.

In the first phase, the goal is to design and implement a working, drop-in replacement for the OpenStack's Virtual DHCP (vDHCP), which must automatically perform the registration of newly connected devices (Neutron port creation) and perform some (device) session control through communication with an AAA.

In the second, the goal is to quickly adapt to a paradigm shift (towards policy-driven chaining) and provide a Deep Packet Inspection (DPI) VNF (called Traffic Classifier), which will be used to mark traffic (to select the SFC) according to policy, device and application protocol. Alongside the Classifier, this dissertation must also provide a Traffic Shaper VNF which will apply throttles according to policy, device and classifier mark. Both the Traffic Classifier and the Traffic Shaper must be configurable "on-the-fly" through a REST based Application Programming Interface (API) (for policy enforcement), needing these APIs to be added to an existing component called VNF-API (for convenience of the vHGW works).

1.3 CONTRIBUTIONS

The dissertation's work herein described in the context of Instituto de Telecomunicações and its projects is also being used in similar Virtual Costumer Premises Equipment (vCPE) research and Proof-of-Concepts (PoCs) within PT Inovação.

During the final stages of phase 1, there was an early submission (full-size paper) made to IEEE Globecom 2015[1] conference (as co-author, being Igor Cardoso the main author and scientific supervision performed by this dissertation's supervisors). However, given the early nature of the work, the submission did not make it through to the conference or the workshops.

A new submission (as main author, being Igor Cardoso co-author and scientific supervision performed by this dissertation's supervisors) with the latest work, more data and a clearer focus has

[1] `http://globecom2015.ieee-globecom.org/`

been made to the IEEE NetSoft 2016[2] conference (and is pending the review process).

1.4 DOCUMENT STRUCTURE

This document is organized in 8 chapters, the first of which is this introduction. Then, it follows:

Chapter 2 - "Key Concepts" A chapter where fundamental concepts are presented, but their scope does not really fall under a literature review chapter.

Chapter 3 - "State of the Art" Here is presented the literature review and research of the current state of the art of standards, tools and other relevant research. The existing work within Instituto de Telecomunicações (that supports the vHGW) is also presented here.

Chapter 4 - "Design and Specification (Phase 1/vDHCP)" An introduction to the "big-picture" assumptions and limitations set for the phase 1 of the vHGW and subsequent design and specification of the vDHCP function.

Chapter 5 - "Implementation and Results (Phase 1/vDHCP)" The compromises made and actual implementation rational of the vDHCP function. Following come the results taken from both the vDHCP and relevant "big-picture" results from the phase 1 PoC.

Chapter 6 - "Design and Specification (Phase 2/Classifier + Shaper)" The new "big-picture" assumptions and limitations set for the phase 1 of the vHGW and subsequent design and specification of the newly required functions (Classifier and Shaper).

Chapter 7 - "Implementation and Results (Phase 2/Classifier + Shaper)" Analyses the compromises made in this second phase and presents the actual implementation rational of the Classifier and Shaper VNFs (along with their REST API). The results follow.

Chapter 8 - "Closing Thoughts" In this final chapter the conclusions are presented, along with the suggested path for future-work.

[2] `http://sites.ieee.org/netsoft/`

CHAPTER 2

Key Concepts

This chapter presents the general concepts which are fundamental to this dissertation's work, but do not really fit under a literature review chapter (such as the Sate of Art) neither fit under an Introduction chapter.

2.1 software defined networking

With a growing demand for data, the coming and going popularity of Peer-to-Peer protocols, the rise of Internet-of-Things and the shift towards providing all kinds of legacy services over the Internet Protocol, coping with the ever changing network needs and growing complexity has become less and less human manageable. Therefore, new ways had to be devised so that reliance on human management can be minimized and (when a human does need to be involved) higher-level abstractions can help deal with unneeded complexity.

A solution came in the form of Software-Defined Networking (SDN), that separates the connections and equipments through which packets flow (forwarding plane) from the control decisions made as to what should be done with those packets (control plane), turning the network into a programmable entity.

This makes networks more agile, as it allows for network administrators to rapidly react and effectively change the network (at the distance of a click) to cope with the ever changing network-traffic demands and/or new requirements.

It also allows for central management, in which the separate equipments that amount for the whole network can be controlled through a (logically centralized) SDN Controller. The SDN Controller is not only able of performing managing tasks and keep track of the network but can also create high-level abstractions of the actual network topology (for instance a single logical switch) which allows for applications to focus in solving their own task (rather than having to handle network complexity).

Last but not least, being a programmable entity also makes way for programmatic configuration. This allows network administrators to write their own computer programs to perform various administration tasks (such as optimization, automatic configuration and security policy enforcement) which are able to be carry-on around the clock and can react immediately to changes in the network as per their programming.

On another note, an integral part of SDN is the need to have a communication's protocol between the control plane and forwarding plane, so that the decisions made in the control plane are actually carried-out in the forwarding plane. The Open Networking Foundation (ONF), which is a non-profit organization charged with the task to promote the adoption of SDN, created OpenFlow as the open-standard protocol for this task (and is in fact the first standard protocol of this kind). There are, however, some other competing protocols (such as Cisco's proprietary OpFlex).

The agility, central management and programmatic configuration make SDN a key to the purposes of the "big-picture" vHGW.

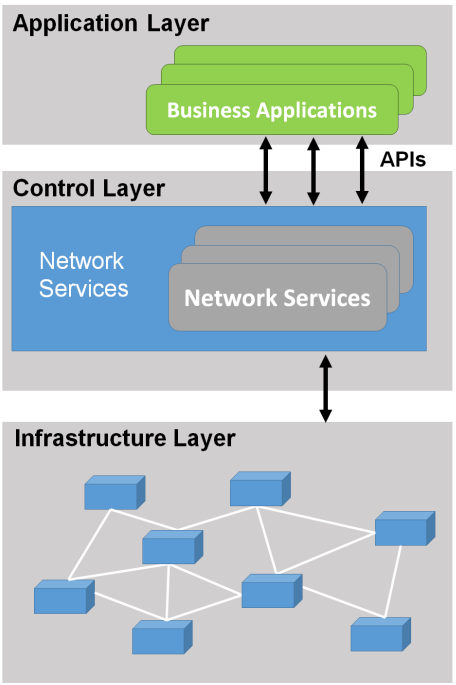

Figure 2.1: An overview of SDN across its various layers of action.

2.2 NETWORK FUNCTION VIRTUALIZATION

NFV is, simply put, the capacity to transform a network function that runs in its own mission-specific hardware (for instance a router with NAT) into a computational analog that is able to perform that same function but in a general purpose machine.

This capability greatly enhances agility, as network functions can now be instantiated (or terminated) on-demand. Time-to-market is also reduced, as business decisions don't need to wait for equipments to be back-ordered to their manufacturer and then physically shipped to the desired location (upon availability). Not having to purchase proprietary hardware which doesn't scale at will and may perform some form of vendor lock-in also creates favorable CAPEX prospects. Likewise, maintaining just general purpose hardware also creates favorable Operating Expenses (OPEX) prospects.

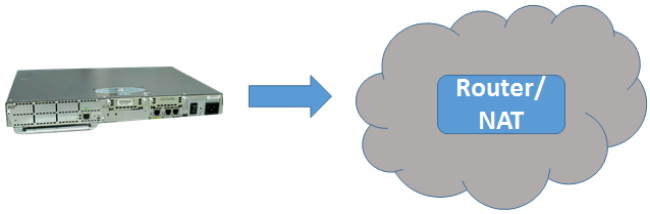

Figure 2.2: A NFV illustration, the virtualization of a router with NAT

Although SDN and NFV can exist separately, when coupled together one suddenly has the capability to instantiate (and manage) full networks on-demand. This capability is a key feature for the purposes of the vHGW.

2.3 SERVICE FUNCTION CHAINING

SFC is the ability to create new (and more complex) network services out of the "concatenation" of existing ones, through the transparent redirection (steering) of the output of one function to the input of the next. The ordered list which contains the functions to be "concatenated" is called a Service Function Chain, while the redirection process is called Traffic Steering.

The Traffic Steering mechanism is (usually) accompanied by a Traffic Classifier, which acts as a selector of network traffic that must go through the chain. Typical classification (selection) is done with a combination of OpenFlow's 12-tuple, which has the following fields: Ingress port; Ethernet source; Ethernet destination; Ethernet type; VLAN id; VLAN priority; IP source; IP destination; IP protocol; IP ToS/DSCP bits; TCP/UDP source ports; TCP/UDP destination ports;

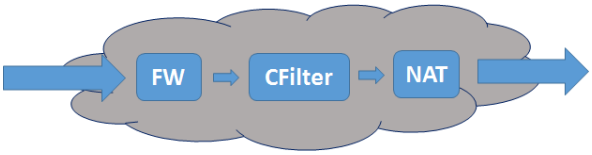

Figure 2.3: A SFC example, transparently replace the regular NAT function with a secure one that also does firewalling and content-filtering.

2.4 CLOUD COMPUTING

Overview. Defining Cloud Computing can be a very treacherous task, as industry introduces many variations to the definition according to what is one's own purpose (read commercial interest – Amazon spins it one way for AWS[1], Salesforce spins it another[2], and so on). As such, through-out this dissertation I will be taking as reference the National Institute of Standards and Technology (NIST)'s definition of Cloud Computing as published in [3] (with minor adaptations where applicable).

[1] `https://aws.amazon.com/what-is-cloud-computing/`
[2] `http://www.salesforce.com/eu/cloudcomputing/`

One can define Cloud Computing as a way to take computational resources across a cluster of machines and make them available through a (configurable) pool of shared resources, having ubiquitous computing, on-demand network access to those resources and cost as prime motivators.

The motivation is a crucial part of the definition, as other concepts share a similar definition (such as High-Performance Computing) however it is the motivation that makes the core concept (architecture path and end-results) differ altogether (for completeness, in contrast to Cloud Computing the motivation of High-Performance Computing is to achieve the best execution time possible for a single job, regardless of cost or how specialized the hardware must be).

Bellow I will further elaborate on the characteristics of Cloud Computing and present the economic principles that support how cost can be lowered through this concept.

As a last general note, an instance of Cloud Computing is commonly referred to as a Cloud.

Characteristics. According to NIST's model, Cloud Computing must possess the following five characteristics:

On-demand self-service The user can provision its computational resources at any time (as needed), in a automatic way that doesn't require human interaction.

Broad network access The capabilities must be available through a network and accessible through standard mechanisms.

Resource pooling The computational resources must be shared across various users in a multi-tenant kind of way, with dynamic resource allocation as user load varies. The user must not be aware of the exact location of those resources, being a location change transparent for the user.

Rapid elasticity Capabilities must be able to be scaled (up or down) at any time.

Measured service Resources usage must be metered (quantifiable) in some way.

Models. Clouds are classed by their service model and by their deployment models.

The Service Model determines the way (level of abstraction) in which the shared resources are made available. When available as the flat-out computational resources they are, we have an Infrastructure as-a-Service (IaaS). When the resources are abstracted and transparently managed by a higher-level programmable computing platform, such as Google App Engine[3], we have a Platform as-a-Service (PaaS). Lastly, when the resources are presented as an end-application, we have a Software as-a-Service (SaaS).

When it comes to deployment models we can have Private Clouds, Community Clouds, Public Clouds and Hybrid Cloud. A Private Cloud is one whose infrastructure is meant for exclusive use of a single organization (that may have multiple users within that organization). However, this does not make any assumption as to the premises in which the infrastructure has to be placed (it may be on-site or off-site). Similarly, there are no assumptions as to who owns, manages or operates the cloud infrastructure (it may be the organization which has exclusive use of that cloud, a third-party or a combination of both).

A Community Cloud is very much like a Private Cloud, only that instead of being exclusive to the use of a single organization it becomes exclusive to a number of organizations that share a community.

On the other hand, a Public Cloud is one whose infrastructure is open to be used by anyone. It is owned, managed and operated by a third-party. Likewise, it exists in the premises of a third-party.

[3] `https://cloud.google.com/appengine/docs/whatisgoogleappengine`

Hybrid Clouds are infrastructures in which it is clearly identifiable (as separated units) multiple cloud deployment-models (Public, Community or Private), but are (somehow) bound together by a technology which allows the migration of data (or applications) between the separate deployment-models.

Economics. The two key principles that give an economical advantage to Cloud Computing are "Utility Pricing" and the "Benefits of Common Infrastructure".

Because costumer's demand often exhibits a spiky behavior through time, having the ability to pay for extra resources only during spikes (rather than owning them at all times) is cheaper when the Utility Premium is lower than the ratio of Peak Demand to Average Demand.

On the other hand, the monetization of the infrastructure is directly tied to its occupation. When you have specialized resources, these can only be employed in those specific tasks, which increases the risk of wastage in the form of idle capacity. Furthermore, specialized resources may also have personal dedicated to that exclusive task and (should equipments need to be repaired or replaced) they also bring other requirements and constraints, which further increases the monetary penalty of this wastage. Cloud Computing solves this problem by providing a common infrastructure that, because it is not tied to any specific task, can maximize the utilization of idle resources through sharing across all tasks. The personal that operates the infrastructure needs less knowledge of the specific task and the general purpose equipments are easier (and cheaper) to replace (for instance by taking advantage of Commercial off-the-shelf (COTS)). Outsourcing also becomes an option since the knowledge required is not specific to the operation but rather related to some general task to which there may already be solutions/services readily available in the market.

CHAPTER 3

State of the Art

In this chapter I will present a literature review and research the current state of technology, standardization efforts, tools capabilities and other efforts developed by third-parties which are within the topic of this dissertation.

I will start by the standardization efforts relevant to the "big-picture" vHGW and then present other vCPE research efforts. Then I will evaluate the Off-The-Shelf capabilities of the currently available tools, followed by the introduction of the prior work developed within Instituto de Telecomunicações to support the vHGW requirements within OpenStack and OpenDaylight.

3.1 Standardization efforts

3.1.1 ETSI NFV / Architecture

In an effort to standardize future work in the NFV field, European Telecommunications Standards Institute (ETSI) created a high-level vision of an architectural framework and design philosophy of the virtual functions [4]. This architecture follows the efforts made in the form of an introductory white-paper [5] and a technical report detailing the use-cases [6].

This vision has three distinct domains, the VNF domain, the infrastructure domain and the management and orchestration domain. The VNF domain relates to how the virtual functions are achieved, so that they can run on the infrastructure on-demand. The infrastructure domain (NFVI), is the one where the actual hardware resides and the virtualization of computational resources happens. Finally, the management and orchestration domain, which is the one that orchestrates and manages the life-cycle of both VNFs and infrastructure resources allotments.

Starting from this vision (figure 3.1), ETSI ends-up proposing a NFV Reference Architectural Framework (figure 3.2), which details the most relevant relationships between the VNFs/infrastructure and the Management and Orchestration plane, with a better insight into what elements make that plane.

A more detailed break-down of this architecture and its interactions can be found both at the original publication [4] and at the Cloud4NFV paper [7].

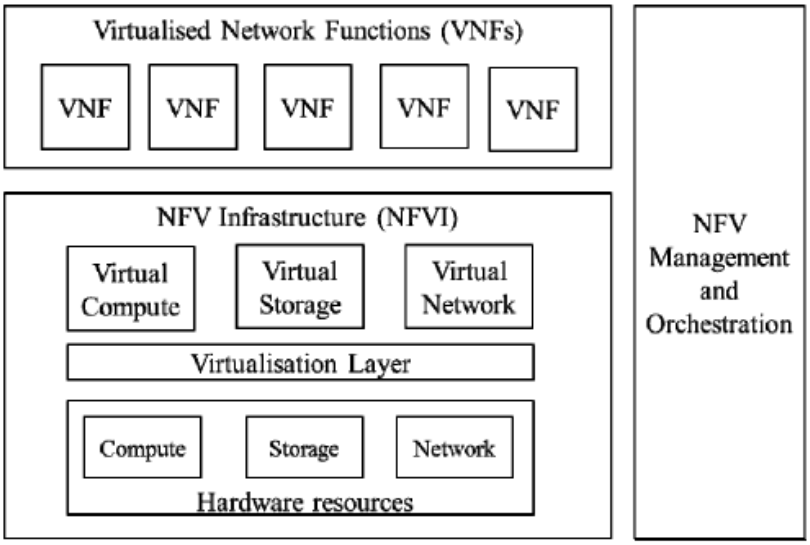

Figure 3.1: The high-level vision of ETSI NFV framework.

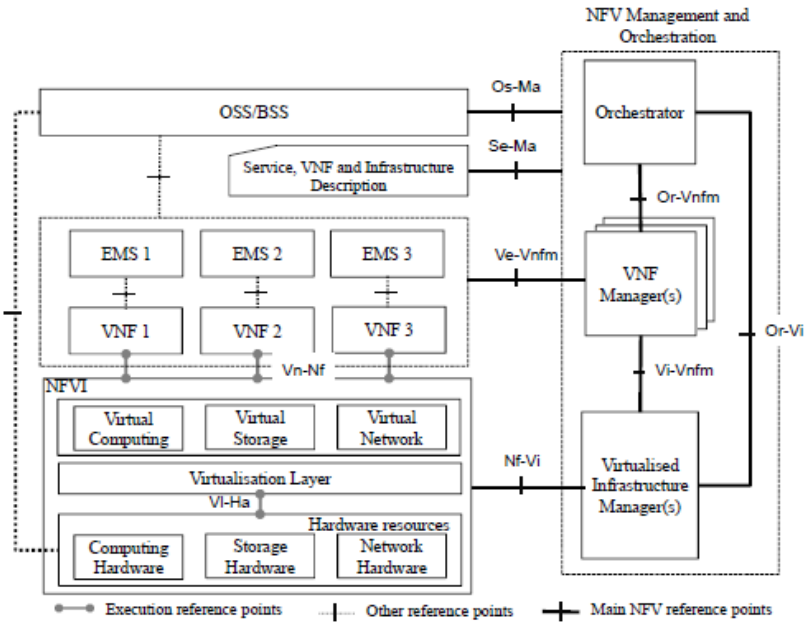

Figure 3.2: ETSI NFV Reference Architectural Framework.

3.1.2 IETF SFC ARCHITECTURE

The Internet Engineering Task Force (IETF) SFC architecture (RFC 7665 [8]) defines a standardized SFC architecture which is independent from the underlying forwarding topology.

Packet classification is done on ingress for subsequent processing by the designated Service Functions (SFs). Upon classification packets are then forwarded (in proper order) through the functions that make the SFC. Packets may be reclassified upon being processed.

The concepts of Service Function Path (SFP) and Rendered Service Path (RSP) are introduced, being the SFP the desired chain of services one wishes to be traversed (multiple instances may be available to provide the same function) and the RSP is the final realization of which instance of each function is designated to be traversed to build that chain.

The Service Function Forwarder (SFF) is in charge of forwarding the packets to the right function at all times (or to return the packet to the proper classifier).

There is metadata enclosed into each packet flow which gives context to the functions (and the SFF). Context may be just chain hop status or can be additional parameters/output of the SFs.

Because not all SFs may be metadata-enabled, the concept of a SFC Proxy was also introduced. This agent will decapsulate the metadata from the packets and deliver them to the functions. Once the functions finishes processing the packet, the proxy may either reencapsulate with a locally configured/generated metadata or return the packet to the classifier for new metadata generation.

Finally, the concept of symmetric and non-symmetric chains was also introduced. Symmetric chains are the ones which may have to be traversed in reverse order under given circumstances (for instance different flow direction, outbound vs inbound).

3.1.3 IETF NETWORK SERVICE HEADER

The IETF Network Service Header (NSH) [9] specifies an encapsulation header for the metadata (SFF control and SFs context) which builds upon the logic presented in the IETF SFC architecture (3.1.2). However, neither NSH or the IETF SFC architecture imply the necessary use of each other.

There are two types of header, MD-type1 which is of fixed size and MD-type2 which allows for a variable number of context headers (which may, by itself, also be of variable size). Both headers share the same Base Header (4-bytes) and Service Path Header (4-bytes).

The Base Header contains a two-bits field with the version of the NSH header (for future backwards compatibility), followed by another two bits for control (identify OAM and critical metadata), 6 reserved bits, 1 byte to select MD type, and finally another byte with the Next Protocol (their term for the original protocol before packet encapsulation).

The Service Path Header contains a 24-bit identifier (that selects the SFP), followed by an 8-bit Service Index which keeps track of the current hop within the SFP/RSP. It is mandated that the first classifier sets the index to 255, being then decremented as it hops through SFs. The responsibility for this decrement is of the SFs themselves (or the Proxy that handles the encapsulation on their behalf). The SFF may perform loop-detection through evaluation of this field in conjunction with the SFP ID, however this is not mandatory.

When the header is MD-type1, the Service Path Header is followed by 4x 4-bytes Mandatory Context Headers. The contents of this context headers are not processed by the SFF being these only relevant to the SFs (or enhanced classifiers).

If the header is MD-type2, the Service Path Header is followed by the optional Variable Length Context Headers (there may be none). The number os optional headers that follow is specified using the length field of the base header (if length is 2, then we have no optional headers). In turn, each variable context header has a new header of its own, which is used to specify the length of that context metadata, the TLV class, type within that class and the critical bit.

3.1.4 CABLELABS

CableLabs is a non-profit research and development consortium founded (and funded) by cable TV operators. Incidentally their research focus on cable TV networks and the challenges of cable TV

operators.

Amongst their research, they are also active in bringing SDN/NFV to cable TV networks, having released a thorough technical report [10]. In this report they cover the virtualization of the Converged Cable Access Platform (CCAP) itself, abstractions for CCAP management under SDN/NFV, topics such as Lawful-Interception, IPTV and "Bring You Own Cable Modem", SFC already using NSH, Intent-Based Networking along with various DOCSIS specific analysis and YANG data models that support the use-cases functionality.

They are actively contributing and incorporating Open Source developments, highlighting technologies such as OpenStack and OpenDaylight (ODL). In this latest report it is noted a contribution for ODL to which they added a PacketCable MultiMedia plug-in, which allows the ODL controller to communicate with an existing CMTS platform using the COPS interface and PCMM data protocol.

Despite the full-on focus on cable TV networks, their research and contributions should not be discarded even in other contexts, being a worthy mention in the context of the "big-picture" HGW (particularly in a convergence scenario with other access networks, there is already this pretty exhaustive research on the topic of cable networks).

3.2 OTHER EFFORTS

In this section I will approach third-party research and industry efforts in this field.

3.2.1 PROOFS-OF-CONCEPT

ETSI PoCs. ETSI has its own set of sanctioned PoCs within the context of "Hot Topics", which are identified, documented and consolidated by the ISG NFV Working Groups[1]. It is in the first completed "Hot Topic", HT01 – Use of SDN in an NFV architectural framework[2], that we can find the Service Chaining for NW Function Selection in Carrier Networks [11] PoC, which addresses many topics of the environment we wish to build to demonstrate our policy-driven vCPE.

In this PoC NTT Corporation used VNFs from providers such as Cisco (CSR1000v[3] for its DPI capabilities), Hewlett-Packard (VSR1000[4] as the vCPE) and Juniper Networks (FireFly[5] as the Firewall). Together this functions were used to successfully demonstrate the feasibility and value-added of SFC in a set of Use Cases (mostly blocks/filtering of services) that focused in the residential HGW.

This PoC has particular significance given its closeness with the objectives and target use-cases of the first phase of the "big-picture" vHGW. Notable differences are the fact it uses proprietary VNFs and their steering strategy used the MPLS tag. Although MPLS is very common in operator networks,

[1] `http://www.etsi.org/technologies-clusters/technologies/nfv`

[2] `http://nfvwiki.etsi.org/index.php?title=HT01_-_Use_of_SDN_in_an_NFV_architectural_framework`

[3] `http://www.cisco.com/c/en/us/products/routers/cloud-services-router-1000v-series/index.html`

[4] `http://www8.hp.com/us/en/products/networking-routers/product-detail.html?oid=5443163`

[5] `http://www.juniper.net/sites/us/en/products-services/security/firefly-perimeter/index.page`

it is not so prevalent within Cloud-Computing datacenter networks. SDN is the bread and butter of Cloud-Computing networks, therefore a steering solution that used OpenFlow would be much more compatible with existing processes.

Cablelabs vCCAP - vCPE. The Cablelabs vCCAP demonstrator [12] is an early PoC for the ongoing architectural work made by Cablelabs to modernize (and standardize NFV/SDN) in cable operator's networks.

In this demonstrator they have used a Raspberry Pi (v1 Model B) to act as the vCPE but were still expecting their CPE to run locally the same network functions (only enhanced by other functions running in the cloud). It features SFC and many of the Open Source technologies such as OpenStack as the cloud platform and OpenDaylight as the SDN controller.

It is a widely used reference of CPE virtualization in North-America, probably due to the prevalence of cable TV networks in that market. Given the fact the Instituto de Telecomunicações PoCs have also used a Raspberry Pi (v1 Model B) as the CPE, there may be a strong temptation to draw parallels with this demonstrator. However, unlike this effort, our Raspberry only handles the tunnel to the cloud and bridges the WiFi adapter with the virtual network. There are no local functions being run, this due to a "big-picture" design choice to simplify the CPE to its bare minimum in an effort to minimize the need for truck-rolls to solve problems in the customers home.

3.2.2 ACTIVE RESEARCH

The field is being actively researched with numerous contributions being frequently published, however I would like to highlight the following articles due to their relevance to some key components of this dissertation (or how they offer an alternative to the approach taken):

A "Tethered Linux CPE for IP Service Delivery" [1], which brings better insight into how a large operator such as Sky views the CPE, it's expected life-cycle alongside with a proposition to use Linux BPF to run virtualized functions on the CPE itself. However, despite BPF holding great promise performance-wise, it may also reduce the agility and time-to-market of new functions – after all, they will have to be developed with the same mindset as a Linux kernel program.

An article which presents performance evaluation of multi-tenant virtual networks in OpenStack [13], which helps to understand how this system may scale (both for phase 1 and phase 2) and anticipate some results in phase 2, as the authors test scenario (like ours) also used a DPI based out of nDPI. However, because they used physical interfaces to make their tests, the total bandwidth value in the case that most resembles the phase 2 Classifier bandwidth measurements is capped to 1 Gbps (the interface limit). Nevertheless, before hitting the interface limit, their results were consistent with the ones achieved in this dissertation (by default, very similar load among tenants).

Still on the topic of bandwidth evaluation, there is the article OpenANFV [14] which presents a way to bring FPGA acceleration to OpenStack (NAT) and the functions such as the DPI. Their solution was shown to outperform the non-accelerated test by quite a large margin (lowest was 8.2x).

Moving to SFC, "Dynamic Chaining of Virtual Network Functions in Cloud-Based Edge Networks" [15], presents a feasibility study with Mininet and POX as SDN controller of the "big-picture" scenarios which will determine the course of this dissertation.

Still on SFC and the SDN controller, given our steering solution uses Instituto de Telecomunicações extensions to ODL, the OpenNF [16] shows an optimization study of how to improve these capabilities

in the SDN controller.

In the spirit of future work and improving the "big-picture", the article "SDN controller for Context-aware Data Delivery in Dynamic Service Chaining" [17] explores NGSONs to enhance SFC by introducing a form of context-awareness, which may allow for better scalability of the cloud network and/or better monetization of the network.

3.3 OFF-THE-SHELF CAPABILITIES

OpenStack. OpenStack is the largest (most widely used) free and Open Source Software for an IaaS Cloud Computing platform. It is organized into many sub-projects which target specific areas of the platform (for instance Nova for Computing, Cinder for Block Storage, Neutron for Networking, and so on), being Neutron the most important one for this dissertation's work. Its open-stance to all technologies has gained a lot of traction within the industry, with many of the big companies in the arena (such as Cisco, Google, IBM, Intel, Hewlett-Packard and others) backing the project and even companies with competing solutions are contributing for the project (such as VMware, Oracle, Citrix, amongst others), which makes OpenStack close to a "de facto" standard for Cloud Computing.

Configuration can be done through REST APIs (separated for each sub-project), command line utilities (Python clients, also one for each sub-project) or through a web interface (which is the Horizon and covers – most of – all other projects).

OpenStack also brings the required building blocks within the ETSI's NVFI (Infrastructure) block, such as the Virtualisation Layer, the Virtual Compute pool (Nova), Virtual Storage Pool (Cinder/Swift) and Virtual Network (Neutron). It may also go further and provide management and orchestration to the infrastructure (through Heat) and means to build some OSS logic (with Ceilometer).

It already integrates with the ODL SDN controller and, on top of previous contributions (3.4.3), it has efforts within the Group Based-Policy[6] subproject to bring SFC to OpenStack. Nevertheless, SFC isn't yet a part of any main-line release of OpenStack, as it is still being developed (for official release).

Another recent effort within the vCPE context is Tacker[7], which brings an NFV Orchestrator with a built-in VNF manager. It is based of ETSI MANO architectural framework, aiming to bring into OpenStack the capability to orchestrate the VNFs end-to-end (but still in very early stages of development).

Development follows a 6-month release cycle, with a dedicated team for (faster) security fixes within the last 2 releases and End-of-Life support that spans for 4-years.

Instituto de Telecomunicações is a contributing partner to the OpenStack project.

OpenDaylight. ODL is an Open Source SDN controller, which (starting with Helium) has some support for SFC using IETF NSH (3.1.3). The process still isn't quite production ready, as it has a few omissions (such as the NSH proxy, the need to implement a classifier and the lack of header decapsulation – due to an uncertainty within the project as to which entity should have that role, the last SFF or the classifier).

It integrates with Open vSwitch (OVS) through the southbound API OVSDB, being OVS the preferred switch of the project (mostly because it is also Open Source Software).

[6] `https://wiki.openstack.org/wiki/GroupBasedPolicy`

[7] `https://wiki.openstack.org/wiki/Tacker`

OpenStack can integrate nicely with this SDN controller, however it will not (yet) take advantage of the newly released SFC capabilities.

Open vSwitch. Open vSwitch is an OpenFlow compatible virtual switch with a southbound API readily available in ODL (OVSDB). It is capable of performing SFC through the use of OpenFlow commands and configuration through OVSDB, however this feature (for the purposes of a vHGW) is not neatly packaged to provide a seamless configuration of steering between VNFs (it is a very manual process).

OpenStack not only integrates nicely with OVS but this is in fact Neutron's default.

3.4 PRIOR WORK

In this subsection I will describe the work carried-out within Instituto de Telecomunicação in the vCPE field. These efforts will be the starting-grounds for the construction of the vHGW, which is the "big-picture" for this dissertation's work.

3.4.1 CLOUD4NFV

Cloud4NFV [7] is a ground-work carried-out by PT Inovação alongside Instituto de Telecomunicações, which aimed to specify a cloud environment that followed NFV standard guidelines for cloud infrastructure management and SDN platforms, with the ultimate goal of demonstrating its viability in the context of virtualizing CPE functions.

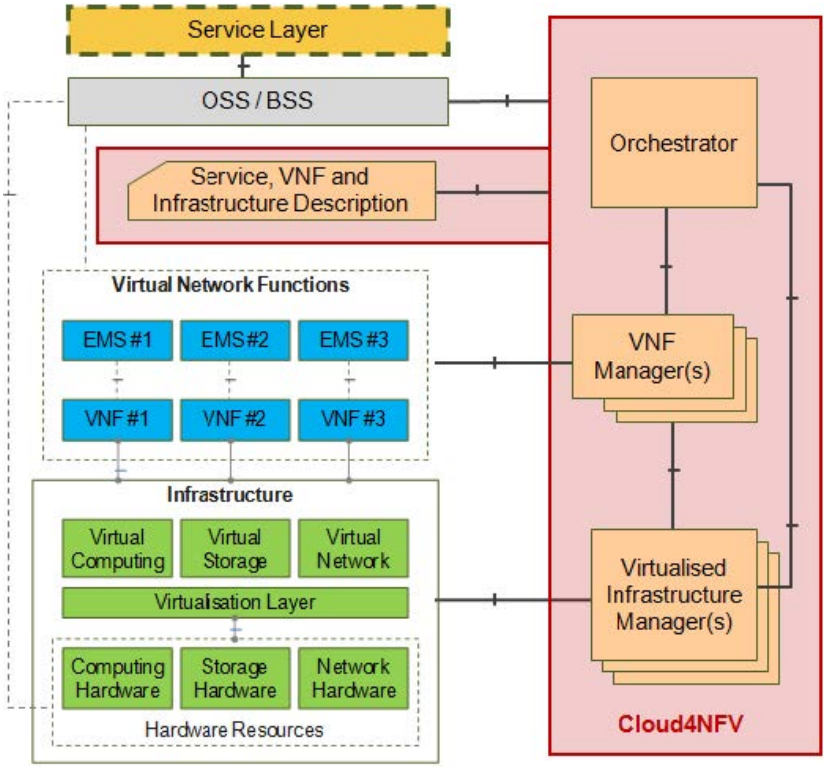

Figure 3.3: Cloud4NFV reference architecture, adapted from ETSI (3.1.1)

The focus was on the Orchestration and Management side, with development made in those areas. A PoC was conceived to support the envisioned architecture, which showed the specified environment was viable.

Since this was of an initial effort, the PoC did lack proper ways of connecting the physical CPE that would act as a vHGW as well as a formal way to connect external (physical) devices to the virtual network. It also didn't have Traffic Steering capabilities to perform SFC and was short on VNFs.

This work's findings and development provides a ground-foundation to the architecture that will be worked upon in this dissertation.

3.4.2 NEUTRON ATTACHMENT POINTS AND EXTERNAL PORTS EXTENSION

Developed for OpenStack's Icehouse release, this Neutron extension brings the capability of extending a virtual network to the outside of the virtual environment, allowing for actual (physical) devices to be connected within that virtual network.

The Attachment Point is the endpoint to which a (GRE) tunnel can connect within a Neutron network. Neutron is therefore the endpoint in the cloud environment Point-of-Presence (PoP), being the other endpoint the CPE which bridges the virtual network to the local switch of the costumer's home network (the pHGW). The External Ports are the representation of each device that connects to the virtual network through an Attachment Point.

It is a result of Igor Cardoso's MSc dissertation [2] and is a crucial building block of the "big-picture" vHGW system to which this dissertation's work will contribute.

3.4.3 NEUTRON TRAFFIC STEERING EXTENSION

Also developed for OpenStack's Icehouse release (in conjunction with the SDN controller Opendaylight Hydrogen, OVS/OVSDB), this Neutron extension brings Traffic Steering and SFC capabilities to Neutron.

Traffic selection/classification is done using the 5-tuple (with the later addition of ToS/DSCP bits), which is a sub-set of Openflow's 12-tuple. Its fields are: IP source; IP destination; IP protocol; TCP/UDP source ports; TCP/UDP destination ports;

These classifiers are configured using the Neutron Client with the following new commands:

steering-classifier-create Create a traffic steering classifier.

steering-classifier-delete Delete a given classifier

steering-classifier-list List traffic steering classifiers that belong to a given tenant.

steering-classifier-show Show information of a given classifier.

steering-classifier-update Update a given classifier.

SFC is built through the construction of port-chains, which are a form of switch-level chaining that takes a flow selected by a classifier and makes it "hop" port-by-port through a configured sequence of switch-ports. The caveat with this form of chaining is that, in order to get that steered traffic processed by generic functions (connected to each of those network ports), the destination MAC address must be changed (at each hop) to the one that corresponds to the function. The new commands that bring this functionality to Neutron, with automatic change of the destination MAC address, are:

port-chain-create Create a port chain.

port-chain-delete Delete a port chain.

port-chain-list List port chains that belong to a given tenant.

port-chain-show Show information of a given port chain.

port-chain-update Update a port chain.

It is the work of Carlos Gonçalves along with João Soares and later maintained by Igor Cardoso.

3.4.4 HORIZON EXTENSION

Presented by Mario Car as a result of his MSc dissertation [18], this Horizon extension brings a GUI to perform the tasks of the previous two Neutron extensions (3.4.2, 3.4.3) in a easier and more user friendly fashion.

At the top of fig. 3.4 we can see the graphical way to create a SFC with regular Virtual Machines (VMs). Traffic Classifiers do need to be created before the SFC though, but they can also be created through Horizon (in a dialog like the one at the bottom-center). At the left we can see all existing SFCs, regardless if they are created through this UI or through the CLI. Finally, at the right we see the new ability of visualizing the attachment points through the network topology.

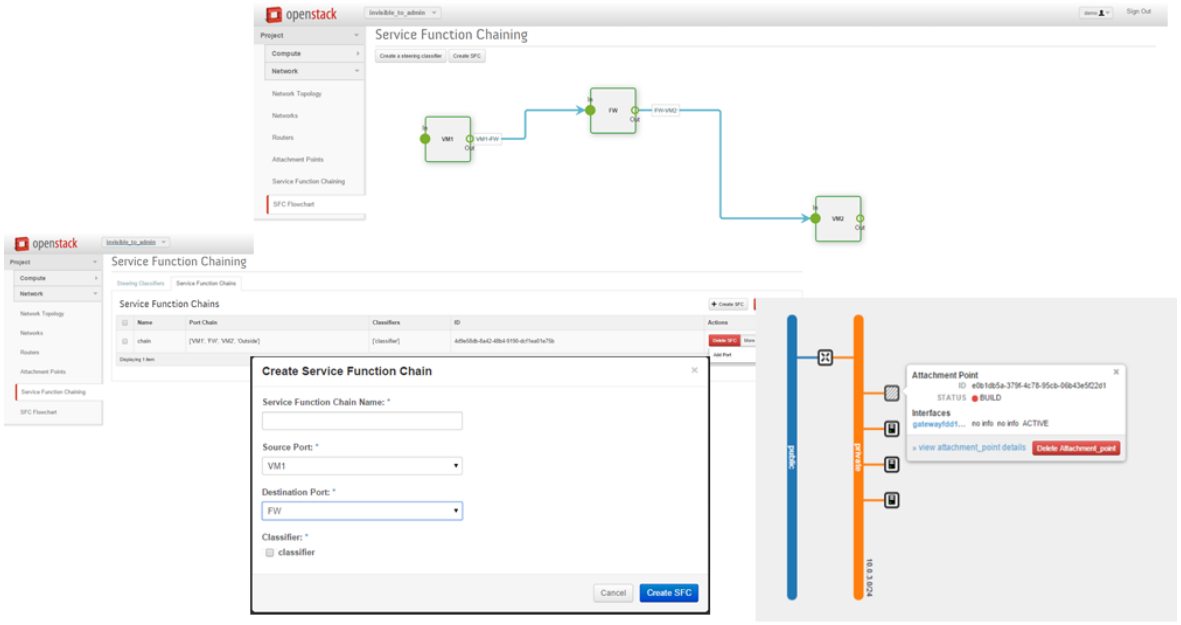

Figure 3.4: An assortment of screenshots (by Mario Car) that showcase the capabilities of his Horizon extension.

CHAPTER 4

Design and Specification (Phase 1/vDHCP)

This section describes the first phase of the work carried-out by this dissertation, the design and specification of the modified vDHCP VNF. This function is meant to be a drop-in replacement of OpenStack's existing vDHCP server, with new functionality for external devices (the automatic Neutron port creation and AAA authorization / accounting events notification). The needs and the rationales behind it come from the requirements of the vHGW system and its Phase 1 PoC (the "big-picture").

I will first start by describing the "big-picture" (the vHGW), followed by the vDHCP function. Lastly I will approach other functions (provided by third-parties) which are relevant to the PoC and the results.

4.1 big picture: the vHGW phase 1

The first phase of the vHGW starts by putting together all prior work (3.4), laying from there a development path to achieve a working architecture for a HGW with dynamic SFC which can be ultimately demonstrated in a live PoC of the system. I will start by presenting the requirements and goals set in this first phase and then move to the architecture of the solution.

4.1.1 requirements and goals

Overview. Having the ETSI NTT Corporation PoC [11] in mind, the main objective for Phase 1 is to ultimately achieve a working PoC with similar capabilites, but without relying on proprietary functions and with real devices being able to connect to the system during the PoC. In order to achieve that, the following points must be met:

1. Devices must be able to connect to the pHGW at any time and acquire basic network connectivity (local IP) within the virtual network.
2. Devices must be able to connect to the Internet through the vHGW (Routing + NAT).
3. There must be a way to configure the Traffic Steering (and build SFCs).
4. We need some VNFs to build Service Function Chains and demonstrate its proper operation.

In order to achieve the first goal (which will be the focus of this dissertation), we need a functioning tunnel that connects the pHGW to the virtual network (provided by 3.4.2) and a way to automatically create external ports (this dissertation's goal for this phase). The second requirement is instantly solved by OpenStack upon achieving the first one.

The third is provided by the prior work (3.4). This also means that, unlike the NTT PoC which tentatively used the MPLS tag as the chain classifier, the steering mechanism uses a 5-tuple subset of OpenFlow's 12-tuple (like described in 3.4.3) to perform dynamic SFC. Although MPLS is quite common in operator's networks (and has wide support by operator's equipment), it is not so widely adopted for Cloud Computing and its datacenters. However, SDN and OpenFlow are widely prevalent in cloud networks. Therefore, a Steering solution that is based-upon OpenFlow is actually more cloud-friendly than a MPLS one.

Lastly, the missing VNFs (for this phase) will be provided by a 3rd-party (Miguel Dias).

Caveats. It is also set as a requirement that external devices have some form of session control (through an external AAA, that communicates through the RADIUS protocol). This must give the external AAA the ability to allow (or deny) the device's connection and track which devices are connected to a given network. Additional session information may also be supplied to the AAA (such as the pHGW identification/wireless SSID/...), which allows for future scenarios in which a virtual network can be accessed through multiple geographical locations and/or other means (such as LTE/3G). Not less important, this AAA may also be used by an external orchestrator (work of Bruno Parreira) to determine if any action must be taken to accommodate the new devices.

It was explicitly specified that, in order to be compliant with PT Inovação vCPE efforts, this session control must be implemented within the DHCP server (more on details will follow in 4.2.1).

4.1.2 ARCHITECTURE OVERVIEW

The architecture for this phase (fig. 4.1) only contemplates a single datacenter/PoP which will run the NVFI with all VNFs. The pHGW with WiFi is, at this point, a standard HGW enriched with the capability to establish a GRE tunnel to the datacenter/PoP, but with all its local functions turned-off. The access network for the pHGW is, for the Instituto de Telecomunicações efforts, its own internal ethernet network (which is a shared network, not a dedicated access). On the other hand, the PT Inovação version uses a GPON network being the pHGW also an ONT.

The role of the datacenter/PoP will be played by a single machine (nicknamed Alex), which has connectivity to both the Instituto de Telecomunicações and PT Inovação networks. More details about the machine will follow in the results section. (5.3)

The external AAA will do just registration and validation of the vDHCP's RADIUS implementation at this point.

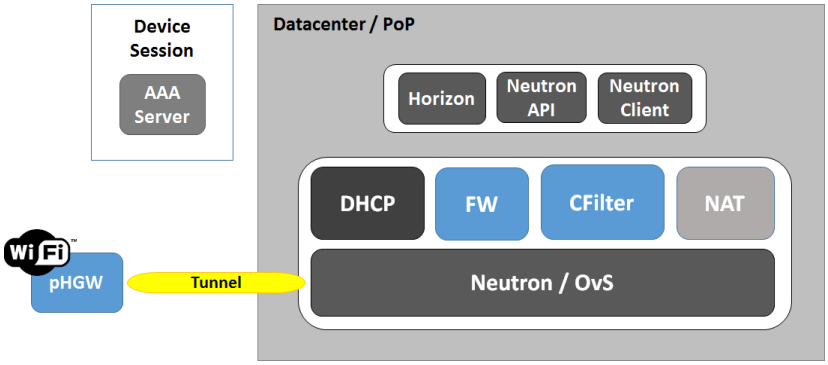

Figure 4.1: Phase 1 high-level architecture.

The PT Inovação vCPE version also adds an orchestrator and VNF manager (developed by Bruno Parreira), a costumer's portal (developed by Miguel Dias) and an own Device AAA (by Paulo Rolo), which are ultimately omitted from this dissertation results (as this "big-picture" changes have no influence on the ability to present this dissertation's work).

4.2 THE vDHCP FUNCTION

Now I will scale-back and focus on this dissertation's actual work, the vDHCP function. I will start by consolidating the requirements of this function, followed by a specification of the solution to be implemented.

4.2.1 REQUIREMENTS

Having in mind the "big-picture" requirements, we can break-down the vDHCP course of work into three action points:

- Provide an automatic way to create external (Neutron) ports upon the connection of a physical device.
- Implement a session control mechanism for those devices, through the DHCP protocol, that interacts with an external AAA entity.
- Integrate neatly with the OpenStack Icehouse environment that has been used for previous development.

With the following additional constraints:

- The vDHCP function must be an extension of the DHCP server that ships by default with Neutron (dnsmasq).
- Both session control and external port creation must be achieved through this vDHCP.
- The corresponding vHGW identifier (its MAC address) must be sent in a RADIUS field for each device session message.

This means we need dnsmasq to be able to trigger an external entity (or perform itself) the Neutron port creation/AAA notifications at specific stages of the DHCP protocol, particularly upon DHCP-DISCOVER. The reason for this is the way Neutron manages the vDHCP pool, in which the only leases that exist are within Neutron ports. Therefore, a newly connecting device which still requires the creation of the Neutron port will never get a DHCP-OFFER from the vDHCP (to which the client would reply with a DHCP-REQUEST), making the discover phase the only reliable way to detect a new device for port creation.

4.2.2 DESIGN

In order to design a solution that achieves the above requirements with a drop-in substitute of Neutron's vDHCP, one must first identify the components that are at play and how they interact with each other. In a stock Neutron installation, the built-in DHCP server is a part of the DHCP Agent, which is by itself commanded according to Neutron Server's configuration (figure 4.2). Neutron Server can be configured (directly) through a REST API or through its command-line Neutron Client (that also uses the API). All configuration is persisted in the Neutron DB, which is shared across major components.

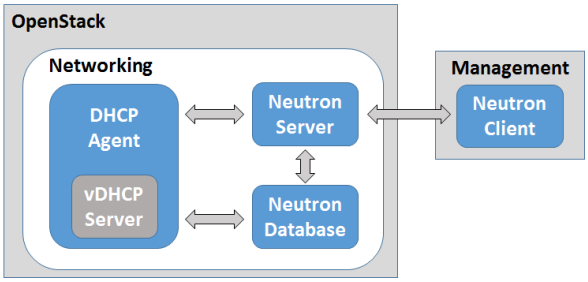

Figure 4.2: Identifying the components at play to implement the changes to stock vDHCP.

So, to preserve the working model of OpenStack Neutron, it becomes clear that not only the vDHCP server itself must be modified but also Neutron Server's API, Neutron's DB and also (for convenience and elegance) Neutron Client. A new interaction must be added for the external Device's AAA and existing interactions are changed through the enrichment of their APIs (figure 4.3).

Because the DHCP Server is a part of DHCP Agent an implementation choice arises as to how will it communicate with the Neutron DB. One course of action is to modify the DHCP Agent to support the new interactions and create a new internal API which allowed the Agent to communicate the new data with the DHCP Server. Another course of action is, since in the broader view both are actually viewed (to the outside) as the same thing (DHCP Agent), the DHCP Server could just connect directly to the Neutron DB (seen as DHCP Agent) and retrieve its configuration data.

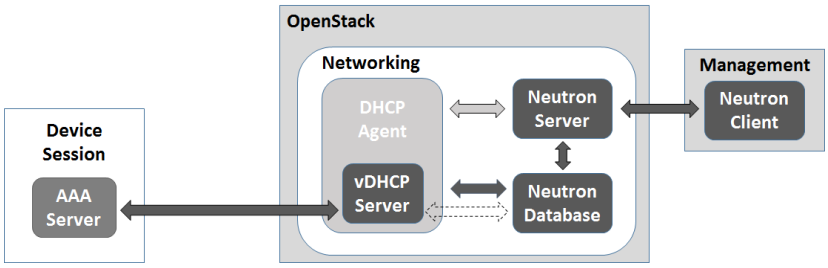

Figure 4.3: vDHCP solution design.

Therefore, the following changes must be made to achieve the solution:

- vDHCP Server:

  AAA Module Allows the vDHCP to communicate with the AAA (via RADIUS) and perform all the required device session operations.

  Neutron DB Module Allows the vDHCP to query Neutron's internal state and retrieve the required association data along with additional fields (such as vHGW identifier) which are needed for the AAA module.

  Change DHCP State-Machine Because dnsmasq's scripting capabilities are limited (does not support DHCP-DISCOVER, only lease handling), this is needed to perform all specified interactions at the right stage of the DHCP protocol.

- Other Neutron Extension(s) for neat integration with OpenStack:

  Extend Neutron Server's API Gives support to the new operations required to associate a network with a vHGW.

  Extend Neutron DB Creates data-structures (tables) that allows to store the new associations and fields that support the new operations.

  Extend Neutron Client Allows the invocation of the new methods through Neutron's command-line interface (and other Python clients that use Neutron Client's API call methods).

4.2.3 SPECIFICATION

In this subsection I will describe the sequence and messages required in order to achieve the vDHCP functionality. I will start by the device connection case (4.4).

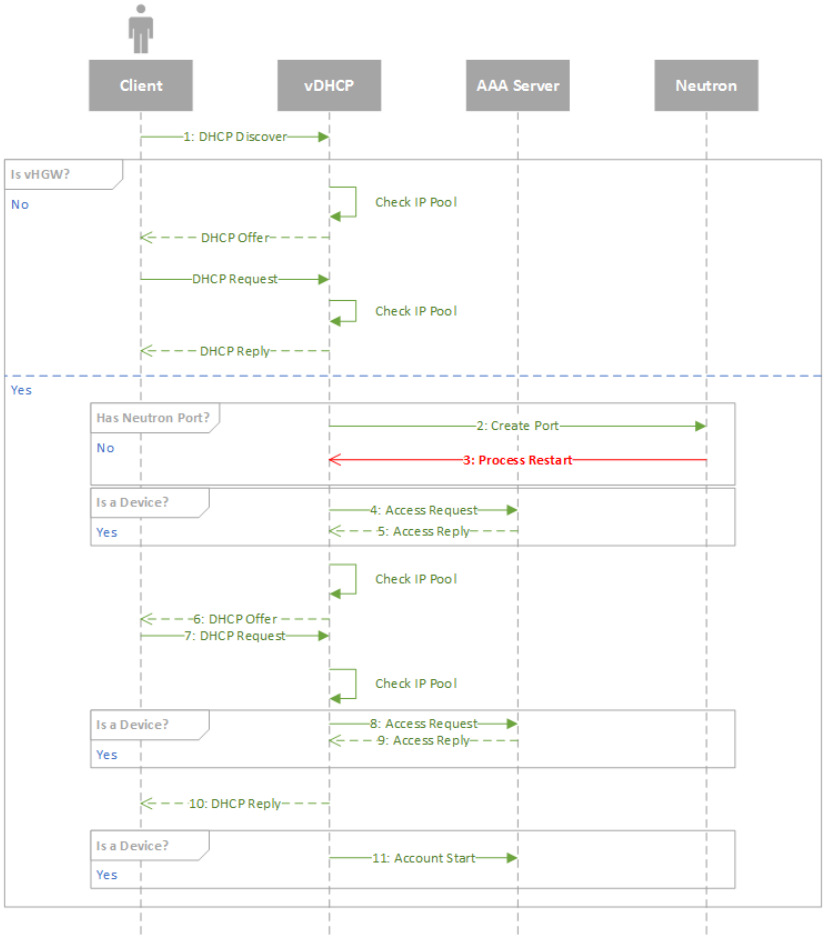

Figure 4.4: vDHCP sequence diagram upon Device connection.

Like any other DHCP server, it all starts when the device sends a DHCP-DISCOVER message (1). Upon reception by the vDHCP, the very first thing required is to check if the network in which the server is running belongs to a vHGW. If it does not, then no additional checks or computations should be done, the vDHCP server will behave (for all purposes) like the unaltered version shipped with OpenStack. However, if the network does belong to a vHGW, then we must first check if the (possible) device already has a Neutron port. If a port already exists then no action is required at this moment, but it should be noted this may not be a message from a device (for instance, it can be another VNF in the network). If the port does not exist, then we have an external device connecting for the first time, so a new Neutron port will be created (2) with the respective MAC address and hostname (if available).

Neutron's default behavior mandates that, upon creating a new Neutron port, a RPC will be sent to the DHCP Agent which will trigger a restart (3) of the vDHCP VNF. This also means the device will have to repeat the DHCP-DISCOVER message (1) in order to resume the sequence. The reason why Neutron has this behavior is due to the fact IP Address Management (IPAM) is not performed by the vDHCP but rather Neutron itself. A new IP address is automatically assigned to the port upon creation and recorded in the Neutron DB. Then, the way Neutron configures that new lease is by re-writing a flat file which has the MAC address(es) and their corresponding IP lease(s). This flat-file is passed as parameter during the boot of the vDHCP, therefore a restart signal (-HUP) must be sent in order to re-read that file and update the lease database.

Next is a check if the DHCP-DISCOVER is actually coming from a device. This check is required to prevent AAA interaction for DHCP messages that originate from VNFs with interfaces in that network. The device check is performed in one of two ways, if there is already a Neutron port created for that MAC address then the Neutron's "device owner" field will tell apart VNFs, routers and other OpenStack owned entities from external devices. On the other hand, if the message comes from a MAC address without a respective Neutron port then it is (by default) an external device.

Once the message is deemed to come from a device, a first Access Request (4) will be made to the AAA which will just check if the ability to connect new external devices is enabled for that pHGW. A single radius pair goes in that request, the NAS-IDENTIFIER which currently is the MAC address of the pHGW (not the device, we are validating an authorization of the pHGW).

The first Access Request (4) is being made after the port creation due to a "big-picture" design choice, in which it was deemed to be more valuable to keep a registry of every device that attempts to connect to the pHGW (to log device connection attempts/improve user experience by fronting the most expensive operation – port creation) over the ability to disable port creation through the AAA.

After the AAA is contacted with that Access Request (4) a Reply message (5) is sent to the vDHCP, which will either authorize the sequence to continue or cause it to exit here.

If access is granted, the vDHCP will search its lease database (the before mentioned flat-file) for the proper IP that matches the requesting MAC address. If the MAC address is not in that file then the sequence will stop here.

Then a DHCP-OFFER (6) is generated and sent to the client, with the IP address that was retrieved from that file. The client will then send a DHCP-REQUEST (7) with the IP address it got from the DHCP-OFFER (6). If it does not, then the next check of the IP poll will fail, a NACK will be generated and the client will not be able to go further in the sequence until it does send the proper IP address.

Once that happens, a second Access Request (8) will be made, this time to validate the ability of that device connecting to that pHGW. This request has more radius pairs, on top of the previous NAS-IDENTIFIER (which currently is the MAC address of the pHGW), it is also sent the device MAC address (in CALLER-STATION-ID), the device IP Address (in FRAMED-IP-ADDRESS) and the device hostname (if available, in FRAMED-CALLBACK-ID). Like before, a Reply message (9) is sent to the vDHCP.

If access is granted, the vDHCP will ACK (10) the DHCP-REQUEST (7), otherwise a NACK will be sent (10) and the sequence will be stopped.

Finally, the AAA is notified to start accounting for that device (11). On top of the radius pairs of the Access Request (8), it also follows the ACCT-STATUS-TYPE set to start (do note this message is of a different kind than the Access Request, only some pairs are common).

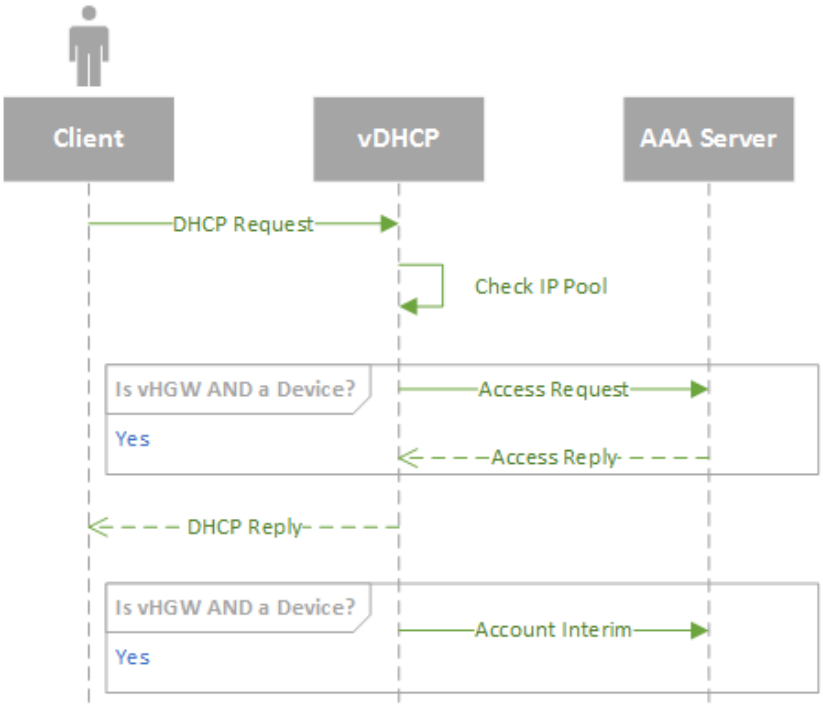

Figure 4.5: vDHCP Lease renew.

As for the lease renew process (figure 4.5), it follows a similar logic to the device connection diagram after the DHCP-REQUEST (7). The difference is the AAA will get an interim instead of an accounting start. This translates into the change of the pair ACCT-STATUS-TYPE from start to interim.

Because IPAM is separated from the vDHCP, this allows to use the lease time (and the renew process) as a means to keep status if the device is still connected to the pHGW or not. Because the vast majority of devices do not release upon disconnecting (read Windows does not), this sequence provides an alternative to work-around that. The precision of the status is directly tied to the lease time, a short lease time gives better precision (but more network overhead) while conversely a longer lease time gives worse precision but lower network overhead.

The release process (figure 4.6) is very much akin to the renew sequence, only with DHCP-REQUEST replaced by DHCP-RELEASE and the accounting message has the ACCT-STATUS-TYPE set to stop instead of interim.

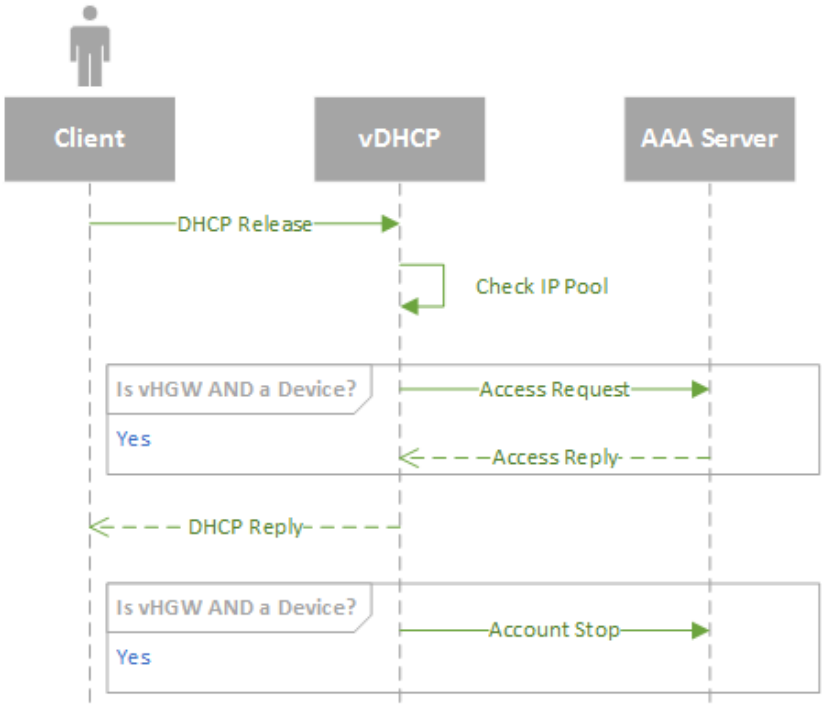

Figure 4.6: vDHCP Release.

4.3 THE OTHER VNFS

In this section I will describe the functions that, although not developed by this dissertation (but were rather by 3rd-parties), are important to mention due to their role in the PoC and their relevance to this dissertation's main topic.

NAT. The NAT function is natively provided by OpenStack and performs NAPT44 for all network traffic that traverses through it. It is a statefull function by nature, which requires that this function must always be traversed for all communications which require NAT, being traversed last when communicating from the virtual network to the outside (device upstream) and traversed first when communicating from the outside to the inside (device downstream).

OpenStack's implementation uses Linux Network Name Spaces[1] which employ the right netfilter rules to perform the function.

Firewall. The Firewall VNF was provided by Miguel Dias and is a regular VM which has an installation of the Alpine Firewall. The purpose of this firewall is to just perform blocks (drop traffic that matches the criteria), not having any kind of Statefull Packet Inspection enabled. As such, no additional care is required as to the placement of the function within a SFC or the need to cover both directions of same traffic (downstream and upstream).

The Firewall does need to be configured (it doesn't just block everything that reaches it), being that configuration performed through SSH (injecting regular iptables rules as needed).

[1] `http://docs.openstack.org/networking-guide/intro_network_namespaces.html`

URL Filter. The URL Filter VNF was provided by Miguel Dias and is a regular VM which has an installation of the DansGuardian content-filter. Because it has an internal HTTP proxy that will retrieve the requests to be filtered by the content filter (Squid proxy), this will create a new connection to fetch that content which will be then used to return to the network the filtered data, thus the 5-tuple that characterizes the flow going into this function will no longer be valid to select the corresponding processed flow that exits the function.

To mitigate this, this function has always been placed at the end of the SFC (the very last VNF), thus not requiring the selection of outgoing flows for further steering.

Much like the Firewall function, the URL Filter does need to be configured (it doesn't just block everything that reaches it). The configuration is also performed through SSH, by handling DansGuardian's configuration files and restarting its process.

CHAPTER 5

Implementation and Results (Phase 1/vDHCP)

I will now present the "big-picture" compromises, followed by the implementation rationales of the vDHCP and then proceed to the results.

5.1 big picture: vHGW general compromises

In order to produce the PoC within the desired time-frame, there was the need to perform some agreeable compromises, which were not handled at this moment.

VNFs Security The VNFs configuration is performed using the costumer's virtual network. Additional care is needed to disallow the use of configuration APIs directly from the costumer's devices/unauthorized agents in the network. An Access Control List (ACL) with the authorized IPs is a starting point, but not a full solution (we need to address the issue of IP Spoofing).

AAA Security As a consequence of the first, the AAA is actually reachable from a costumer device/unauthorized agent and additional steps must be taken to avoid the possibility of a device injecting unwanted (spoofed) RADIUS traffic to the network. The risk here is not only a breach of authenticity (should the attacker acquire the shared secret and be able to generate valid RADIUS messages) but also availability (a denial-of-service attack to either the AAA server or the DHCP server that acts as a AAA client).

IP/MAC/ARP Spoofing We lack a way to prevent a costumer's device from injecting network traffic with forged IPs/MACs (which allows for attacks such as ARP spoofing). Although this is also an issue in most other home networks, the new capabilities introduced with network virtualization (at a very elemental level one could have policy-based switching) should be used to preemptively mitigate the effects of these attacks.

Single Tenancy The VNFs do not currently support Multi-Tenants, which means they need to be instantiated per costumer's network (instead of being shared across many customers). This

translates into more running VMs, which adds an increased management complexity (more VMs to track in order to determine configuration addresses) and a higher virtualization overhead (since the hypervisor is KVM, this means a new OS must be running for each instantiated VNF). It does however provide better resilience against failure propagation across tenants, if a VNF crashes then it will only affect the functionality for that tenant.

VNFs Failure-Detection / High-Availability No considerations were made (so far) to implement a Fail-Over mechanism for High-Availability of the VNFs.

5.2 vDHCP implementation

The vDHCP is implemented by making use of the original source code of the *Dnsmasq*[1] lightweight DHCP server (that happens to be OpenStack Neutron's default DHCP server), through the addition of new C modules that are built, linked and used in the resulting (modified replacement) *Dnsmasq* binary. Each of the module's achieved functionality is described in the subsections that follow. All new modules were documented in *Doxygen*'s[2] format.

In order to ease the effort needed to reconfigure those modules, an INI file was created to facilitate the configuration of the general aspects. The INI parsing is done using the C library *inih*[3].

Additionally, there is also an OpenStack Neutron's extension to integrate (configure and enable) the new functionalities within a Neutron virtual network.

Other considerations. Given dnsmasq (and the new modules) are written in C, a non-memory safe language, additional care was taken to prevent use-after-free exploits (dereferenced pointers were always explicitly set to NULL after free) and there was also care to prevent buffer overflow exploits (through the use of memory functions which have a bounded number of memory positions set adequately to the valid size, for instance *snprintf*). The format string was also always specified, preventing format string attacks.

5.2.1 dnsmasq: neutron db handler

The Neutron DB Handler was made using the package *libmysqlclient-dev*[4] available in Ubuntu Trusty's distribution to interact with the Neutron DB alongside direct calls to Neutron CLI client. This module will perform automatic port creation when a new external device connects to the virtual network. The documented *.h* file of the actual implementation is available in the Appendix A.1, which helps to get a clearer developer-oriented vision.

The major methods to elaborate on are:

- Check if the virtual network belongs to a vHGW.

[1] `http://www.thekelleys.org.uk/dnsmasq/doc.html`

[2] `http://www.stack.nl/~dimitri/doxygen/`

[3] `https://github.com/benhoyt/inih`

[4] `https://web.archive.org/web/20150512180357/http://packages.ubuntu.com/trusty/libmysqlclient-dev`

- Check if the MAC address belongs to an external device.
- Create the actual Neutron port.
- Update a Neutron port.

Given the vDHCP also performs session control with the DHCP protocol (through the module that follows in the next subsection), it is required to provide a way outside of the port creation method to identify if the network belongs to a vHGW and then if the MAC address is of an external device.

The first practical requirement comes from the fact this vDHCP replaces the OpenStack's built-in version, so it becomes necessary to ensure the regular behavior of that DHCP outside a vHGW tenant network (automatic port creation upon a spurious device makes traffic in the tenant's network is an insecure behavior that goes against the expected defaults of other OpenStack installations). A check is performed against the association table in Neutron DB and, if the network does not belong to a vHGW, then the vDHCP will behave like an unaltered version of dnsmasq.

The second is because the VNFs are actually within the same network as the tenant's devices (and having DHCP available for automatic network configuration of VNFs is a very desirable feature), so we need a way to exclude the requests from functions from those of external devices (otherwise the Device AAA will be populated with VNFs as devices).

Alongside port creation (with device association), we also need a way to update Neutron ports status so that the AAA is properly notified with the right messages and fields (for instance, port status is used to signal if there should be a RADIUS accounting start or an interim).

Other considerations. This module's functions were constructed being mindful of the need to prevent SQL injection attacks (for instance the hostname *"VAC-PHONE'; DROP DATABASE neutron_ml2; - -"*). Although prepared statements or SQL-side procedures were not used, the only value controllable by the device which is used in a SQL query is the MAC address. Proper verification of MAC format is always made before querying (hostname is not passed through SQL but rather Neutron CLI, which uses prepared statements).

5.2.2 DNSMASQ: RADIUS HANDLER

The RADIUS Handler was made using the *FreeRADIUS Client Library*[5]. The documented *.h* file of the actual implementation is available in the Appendix A.2, which helps to get a clearer developer-oriented vision.

The major methods it implements are:

- Access Request (with a special method for DHCP-DISCOVER).
- Accounting Start.
- Accounting Stop.
- Accounting Interim.

Access Request requires a different method for the DHCP-DISCOVER because the RADIUS fields are different, in the discover phase only the HGW id is sent, while in the request phase the device identifier, IP address and hostname are added.

[5] `http://freeradius.org/freeradius-client/`

5.2.3 DNSMASQ: DHCP STATE-MACHINE CHANGES

In order to implement the sequence diagrams at the proper stage of the DHCP protocol, I had to perform changes into the state-machine that implemented the protocol. I will highlight the changes made in a C-alike pseudo-code (given this is the easier way to describe them). The actual implementation of each pseudo-code is also available in the Appendix A.3.

On DHCP-DISCOVER:

```
if (isVHGW())
{
  if(isDevice(mymac))
  {
    boolean is_down = (getportstatus(mymac) == DOWN);
    if(is_down && doAuthorizeDiscover())
    {
      if (access_request_discover() != 0)
      {
        return 0;
      }
    }
  }
  else
  {
    createport(mymac, client_hostname); // dhcp-agent will restart the
        proccess!
  }
}
```

On DHCP-REQUEST:

```
if (isVHGW())
{

  if (isDevice(mymac))
  {
    boolean is_down = (getportstatus(mymac) == DOWN);
    if (is_down)
    {
      if(doAuthorizeRequest())
      {
        if (access_request(mymac, ip_address, client_hostname) != 0)
        {
          return 0; // <--- NACK generation not required.
        }
      }

      updateport(mymac, ACTIVE);
      account_start(mymac, ip_address, client_hostname);
    }
    else
    {
```

```
      if (notifyOnRenew())
      {
        if (doInterim())
          account_interim(mymac, ip_address, client_hostname);
        else
          account_start(mymac, ip_address, client_hostname);
      }
    }

  }
}
```

On DHCP-RELEASE:

```
if (isVHGW())
{
  if (!doAuthorize()) updateport(mymac, DOWN);  // Avoid Policy rules
      reinstalation.
  if (notifyOnRelease() && isDevice(mymac))
  {
    // Notify RADIUS
    account_stop(mymac, ip_address, client_hostname);
  }
}
```

On DHCP-INFORM:

```
if (isVHGW() && doAuthorizeInform())
{
  if(isDevice(mymac))
  {
    if (access_request(mymac, ip_address, client_hostname) != 0)
    {
      message = ignored;
    }
  }
}
```

5.2.4 NEUTRON EXTENSION

This Neutron Extension allows the registration of a given vHGW into a Neutron Network (which enables the automatic external-port creation and DHCP configuration for external devices). It was developed using Neutron's existing extension framework, which allows for API enrichment and DB handling via Python code.

Neutron's Python client was also extended so that the new API functions were also supported from the command-line. As a end result the following commands were made available:

vhgw-network-create Register a vHGW and associate a neutron network to its entry.

vhgw-network-delete Delete a given vHGW.

vhgw-network-list List the registred vHGWs and their corresponding networks.

vhgw-network-show Show the vHGW corresponding neutron network.

vhgw-network-update Update the neutron network associated to a vHGW

5.3 RESULTS

The following results were gathered in two different machines. The PoC(s) were done using Alex, a 2x Intel(R) Xeon(R) CPU E5607 @ 2.27GHz with 16GB of DDR3 RAM @ 1333Mhz. The other results (unless stated otherwise) were gathered using Bica, a 2x Intel(R) Xeon(R) CPU X5570 with 48GB of DDR3 RAM @ 1333Mhz.

Both machines feature the same OS, an Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-63-generic x86_64), with similar configuration.

5.3.1 Proof-of-Concept

The PoC consisted of a Raspberry Pi with a WiFi adapter acting as a pHGW (establishing the GRE tunnel to the datacenter) and the machine nicknamed Alex performing the datacenter role.

We had the two functions described in the architecture already instantiated in the PoP (Firewall and Content Filtering, 4.1.2), being the SFC configured dynamically (on-the-spot) as needed. Devices were allowed to connect freely and put on the default chain (which was a sort-of "no chain" that would just follow the regular Neutron network behavior). The Firewall was used to block all traffic from a given device or to block just some applications (in this case the game OpenArena), while the Content Filter would inspect web-pages and block those that matched the filters introduced at the moment (although complex filters were possible, the more usual demonstration was just a website like "abola.pt").

The PoC was run more than once and with different audiences, allowing for different users with different skill-sets to use the system (nevertheless the users were mostly from an engineering background).

Upon completing a few runs of a live PoC, the observational results are that this vHGW (as presented) is already capable of seamlessly replacing a conventional HGW without giving away to the live audience that the network functions aren't running on the local hardware (while performing just the duties of that conventional HGW). Once we configure SFC and the respective VNFs, then the audience starts to realize it must not be (just) the conventional hardware performing those actions as the services provided (in this case the content filtering) required more granular control (and computational power) than would be possible by our Raspberry Pi.

The system with SFC was proven to be functioning properly, with chain changes happening in a way that was perceived by the user as being "instant". The general impressions were very positive (as described above) only with some rare concerns about data privacy (that were quickly dissipated once

the critic realized the role of a conventional HGW is exactly to give your data to the operator so that he can route it to the Internet).

PT Inovação also ran their set of PoCs, with different pHGW equipment and their own additions, but with similar results (only better user experience given there was a costumer portal available).

More objective results follow next.

5.3.2 vDHCP

In order to produce objective data of the vDHCP's performance, I have measured the times for acquiring an IP configuration with the vDHCP and then compared them with a traditional HGW (a Linksys WRT54GL v1.1, running DD-WRT SVN 14896 – which also uses dnsmasq as DHCP server). Given that, upon the first connection to the virtual environment the vDHCP has to create a new port for the device and then reset the DHCP proccess, I have also measured the time it takes to acquire an IP address for the first time versus the subsquent attempts. Each test was repeated 100 times in order to produce statistically relevant data. The DHCP client was dhcpcd5 v6.0.5.

Test	Average	Std. Dev.
Time for connection on traditional HGW	0.525 s	0.289 s
Time for first connection	4.071 s	0.616 s
Time for subsequent connections	0.042 s	0.007 s

Table 5.1: Time required for the device's DHCP configuration.

The hard-data coupled with the observational results from the PoC allow me to conclude that the longer time for the first connection (around 4 seconds) is not significant or perceivable in any negative way by the user, no complaints were made in this regard despite users being unaware to the fact the first connection would take longer. Given that subsequent connections are handled even faster than the legacy HGW, the vDHCP shouldn't pose any performance concerns in this regard.

Dnsmasq is not multi-threaded or allow for any concurrent processing, therefore attempting to test simultaneity would only result in the linear sum of the time required to process the queued requests (that arrived first) plus the time of that request.

Another test being considered was to determine the "cut-off" point where the vDHCP/dnsmasq would no longer queue requests but rather force the client to repeat the request. However, the only change the vDHCP has versus dnsmasq is the port creation that happens upon a DHCP-DISCOVER. Given that Windows DHCP client has a time-out for the first DHCP-DISCOVER of 5 seconds (and that the first connection takes 4 seconds), evaluating this change would be moot: only one first connection happens at a time, others will have to repeat the request.

5.3.3 NAT

Given that IPv4 is still the most widely used protocol in the Internet, due to its inherent scarcity NAT is a critical function of any HGW.

Having this in mind, I have proceeded to test the NAT bandwidth attainable by this virtual environment under different circumstances (cardinality of networks and devices). For that purpose

I have created the necessary number of networks (with respective routers) to which I have attached Linux Network Namespaces that simulate each of the clients.

iperf (version 2.0.5 (08 Jul 2010) pthreads) was the benchmark tool of choice, being configured for TCP protocol (also the most used protocol in the Internet) with 5 parallel connections (per client). This number of parallel connections was determined (beforehand) through crude experimentation, being this a value that yielded both the highest bandwidth and the most repeatable results.

All tests (with the sole exception of the last GRE tests) were repeated 100 times in order to produce statistically relevant data.

I will start by presenting the bandwidth graphs pertaining to each test scenario and then proceed to a more detailed analysis of the results (with tables). The graphs which show similar data (but in a different scenario/device) have the same x-axis scale so that they can be directly compared.

The first set of graphs show the scenario with 1 Network and 10 Devices in that network (fig. 5.1), then the next set will show the scenario with 10 Networks and 1 Device in each (fig. 5.2).

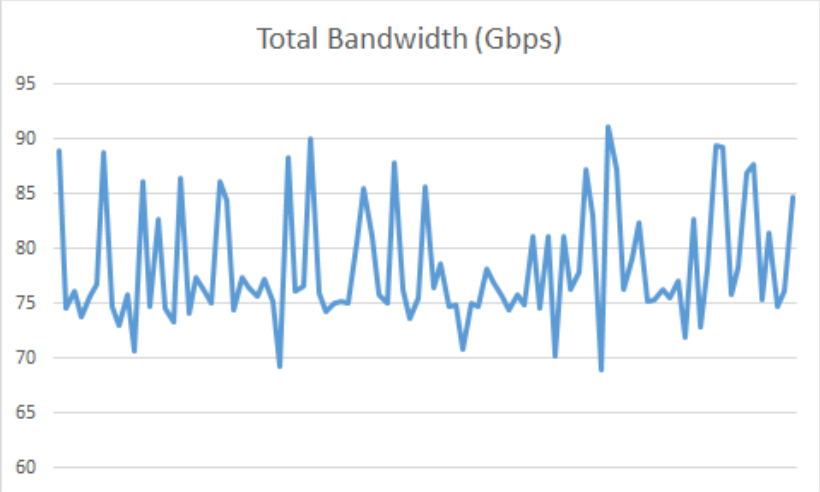

Figure 5.1: NAT Total Bandwidth (1 Network, 10 Devices).

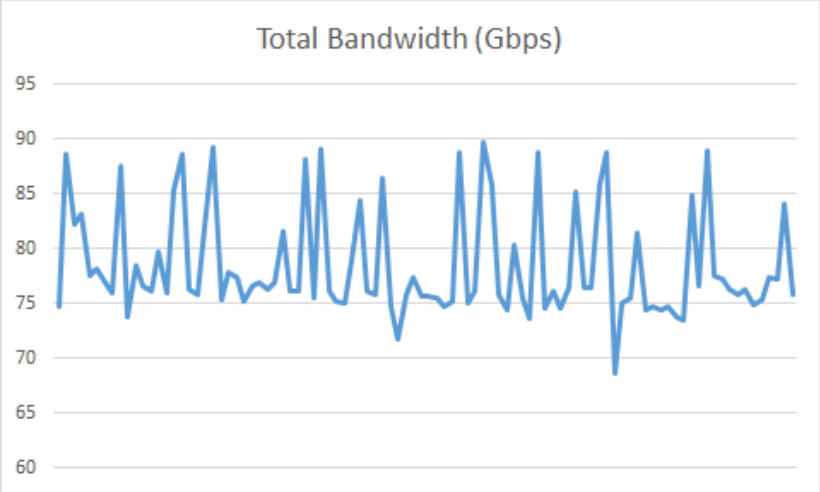

Figure 5.2: NAT Total Bandwidth (10 Network, 1 Device in each).

Test	Average	Std. Dev.
Single Device (1 Network)	60.98 Gbps	1.98 Gbps
1 Network with 10 Devices	78.30 Gbps	5.31 Gbps
10 Networks with 1 Device (each)	78.45 Gbps	4.93 Gbps

Table 5.2: Total NAT Bandwidth.

Device	(1 LAN) Avg	Std. Dev.	(10 LAN) Avg	Std. Dev.
0	7.73 Gbps	1.23 Gbps	7.84 Gbps	1.26 Gbps
1	7.65 Gbps	1.01 Gbps	7.80 Gbps	1.30 Gbps
2	7.73 Gbps	1.65 Gbps	7.84 Gbps	1.38 Gbps
3	7.85 Gbps	1.46 Gbps	7.89 Gbps	1.46 Gbps
4	7.92 Gbps	1.23 Gbps	7.84 Gbps	1.54 Gbps
5	7.86 Gbps	1.35 Gbps	7.86 Gbps	1.45 Gbps
6	7.97 Gbps	1.48 Gbps	8.03 Gbps	1.65 Gbps
7	7.67 Gbps	1.48 Gbps	7.50 Gbps	1.24 Gbps
8	7.82 Gbps	1.39 Gbps	8.21 Gbps	1.87 Gbps
9	8.10 Gbps	1.80 Gbps	7.65 Gbps	1.51 Gbps

Table 5.3: Average NAT throughput per Device.

This allows to conclude that, in every sense, the two scenarios are equal in behavior (within the margin of error). It should be noted that, versus the single device (in a single network), we can see some gains in bandwidth (outside the margin of error) by increasing the number of simultaneous devices. Furthermore, the total attainable bandwidth is likely to well surpass the achievable by the network cards that connect the machine to the physical network (in this case we had available 5x 1Gbit Ethernet adapters).

However, for the purposes of this vHGW, we need to account for the effects caused by the encapsulation of the tunnel. In order to get a sense of how the GRE tunnel affects the above results, I have setup a simulated pHGWs (one bridge in OVS per simulated gateway) which connects the clients to the environment through a GRE tunnel. The tests were repeated only 10 times, just to get a sense of the scale.

Test	Average	Std. Dev.
GRE - Single Device	8.10 Gbps	0.22 Gbps
GRE - 1 LAN x 10 Devices	13.83 Gbps	1.10 Gbps
GRE - 10 LANs (1 Device each)	13.64 Gbps	0.57 Gbps

Table 5.4: Total NAT throughput (with the pHGW connected through a GRE tunnel).

Although the drop in performance is substantial when using the GRE tunnel, given the added-value in flexibility it provides and the fact it still allows for enough performance to surpass all the network bandwidth possible to be achieved with the installed network adapters, I believe it is a very worthy trade-off that (in this case) presents no actual penalty.

For more detailed analysis of the GRE tunnel, Igor Cardoso spent most of his MSc dissertation [2] on the matter.

5.3.4 SERVICE FUNCTION CHAINING

For the objective evaluation of SFC performance (within the context of the vHGW) it was measured how long did it take to perform a chain change. This test measures the time it takes from the call of the API (to create the new steering classifier, inclusive) up to the moment traffic effectively starts entering a different SFC. The test was repeated 100 times.

Test	Average	Std. Dev.
Time needed to apply a new chain	131.01 ms	16.78 ms

Table 5.5: Time to apply a new chain.

The approximately 130 ms are fast enough to be perceived as an "immediate" reaction when triggered and observed by a human, being well within the speed requirements of the block VNFs used in this PoC. However, there may be scenarios in which a 130 ms delay may become too much. For instance, if one were to develop a security application with Traffic Inspection that didn't block traffic itself but rather performed live blocks/deeper analysis through chain changes, then the 130 ms delay would likely allow a "to be blocked" connection to be established and have a few packets exchanged before the chain would be changed.

Afterwards, it was measured how latency behaves as the service chain grows in length. The VNFs used in this test were Debian 8 images, running within a KVM hypervisor and with their kernel set to forward the packets. Latency was measured using the ping command (repeated 1000 times per test). Because ping actually measures RTT, one must have in mind the added contribution to the result made by the ICMP-REPLY packet (that does not enter any chain, as per configuration).

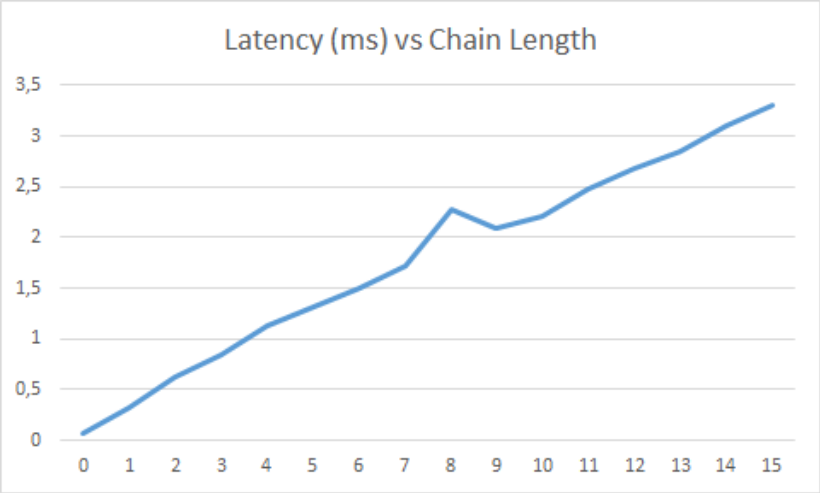

Figure 5.3: Latency trend as the chain length grows.

Outside of a "fluke" that happened when the chain length is 8, we can conclude the latency grows in a linear proportion to the chain length. It was also determined that (excluding the case with 8 hops) on average the latency grows at 0.245 ms per added hop (with a margin of error of 0.103 ms). In this instance I will also provide the data in tabular form.

CHAPTER 6

Design and Specification (Phase 2/Classifier + Shaper)

Upon successful completion of the first phase objectives, new "big-picture" goals have been set to enrich the first architecture (and PoCs), which required this dissertation to create a Traffic Classifier VNF, a Traffic Shaper VNF and extend a communications API to configure these VNFs. I will first start by describing the "big-picture" changes in the vHGW scenario, followed by the design and specification of each of the VNFs (along with the communications API).

6.1 big picture: the vHGW phase 2

In the mobile world, there is Huawei's Service Based Routing (SBR) [19] architecture which takes the logical blocks of the Policy and Charging Enforcement Function (PCEF)/Traffic Detection Function (TDF) and couples the Traffic Classifier capabilities with new Traffic Steering/SFC (figure 6.1). This allows for more granular control over traffic, with more suitable (and powerful) processing chains that best fit that classification and policy. SDN and NFVs within a cloud environment are also in the evolution path of that architecture.

A major change in the "big-picture" passes by drawing a parallel with the mobile world, introducing capabilities similar to the SBR into the vHGW. Although the vHGW is currently being used as a replacement CPE within the land-line context, this change allows for a clearer convergence path between land-line and mobile accesses, in a fashion which is familiar and follows the same patterns as existing mobile solutions.

I will start by presenting the new requirements and goals set for this phase, followed by the respective architecture.

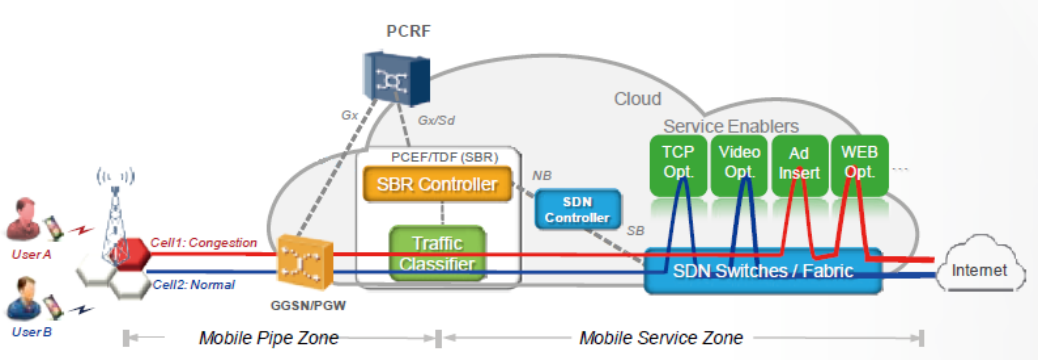

Figure 6.1: Huawei's SBR architecture

6.1.1 REQUIREMENTS AND GOALS

In this phase the vHGW will become policy-driven, with SFC being selected based on traffic classification and policy (configurable per device and/or subscription). As a means to demonstrate its value-added, on top of the Classifier which will introduce the classification (this dissertation's work), two additional VNFs will be introduced, a Traffic Accounter (by Igor Cardoso) and a Traffic Shaper (also part of this dissertation's work).

New use-cases are introduced to take advantage of policies. One new use-case is to have policies with traffic quotas per classification, having the accounter trigger a quota expired event and subsequently having the Shaper throttle just the traffic with that classification (and that device or subscription) to a given configurable value.

Another use-case is to zero the accounting of given classifications, as means to avoid charging of that traffic.

Variants such as blocking instead of Shaping are also contemplated in the use-cases, which gives support for the first phase use-cases.

Summarizing the new components and its respective functionality requirements:

- A Traffic Classifier VNF
- A Traffic Accounter VNF
- A Traffic Shaper VNF
- Extend the VNF Manager's API so that it can also configure those VNFs

The Traffic Classifier must be able to inspect network traffic up to the application level, being able (for instance) to tell apart Youtube traffic from Skype traffic. The classification marks must be configurable and may change while the classifier is already up and running.

The Traffic Accounter must the able to count the amount of data that has been accrued by a device in any given protocol (as classified by the classifier). It must also trigger configurable events (such as a quota was reached for a given protocol). Additionally, a device may also have to be triggered for throttling (in all protocols) to a given rate.

The Traffic Shaper must throttle the traffic of a given protocol (as classified by the classifier) to a given rate. The rate and protocols to throttle must be configurable and may change while the shaper is up and running.

All configuration happens through the REST-based VNF API.

The end-goal is to make a PoC which demonstrates a system which can apply external policies that introduce varying levels of service according to protocol, quota and time of day, through the use of SFC.

Do note that agility and time-to-market will also be put to the test, with short implementation times being granted and being expected the reuse of the previous architecture (and existing components) to the most.

6.1.2 ARCHITECTURE OVERVIEW

Building on top of the previous architecture in this phase, aside from the additional VNFs, we will also have a Policy Server (which will give policies to the Classifier), a User Management Portal (which will allow the end-user to configure his vHGW) and a VNF API which will allow the configuration of the VNFs by other architecture elements (such as the Portal and the Policy Server).

Both the Policy Server and the User Portal could be run either inside the Datacenter/PoP or as an external element. In the course of this dissertation the setup made had the Policy Server as an external element and the User Portal served from the Datacenter/PoP.

The role of the Datacenter/PoP will, in this phase, be played by a single machine (nicknamed Bica), more details about the machine will follow in the results section.

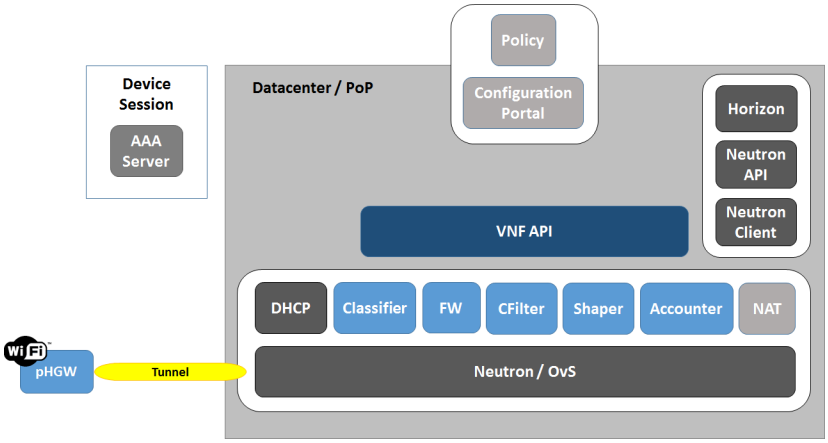

Figure 6.2: Phase 2 high-level architecture.

Like in phase 1, this architecture also keeps compatibility with PT Inovação vCPE efforts, being the Policy Server and Costumer Portal in fact parts taken from that effort.

6.2 VIRTUAL NETWORK FUNCTIONS

In this section I will zoom into the requirements of this dissertation's VNFs.

6.2.1 TRAFFIC CLASSIFIER

The Traffic Classifier inspects network traffic up-to the application layer, marks it (here using the Type of Service (ToS) bits) and then re-introduces it into the network in an already classified state. It must be able to be reconfigured while running, without requiring a restart of the classifier. The protocols which need to be detected are specified beforehand and will not change while running, only the marks which correspond to each protocol will be reconfigured as required.

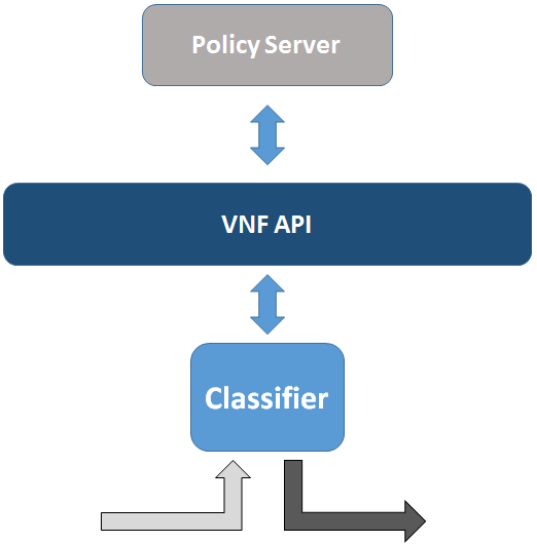

Figure 6.3: Traffic Classifier High-Level view.

The protocols which it must be able to detect (in order to support the use-cases) are: HTTP; Facebook; Youtube; OpenArena; Twitter; Skype; Viber; Whatsapp; P2P (Bittorrent, eDonkey, gnutella, Fasttrack and DCC);

Mark reconfiguration needs to be made available through the VNF API (6.3). A "big-picture" requirement was introduced in this dissertation, in the form of a translation table between a RuleId and detected protocol/service. The table is presented in 6.1 as specified.

The classifier will always return the most granular classification of any protocol/service. That is, taking Facebook as an example, despite being a service that also runs over HTTP the classifier will (to the best of its abilities) classify all packets pertaining to that service as being just Facebook (not HTTP).

Rule ID	Service
1	HTTP
2	P2P
3	Facebook
4	Youtube
5	Quake III/OpenArena
6	Reserved for Meo Go
7	Twitter
8	Skype
9	Whatsapp
10	Viber

Table 6.1: Map Traffic Classifier RuleId to Protocol in VNF-API.

6.2.2 TRAFFIC SHAPER

The Traffic Shaper throttles the traffic marked by the classifier to a given rate, depending not only on the mark but also the rules created for specific devices or for the whole subscription. The sum of the rates of all devices cannot surpass the whole subscription's limit. The rate, devices and marks to throttle must be configurable and may change while the shaper is up and running.

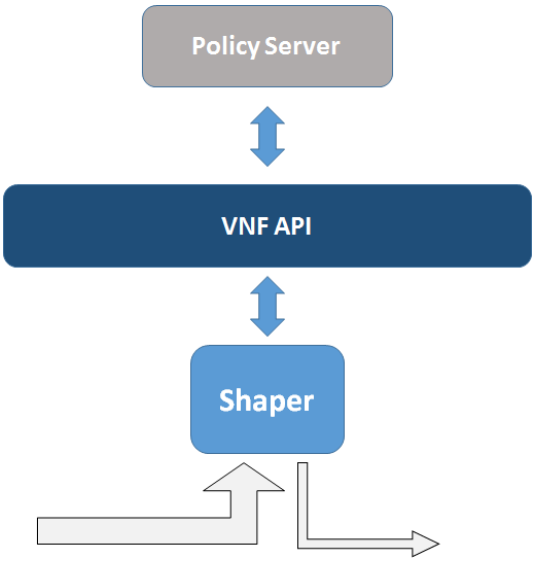

Figure 6.4: Traffic Shaper High-Level view.

The required functionality is:

- Cap/Uncap the total bandwidth of a Subscription.
- Cap/Uncap given Device (identified by its IP address).
- Cap/Uncap a given Service (identified by its ToS/DSCP) for a given Device.

Additionally, when the same filter (selector) is configured multiple times, the latest configuration of that filter must overwrite all previous attempts.

6.3 VNFS COMMUNICATIONS API

The VNFs Communications API is of pivotal importance given that this is the component which will handle all interactions (configuration, reporting or event triggering) of all functions with outside elements (at this stage only the Policy Server).

It builds on a framework started (internally) by Miguel Dias and then continued (also internally) by Igor Cardoso.

For the purposes of this dissertation's work, the requirements are that it must allow the fulfillment of all operations specified in 6.2.1 and 6.2.2 by exposing a REST API to the elements outside of the VNFs (the way that those VNFs communicate with the API is of "open-choice").

CHAPTER 7

Implementation and Results (Phase 2/Classifier + Shaper)

I will now present the "big-picture" compromises, followed by the implementation rationales of the Classifier and Shaper (with their respective API) and then proceed to the results.

7.1 big picture: vHGW general compromises

The classifier marks the traffic using the ToS bits and, subsequently, chain selection is made upon the ToS. This has the drawback that some services may already use ToS (for instance for Quality of Service (QoS), such as SIP) or may alter the ToS value at will. For instance, HTTP proxy functions will receive marked traffic, open a new connection to the outside and then (very likely) re-introduce packets without the original mark.

Although the classifier will ensure that all packets exit with a valid ToS (within the context of the configured rules), it cannot ensure that other functions wont change it internally.

Additionally, OVS/OpenFlow configuration imposed that the ToS bits were used like DSCP (that is, only the 6 most significant bits of the ToS field could be used).

7.2 implementation

I will now move to the implementation of the VNFs themselves. In this phase agility and time-to-market were put to the test, with short implementation times being granted (no more than a few weeks). Therefore, quick and easily adaptable solutions were in order.

7.2.1 TRAFFIC CLASSIFIER VNF

The Traffic Classifier was built using Linux netfilter[1] (commonly known for the iptables command) and a modified version ntop's nDPI[2] (that worked with the Linux kernel 3.13). Other alternatives were also considered, such as the old L7-filter[3] and libprotoident[4]. However, L7-filter is mostly a precursor work that lead to the creation of ndpi-netfilter, with the additional inconvenience of no longer being actively developed which translated to not supporting all the required protocols. On the other hand, libprotoident would require far more time to build the Classifier, therefore nDPI was preferred.

Netfilter was preloaded with rules that configured the filters for the protocols which it was required to detect and gave those connections an internal mark (using conntrack and a target in mangle PREROUTING). Then, when the packets are about to be re-introduced into the network, the internal mark is translated to the configured ToS (in mangle POSTROUTING).

The use of conntrack avoids the need to process all packets of an already classified flow. Conversely, a characteristic of this solution is that all packets of an uncategorized protocol will always be processed by the function, making this the worst case performance-wise.

The communication between the VNF-API and this function is done via SSH, which will run previously prepared shell scripts that match the API calls. This scripts are in the home of the API's user, which will take (as parameters) the data passed via REST to the VNF-API.

The Classifier VM itself is an Ubuntu 14.04.03 LTS, with dropbear[5] as the SSH server.

I will present the iptables rules that perform this classification and marking process in appendix (A.4), as this is a straight-forward way to demonstrate the intrinsic elegance and adaptability of the solution (rather than just make the claim).

The details of the API are in the VNF-API subsection (7.2.3).

7.2.2 TRAFFIC SHAPER VNF

The Traffic Shaper was implemented using Linux Traffic Control[6], building a HTB tree with FQ_Codel[7] queues for better usage of the shaped bandwidth. Pre-built scripts such as The Wonder Shaper[8] were also considered, however using Linux Traffic Control directly is the way that gives the most flexibility to adapt to future requirements, therefore this was the implementation's choice.

The communication between the VNF-API and this function is done via SSH, through the execution of a collection of prepared shell scripts present in the home of the API's user. This scripts will take (as parameters) the data passed via REST to the VNF-API and then pass the (now validated) data to a generator built in C which will (itself) create a new script (similar to The Wonder Shaper) which will configure the shaping itself through Linux Traffic Control.

The function's configuration data is persisted across calls in a flat-file structure which is stored in a tmpfs file-system. The generator will then read those files and keep previous rules.

[1] `http://www.netfilter.org/`
[2] `https://github.com/betolj/ndpi-netfilter`
[3] `http://l7-filter.sourceforge.net/`
[4] `http://research.wand.net.nz/software/libprotoident.php`
[5] `https://matt.ucc.asn.au/dropbear/dropbear.html`
[6] `http://www.lartc.org/`
[7] `http://www.bufferbloat.net/projects/codel/wiki`
[8] `http://lartc.org/wondershaper/`

The VM itself is an Ubuntu 14.04.03 LTS, with dropbear[9] as the SSH server.
The details of the API and all its methods are in the VNF-API subsection (7.2.3).

7.2.3 VNFS COMMUNICATION API

The VNF-API contains portions that were developed either by myself or others. In this section I will only document my own methods, which are the ones that pertain to this dissertation. As an additional note, I have also documented these methods in other "external" formats to promote its use in further developments.

Starting by the classifier function, the "big-picture" table which translates the Rule ID into the detected protocol (table 6.1) is fundamental for the use of the REST configuration API.

The Classifier's API is made of a single method, which associates a ToS to one of the RuleIds.

Resource path: /classifier/{customer}/{ip}
HTTP verb: POST
Customer ID (customer): The customer identifier.
Session (ip): The device IP address which identifies the session.
Data: (array of tuples)

- Rule ID (ruleid): Rule identifier;
- Service (tos): Value to be applied to the packets that match this Rule ID;

(All fields of the tuples are mandatory. Array must be non-empty.)

Following is the Shaper's REST API, which has many more methods than the Classifier.

> Cap the Total Bandwidth

Resource path: /shaper/{customer}
HTTP verb: POST
Customer ID (customer): The customer identifier.
Data:
Upload Rate (ul_rate, optional*): The rate limit (in kilobits per second).
Download Rate (dl_rate, optional*): The rate limit (in kilobits per second).

(*) at least one must be present

> Uncap the Total Bandwidth

Resource path: /shaper/{customer}
HTTP verb: PUT
Customer ID (customer): The customer identifier.

[9] `https://matt.ucc.asn.au/dropbear/dropbear.html`

Data:

Upload Rate (ul_rate, optional*).
Download Rate (dl_rate, optional*).
(*) at least one must be present

(Note: This method will preserve any Device cap that may follow, ie. it will match the network interface maximum bandwidth without removing the leafs)

> Uncap the Total Bandwidth (Hard)

Resource path: /shaper/{customer}
HTTP verb: DELETE
Customer ID (customer): The customer identifier.

(Note: Drops all Device caps that may follow, ie. it will cut the tree by its root.)

Cap a Device / Service

Resource path: /shaper/{customer}/{ip}
HTTP verb: POST
Customer ID (customer): The customer identifier.
Device (ip): The device IP address.
Data:

- **Upload Rate (ul_rate, optional*):** The upload rate limit (in kilobits per second) for the capped device.
- **Download Rate (dl_rate, optional*):** The download rate limit (in kilobits per second) for the capped device.
- **Array of (optional*):**
  - Service (tos), Upload Rate (ul_rate, optional**), and Download Rate (dl_rate, optional**): The ToS tag that identifies the service with the respective rate limits for that service.

(*, **) at least one must be present, in their respective grouping

> Uncap a Device

Resource path: /shaper/{customer}/{ip}
HTTP verb: PUT
Customer ID (customer): The customer identifier.
Device (ip): The device IP address.
Data:

Upload Rate (ul_rate, optional*).
Download Rate (dl_rate, optional*).
(*) at least one must be present

(Note: This method will preserve any Service cap that may follow for that Device, ie. it will match the total bandwidth cap without removing the leafs)

> Uncap a Device (Hard)

Resource path: /shaper/{customer}/{ip}
HTTP verb: DELETE
Customer ID (customer): The customer identifier.
Device (ip): The device IP address.

(Note: Drops all Service caps that may follow for that Device, ie. prune at the Device cap)

> Uncap a Service

Resource path: /shaper/{customer}/{ip}/{tos}
HTTP verb: DELETE
Customer ID (customer): The customer identifier.
Device (ip): The device IP address.
Service (tos): The ToS tag that identifies the service.

7.3 RESULTS

The following results were gathered using Bica, a 2x Intel(R) Xeon(R) CPU X5570 with 48GB of DDR3 RAM @ 1333Mhz running Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-63-generic x86_64).

7.3.1 TRAFFIC CLASSIFIER

The Traffic Classifier was shown to be providing the required functionality during internal testing. In order to evaluate its performance, I will repeat the same bandwidth tests that were used for the NAT function evaluation (5.3.3) and perform an analysis of those results.

The Classifier VNF was initially given 1 vCPU and 512MB of RAM. A second test for the single network with 10 devices was also made with 8 vCPUs and 4GB of RAM.

One very important aspect of this test is that, the traffic generated by the benchmark tool (iperf) does not fall under any of the protocol filters. This is the worst case scenario for the performance of the classifier, as this way every single packet will have to be analyzed (while for known protocols, once the identification is made, that connection is no longer passed through the filters – only the previous mark is restored from the conntrack and applied to the ToS field).

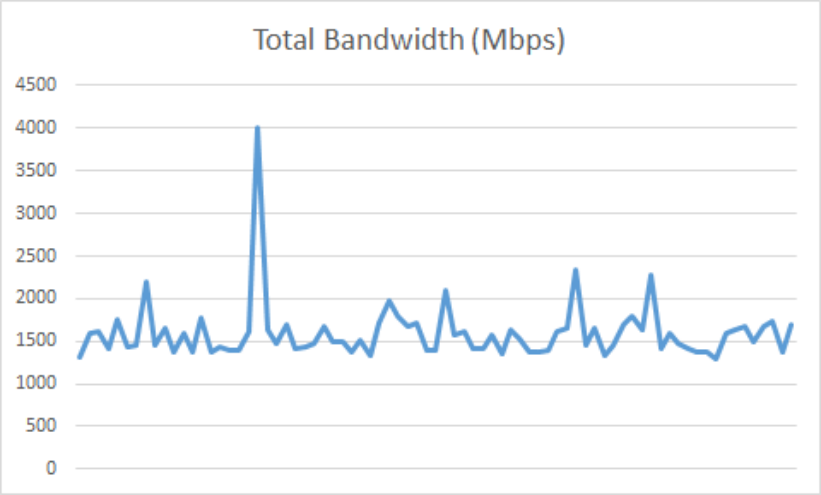

Figure 7.1: Traffic Classifier Total Bandwidth (1 Network, 10 Devices).

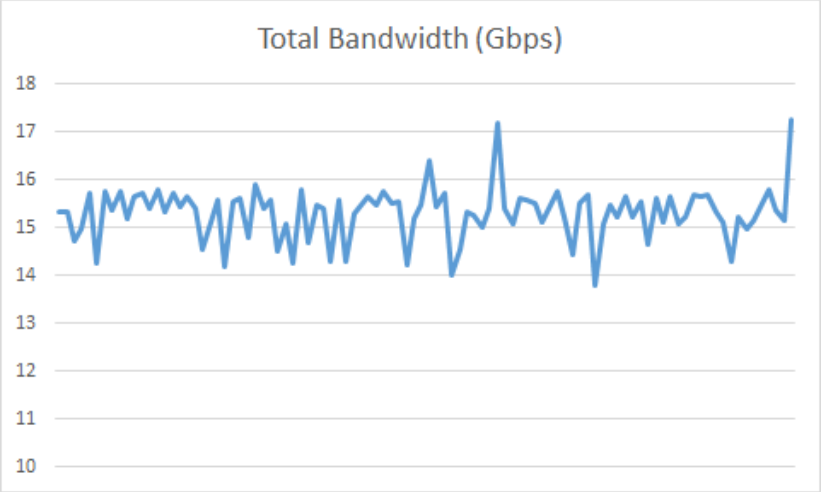

Figure 7.2: Traffic Classifier Total Bandwidth (10 Networks, 1 Device each).

Test	Average	Std. Dev.
Single Device (1 Network)	4.30 Gbps	0.19 Gbps
1 Network with 10 Devices	1.60 Gbps	0.35 Gbps
10 Networks with 1 Device (each)	15.31 Gbps	0.55 Gbps

Table 7.1: Total Classifier throughput (iPerf in TCP mode. Each device makes 5 parallel connections.)

Given the glaring performance gap between the 1 Network with 10 Devices scenario and the other scenarios (almost a tenth of 10 Networks with 1 Device each), I initially believed the reason for this would likely be due to the lack of sufficient vCPUs alloted to the instance (VM) in which the classifier was running. Upon rectifying that "issue" (plus increasing the amount of RAM) and repeating the test, to my surprise, the results were pretty much equal (within the margin of error).

Upon further digging into the matter I arrived to conclusion that, unlike the main ntop nDPI project, the chosen ndpi-netfilter fork is not multi-threaded (and therefore does not take advantage of

multiple CPUs/cores). That is likely to cause a large queuing of packets to be processed (perhaps even packet drop) and, given the traffic is TCP, that may also cause a notable performance drop due to window adjustment and possible retransmissions.

Although I could do a more detailed break-down of the behavior per-device (like presented in 5.3.3), it does not seem to bring much value given how far apart the performance is for such a fine-grained comparison.

It must however be noted that, in all scenarios, the performance achieved is enough for the context of the PoC (and even in the context of a production HGW since, after all, the worst case is able to process over 1 Gbps of data for a single HGW).

7.3.2 TRAFFIC SHAPER

The Traffic Shaper has been validated to be working properly, with the bandwidth being capped as configured through the API.

Gathering hard-data for the shaper has been a challenge, given that bandwidth tests only show what is the expected behavior, CPU load tests show that the benchmarking tool is by far taking more load than the function itself (over 4 cores of CPU power when the function takes little percentage of a single core) and the same for the memory usage.

Ultimately it was registered how latency behaved with the use of this function. First gathered the "control" data without the traffic being steered into the shaper. Then I measured the effects of steering the traffic into the shaper but without doing any actual shaping (just a plain forward). It must be noted that, in the chaining configuration of these tests, the traffic will be steered into the shaper on two occasions: when the traffic is going upstream and when it is going downstream. From there followed a test in which the traffic was being shaped to 128 Kbps in just one direction. Finally, I performed the test of shaping to 128 Kbps in both directions.

All tests were done with the ping command and repeated 1000 times (in order to produce statistically relevant data). The "In-Chain (Disabled)" test case acts as a depiction of the latency observed in the full chain.

Figure 7.3: Traffic Shaper Latency Overview.

Test	Average	Std. Dev.	Min	Max
Control	0.066 ms	0.043 ms	0.029 ms	1.173 ms
In-Chain (Disabled)	0.493 ms	0.155 ms	0.202 ms	4.289 ms
One Direction (128 Kbps)	0.519 ms	0.115 ms	0.240 ms	2.125 ms
Both Directions (128 Kbps)	0.538 ms	0.103 ms	0.252 ms	1.898 ms

Table 7.2: Shaper Latency.

We can conclude the behavior (once the traffic is steered into the Shaper) is similar (within the margin of error) across all configurations of the shaper (disable, shaping in just one direction or both).

Once compared with the control, we can observe that the shaper follows the same trend measured in the subsection where the SFC was evaluated (5.3), namely the increase in latency of approximately 0.250 ms per hop. Because ping measures RTT, traffic will be steered 2 times into the shaper (once on the ICMP-REQUEST and then another on the ICMP-REPLY), which is equivalent to having 2 chain hops (therefore about 0.5 ms, which is within the margin of error of the results obtained).

CHAPTER 8

Closing Thoughts

8.1 conclusions

This dissertation presented the required VNFs that allowed the "big-picture" to demonstrate a vHGW with SFC in Cloud environments to solve the HGW problem was not only viable but could also deliver on its potential.

In phase 1 the vDHCP allowed for the connection of external devices to the Neutron network, which through the automatic port creation enabled automatic IP connectivity configuration in the device. The vDHCP has adequate performance, as shown both in laboratory tests and the user experience during the PoCs.

It also contributed to build the working grounds for a vCPE and capture valuable data that helps understand better how SFC behaves in the real-world, how the virtual functions perform their tasks, as well as how other critical functions (provided by the cloud environment, for instance NAT) behave within this context. The feedback of real users interacting with this system was highly gratifying and equally invaluable, as the user experience will be without a doubt a deciding factor for actual field applicability of this concept. The collaboration with the research unit of our largest operator gave great insight and, ultimately, highlighted the most imminent operator concerns and kept the works within a fairly realistic context.

The work carried-out during phase 2 presented a different kind of challenge, one that tested agility and the ability to rapidly develop new functions and integrate new paradigms in this vHGW system.

The shift towards a policy-driven HGW and the delivery of the Classifier and Shaper VNFs will for sure help construct more complex scenarios that better showcase the true potential of this system, with the capability to distinguish traffic all the way up to the application and perform policy-based chain selection upon this classification. Laboratory performance showed no impediment, despite the nuisance of the lack of efficient multi-threading in the DPI.

The new capacity of triggering events based on classification and the reaching of a certain quota opens doors to more interesting operator scenarios, in which dynamic bandwidth capping may be applied after reaching a certain P2P quota or an OTT agreement may be reached to sponsor the use of a certain application (for instance clients get free Facebook, sponsored by some promotional action). Having already developed and tested the basic functionality of the required VNFs, that integrate within a unified VNF-API, takes us one step closer to making the demonstration (PoC) of these new scenarios a reality.

Other players are also actively working on similar concepts, which is a great opportunity to learn from their experiences and expertise (contribute to the discussion if possible), and this also validates the merits of continuing the work made so far.

Next I will point-out a few possible ways in which the "big-picture" can evolve.

8.2 FUTURE WORK

High-Availability and Failure Detection

Although this was actually already pointed-out during the shortcomings of the system, having High-Availability and Failure Detection mechanisms would (without any doubt) allow to bridge the gap faster between what is a PoC and a real-world deployable product.

Service Context Mobility

We could shift towards a model which allows for all user's (service) context to be carried-over and accessible regardless if that user is using his own subscription or someone else's (within the same operator). That is, the services available for use within your subscription could seamlessly still continue to recognize you and employ your settings even when using someone else's subscription.

Merge Mobile with Landline Accesses

Why stop with just service context? Since your home network is now a virtual entity in the PoP one could just as well allow mobile accesses (3G or 4G for instance) to connect to your home network and communicate with the other devices just like they were connected to your pHGW.

Explore OTT Oportunities

Having a new (lucrative) business idea is hard, however we now have the capabilities to rapidly deploy innovative ideas and explore opportunities with Over-The-Top (OTT) providers, such as sponsored content, burstable bandwidth over the user's contract (when possible) or even freemium accesses.

Expand Beyond the Home Gateway

The HGW was just a chosen problem to which SFC and Cloud environments could bring an advantage. However, there are other CPEs (such as Set-Top Boxes) which could also benefit from this.

APPENDIX A

Appendices

A.1 VDHCP NEUTRON.H

```
/*!
\file   neutron.h
\brief  Allows Dnsmasq to identify vHGW networks and perform Neutron port
    related actions in Openstack.
\author Vitor Cunha (vitorcunha@ua.pt)
\date   February 5th, 2015
*/

typedef enum {DOWN, ACTIVE} portstatus_t;

/*!
\brief    Performs the initialization (optional)
\details  This method performs all necessary actions to initialize the
    internal variables and make the module ready for usage.
If not called beforehand, the functions which depend on this initialization
    will automatically call this method.
*/
void neutron_init(void);

/*!
\brief    Check if the Network to which Dnsmasq is attached belongs to a vHGW
\details  This method will acquire the network_id from the path of the
    hostsfile and then query the Neutron DB to check if
that Network is in use by a vHGW.

\return 0 is false, 1 is true, -1 if an SQL error occurred.
*/
int isVHGW(void);

/*!
\brief    Check if this MAC belongs to a client Device
\details  This method will query the Neutron DB to check if this MAC belongs
    to any OpenStack element.

\return 0 is false, 1 is true, -1 if an SQL error occurred.
*/
int isDevice(char* mac);

/*!
\brief    Create a Neutron port for the Client device
\details  This method will create a new Neutron port which represents the
    client device in the respective Neutron network.

\param mac The client MAC address.
\param name The client Hostname.
\return 0 on success, -1 otherwise.
*/
int createport(char* mac, char* name);

/*!
```

```
\brief    Update Neutron port status
\details  This method will change the status of the Neutron port which
    represents the client device in the respective Neutron network.

\param mac The client MAC address.
\param status Neutron port status {DOWN, ACTIVE}
\return 0 on success, -1 on SQL error, respective errno otherwise.

\warning   \a mac must be verified beforehand to be of a valid format,
    otherwise it may allow for a SQL injection attack!\n
\b Tip: if you use \a print_mac() from \a utils.c you will remove any chance
    of an injection attack.\n\n
*/
int updateport(char* mac, portstatus_t status);


/*!
\brief    Gets the status of a Neutron port
\details  This method will retrieve the status of the Neutron port which
    represents the client device in the respective Neutron network.

\param mac The client MAC address.
\return {DOWN, ACTIVE} or -1 on SQL error.

\warning   \a mac must be verified beforehand to be of a valid format,
    otherwise it may allow for a SQL injection attack!\n
\b Tip: if you use \a print_mac() from \a utils.c you will remove any chance
    of an injection attack.\n\n
*/
getportstatus portstatus_t(char* mac);

/*!
\brief    Get vHGW Identifier
\details  This method returns the vHGW identifier to be used with Device AAA.

\return The vHGW identifier or NULL if it's not a vHGW network.
*/
char* getVHGWid(void);

/*!
\brief    Gets the hostname from a Neutron port
\details  This method will retrieve the hostname of the device from the
    respective Neutron port.

\param mac  The client MAC address.
\param name Output pointer (memory allocated inside)
\return len(name) or -1 on SQL error.

\warning   \a mac must be verified beforehand to be of a valid format,
    otherwise it may allow for a SQL injection attack!\n
\b Tip: if you use \a print_mac() from \a utils.c you will remove any chance
    of an injection attack.\n\n
*/
int getportname(char* mac, char** name);
```

```
/*!
\brief    Get the string index at which the hostname DNS suffix starts
\details  This method retrieves the string index where the hostname's DNS
    suffix starts. Because Neutron doesn't allow to name ports with
the DNS suffix included, we will need to remove it in order the maintain
    consistency with hostname reported on the RADIUS notifications.
\return idx if a suffix exists, -1 if not.
*/
int idxDomain(char* name);
```

A.2 VDHCP RADIUS.H

```
/*!
\file   radius.h
\brief  Allows Dnsmasq to perform accounting actions in a RADIUS server.
\author Vitor Cunha (vitorcunha@ua.pt)
\date   April 15th, 2015
*/

/*!
\brief    Sends an Account Start notification.

\param mac The client MAC address.
\param ip The client IP address.
\param name The hostname.
\return 0 on success, negative otherwise.
*/
int account_start(char *mac, char *ip, char* name);

/*!
\brief    Sends an Account Stop notification.

\param mac The client MAC address.
\param ip The client IP address.
\param name The hostname.
\return 0 on success, negative otherwise.
*/
int account_stop(char *mac, char *ip, char* name);

/*!
\brief    Sends an Account Interim/Alive notification.

\param mac The client MAC address.
\param ip The client IP address.
\param name The hostname.
\return 0 on success, negative otherwise.
*/
int account_interim(char *mac, char *ip, char* name);

/*!
\brief    Sends an Access request.

\param mac The client MAC address.
\param ip The client IP address.
\param name The hostname.
\return 0 on success, negative otherwise.
*/
int access_request(char *mac, char *ip, char* name);

/*!
\brief    Sends an Access request during DHCP-DISCOVER
```

```
\param mac The client MAC address.
\return 0 on success, negative otherwise.
*/
int access_request_discover();

/*!
\brief    Performs the initialization (optional)
\details  This method performs all necessary actions to initialize the
    internal variables and make the module ready for usage.
If not called beforehand, the functions which depend on this initialization
    will automatically call this method.
*/
int radius_init(void);

/*!
\brief    Notify RADIUS also on DHCP-RENEW?
\details  This method checks if the server is configured to also notify the
    RADIUS server on DHCP-RENEW.
\return 1 if yes, 0 if no.
*/
int notifyOnRenew(void);

/*!
\brief    Notify RADIUS on DHCP-RELEASE?
\details  This method checks if the server is configured to also notify the
    RADIUS server on DHCP-RELEASE.
\return 1 if yes, 0 if no.
*/
int notifyOnRelease(void);

/*!
\brief    Notify with INTERIM on DHCP-RENEW?
\details  This method checks if the server is configured to notify the RADIUS
    server with an Account INTERIM on DHCP-RENEW.
\return 1 if yes, 0 if no.
*/
int doInterim(void);

/*!
\brief    AAA Authorize
\details  This method checks if the AAA is being check for Authorization
\return 1 if yes, 0 if no.
*/
int doAuthorize(void);

/*!
\brief    Perform AAA Authorize on DHCP-DISCOVER
\details  This method checks if the AAA is being check for Authorization
\return 1 if yes, 0 if no.
*/
int doAuthorizeDiscover(void);

/*!
\brief    Perform AAA Authorize on DHCP-REQUEST
```

```
\details  This method checks if the AAA is being check for Authorization
\return 1 if yes, 0 if no.
*/
int doAuthorizeRequest(void);

/*!
\brief    Perform AAA Authorize on DHCP-INFORM
\details  This method checks if the AAA is being check for Authorization
\return 1 if yes, 0 if no.
*/
int doAuthorizeInform(void);
```

A.3 VDHCP STATE-MACHINE CHANGES

On DHCP-DISCOVER:

```
if (isVHGW())
{
  char mymac[18];
  print_mac(mymac, emac, emac_len);

  // Strip the domain from the hostname
  int idx = idxDomain(client_hostname);
  if (idx != -1) client_hostname[idx] = '\0';

  if(isDevice(mymac))
  {
    struct in_addr a;
    if (&mess->yiaddr) memcpy(&a, &mess->yiaddr, sizeof(a));
    char is_down = (getportstatus(mymac) == DOWN);
    if(is_down && doAuthorizeDiscover())
    {
      if (access_request_discover() != 0)
      {
        return 0;
      }
    }
  }
  else createport(mymac, client_hostname);  // dhcp-agent will restart the
      proccess!

  // Reattach the domain to the hostname
  if (idx != -1) client_hostname[idx] = '.';
}
```

On DHCP-REQUEST:

```
if (isVHGW())
{
  char mymac[18];
  print_mac(mymac, emac, emac_len);

  if (isDevice(mymac))
  {
    char is_down = (getportstatus(mymac) == DOWN);

    // Strip the domain from the hostname
    int idx = idxDomain(client_hostname);
    if (idx != -1) client_hostname[idx] = '\0';

    // Perform the RADIUS notification
    struct in_addr a;
    if (&mess->yiaddr) memcpy(&a, &mess->yiaddr, sizeof(a));
    if (is_down)
```

```
    {
      if(doAuthorizeRequest())
      {
        if (access_request(mymac, &mess->yiaddr ? inet_ntoa(a) : NULL,
            client_hostname) != 0)
        {
          return 0; // <--- NACK generation not required.
        }
      }

      updateport(mymac, ACTIVE);
      account_start(mymac, &mess->yiaddr ? inet_ntoa(a) : NULL,
          client_hostname);
    }
    else
    {
      if (notifyOnRenew())
      {
        if (doInterim())
          account_interim(mymac, &mess->yiaddr ? inet_ntoa(a) : NULL,
              client_hostname);
        else
          account_start(mymac, &mess->yiaddr ? inet_ntoa(a) : NULL,
              client_hostname);
      }
    }

    // Reattach the domain to the hostname
    if (idx != -1) client_hostname[idx] = '.';
  }
}
```

On DHCP-RELEASE:

```
if (isVHGW())
{
  char mymac[18];
  print_mac(mymac, emac, emac_len);
  if (!doAuthorize()) updateport(mymac, DOWN);  // Avoid Policy rules
      reinstalation.

  if (notifyOnRelease() && isDevice(mymac))
  {
    // Strip the domain from the hostname
    int idx = idxDomain(client_hostname);
    if (idx != -1) client_hostname[idx] = '\0';

    // Notify RADIUS
    struct in_addr a;
    if (&mess->ciaddr) memcpy(&a, &mess->ciaddr, sizeof(a));
    account_stop(mymac, &mess->ciaddr ? inet_ntoa(a) : NULL, client_hostname);
```

```
    // Reattach the domain to the hostname
    if (idx != -1) client_hostname[idx] = '.';
  }
}
```

On DHCP-INFORM:

```
if (isVHGW() && doAuthorizeInform())
{
  char mymac[18];
  print_mac(mymac, emac, emac_len);

  // Strip the domain from the hostname
  int idx = idxDomain(client_hostname);
  if (idx != -1) client_hostname[idx] = '\0';

  if(isDevice(mymac))
  {
    struct in_addr a;
    if (&mess->yiaddr) memcpy(&a, &mess->yiaddr, sizeof(a));
    if (access_request(mymac, &mess->yiaddr ? inet_ntoa(a) : NULL,
        client_hostname) != 0)
    {
      message = _("ignored");
    }
  }
}
```

A.4 TRAFFIC CLASSIFIER

```
# Generated by iptables-save v1.4.21 on Wed Jul  1 22:18:15 2015
*mangle
:PREROUTING ACCEPT [19:1336]
:INPUT ACCEPT [19:1336]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [10:1424]
:POSTROUTING ACCEPT [10:1424]
:DPI - [0:0]
:RULES-DEFAULT - [0:0]
-A PREROUTING -m connmark ! --mark 0 -j ACCEPT
-A PREROUTING -m conntrack ! --ctstate ESTABLISHED -j CONNMARK --set-mark 0
-A PREROUTING -d 192.168.5.0/29 -j ACCEPT
-A PREROUTING -m connmark  --mark 0 -j DPI
-A POSTROUTING -j RULES-DEFAULT
-A DPI -m ndpi --http           -j CONNMARK --set-mark 4
-A DPI -m ndpi --facebook       -j CONNMARK --set-mark 6
-A DPI -m ndpi --youtube        -j CONNMARK --set-mark 7
-A DPI -m ndpi --quic           -j CONNMARK --set-mark 7
-A DPI -m ndpi --quake          -j CONNMARK --set-mark 8
-A DPI -m comment --comment "protocol␣OPENARENA" -p udp --sport 27960 -j
    CONNMARK --set-mark 8
-A DPI -m ndpi --twitter        -j CONNMARK --set-mark 10
-A DPI -m ndpi --skype          -j CONNMARK --set-mark 11
-A DPI -m ndpi --whatsapp       -j CONNMARK --set-mark 12
-A DPI -m ndpi --whatsapp_voice -j CONNMARK --set-mark 12
-A DPI -m ndpi --viber          -j CONNMARK --set-mark 13
-A DPI -m ndpi --bittorrent     -j CONNMARK --set-mark 5
-A DPI -m ndpi --edonkey        -j CONNMARK --set-mark 5
-A DPI -m ndpi --gnutella       -j CONNMARK --set-mark 5
-A DPI -m ndpi --fasttrack      -j CONNMARK --set-mark 5
-A DPI -m ndpi --directconnect -j CONNMARK --set-mark 5
-A RULES-DEFAULT -j TOS  --set-tos 0x00/0xff
COMMIT
# Completed on Wed Jul  1 22:18:15 2015
# Generated by iptables-save v1.4.21 on Wed Jul  1 22:18:15 2015
*filter
:INPUT ACCEPT [186:13056]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [99:13096]
COMMIT
# Completed on Wed Jul  1 22:18:15 2015
```

```
*mangle
-N RULES-DEVICEX
-A RULES-DEVICEX -m connmark  --mark 0   -j TOS  --set-tos 0x00/0xff
-A RULES-DEVICEX -m connmark  --mark 4   -j DSCP --set-dscp 4
-A RULES-DEVICEX -m connmark  --mark 5   -j DSCP --set-dscp 5
-A RULES-DEVICEX -m connmark  --mark 6   -j DSCP --set-dscp 6
```

```
-A RULES-DEVICEX -m connmark  --mark 7   -j DSCP --set-dscp 7
-A RULES-DEVICEX -m connmark  --mark 8   -j DSCP --set-dscp 8
-A RULES-DEVICEX -m connmark  --mark 10  -j DSCP --set-dscp 10
-A RULES-DEVICEX -m connmark  --mark 11  -j DSCP --set-dscp 11
-A RULES-DEVICEX -m connmark  --mark 12  -j DSCP --set-dscp 12
-A RULES-DEVICEX -m connmark  --mark 13  -j DSCP --set-dscp 13

-A POSTROUTING -s 192.168.5.100 -j RULES-DEVICEX # <-- 192.168.5.100 is
    DEVICEX IPv4 address!
```

References

[1] F. Sanchez and D. Brazewell, "Tethered linux CPE for IP service delivery", in *Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015, London, United Kingdom, April 13-17, 2015*, 2015, pp. 1–9. DOI: 10.1109/NETSOFT.2015.7116166.

[2] I. D. Cardoso, "Network infrastructure control for virtual campuses", Master's thesis, Universidade de Aveiro, 2014.

[3] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing", Gaithersburg, MD, United States, Tech. Rep., 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2206223.

[4] ETSI, *Network functions virtualization (nfv): architectural framework, technical report etsi gs nfv 002 v1.1.1*, Oct. 2013.

[5] ——, *Network functions virtualisation – introductory white paper*, Oct. 2012.

[6] ——, *Network functions virtualization (nfv): use cases, technical report etsi gs nfv 001 v1.1.1*, Oct. 2013.

[7] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento, "Cloud4nfv: A platform for virtual network functions", in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, Oct. 2014, pp. 288–293. DOI: 10.1109/CloudNet.2014.6969010.

[8] J. Halpern and C. Pignataro, *Service function chaining (sfc) architecture*, RFC 7665 (Informational), Internet Engineering Task Force, Oct. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7665.txt.

[9] P. Quinn and U. Elzur, "Network service header", IETF Secretariat, Internet-Draft draft-ietf-sfc-nsh-01, Jul. 2015. [Online]. Available: https://www.ietf.org/id/draft-ietf-sfc-nsh-01.txt.

[10] C. T. Laboratories, "Virtualization and network evolution", Cable Television Laboratories, Technical Report VNE-TR-SDN-ARCH, Jun. 2015. [Online]. Available: http://www.cablelabs.com/specification/sdn-architecture-technical-report/.

[11] K. Yogo, "Service chaining for nw function selection in carrier networks", NTT corporation, PoC Interim Report, May 2014. [Online]. Available: http://nfvwiki.etsi.org/index.php?title=Service_Chaining_for_NW_Function_Selection_in_Carrier_Networks.

[12] M. Kloberdans, *Virtualizing the home network*, 2015.

[13] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Performance of multi-tenant virtual networks in openstack-based cloud infrastructures", in *Globecom Workshops (GC Wkshps), 2014*, Dec. 2014, pp. 81–85. DOI: 10.1109/GLOCOMW.2014.7063390.

[14] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "Openanfv: Accelerating network function virtualization with a consolidated framework in openstack", in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: ACM, 2014, pp. 353–354, ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2631426.

[15] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Dynamic chaining of virtual network functions in cloud-based edge networks", in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, Apr. 2015, pp. 1–5. DOI: 10.1109/NETSOFT.2015.7116127.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control", in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: ACM, 2014, pp. 163–174, ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626313.

[17] B. Martini, F. Paganelli, A. Mohammed, M. Gharbaoui, A. Sgambelluri, and P. Castoldi, "Sdn controller for context-aware data delivery in dynamic service chaining", in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, Apr. 2015, pp. 1–5. DOI: 10.1109/NETSOFT.2015.7116146.

[18] M. Car, "Openstack service function chaining interface", Master's thesis, Universidade de Aveiro, 2015.

[19] Huawei, *Enabling agile service chaining with service based routing.* [Online]. Available: http://www.huawei.com/ilink/en/download/HW_308622.