POLITECNICO DI TORINO

COLLEGIO DI INGEGNERIA ELETTRONICA,
DELLE TELECOMUNICAZIONI E FISICA

Master Degree in Communications and Computer Networks Engineering

Master Degree Thesis

Automating deployment of high speed software routers inside Virtual Private Clouds

POLITECNICO
DI TORINO

Supervisors:
prof. Paolo Giaccone
prof. Dario Rossi

Candidate
Francesco Spinelli

Academic Year 2017 -2018

Ai miei genitori, primi sostenitori di ogni mia scelta, a mia sorella, che mi é sempre stata accanto nei momenti piú difficili e ai miei amici che mi hanno accompagnato nel corso di questa avventura

Summary

In recent years, Cloud Computing has emerged as one of the most predominant paradigm in the ICT world, together with NFV (Network Function Virtualization) and SDN (Software Defined Network). For instance, services used every day by millions of people such as *Dropbox*, *Google Drive*, social networks as *Facebook* and *Twitter*, but also companies, rely on Cloud Computing, since it allows to perform outsourcing, minimizing hence part of the costs due to to buy and maintain expensive computer infrastructure. In this environment, Software Routers appear and, among them, Vector Packet Processor (VPP), which is a framework for building high-speed data plane functionalities in software [37]. VPP exploits kernel-bypass techniques and its main novelty is the processing of batch of packets, instead of processing packets one by one, allowing to have better performances. Inside this context, it would be interesting to have VPP inside an Amazon's Virtual Private Cloud (VPC) [17] with respect to the common Amazon's routers. This study hence consist, firstly, in automating the deployment of VPP using a cloud orchestrator tool: Terraform [28]. Secondly, repeating the same configuration in different Amazon's Region taking advantage of our script, we connect them together through the new Segment Routing version 6 (SRv6) protocol and finally we perform several experimental measurements. The thesis is structured as follows:

Chapter 1 dwells on the main topics this work touches: Cloud Computing, SDN, NFV (especially Software Routers and VPP). It introduces a new protocol VPP implements in its 18.04 release: *Segment Routing Version 6*, which is the IPv6 version of Segment Routing protocol, and finally, we briefly explain the contributions of our work.

Chapter 2 describes the first goal of the thesis: the VPP deployment inside a Virtual Private Cloud in Amazon's cloud infrastructure and, afterwards, the efforts on automating this process. Furthermore, we briefly review the related work in cloud automation and which Cloud Orechestration tools we considered. Afterwards, we shift our attention on the selected tool: Terraform. We describe its strong and weak points and we quickly overview some of its features, with a brief hands-on on the code developed.

Chapter 3 describes the methodology we use for our experiments: what meta data we choose and the tools to evaluate them. We overview the related work about performance measurements in the Amazon's cloud environment, and finally we describe the experimental scenarios we built taking advantage of our automating script, with also a quick overview to the technical characteristics of the instances.

Chapter 4 shows some of the most meaningful experimental results we obtained and our comments about them. Furthermore, we show several figures about the Time To Live, Round Trip Time and the Throughput values achieved, with and without VPP presence.

Finally, Chapter 5 sums up the results obtained and explains some of the possible future work.

In the Appendixes, we put the commands used to install VPP, the whole Terraform script and more figures obtained from our performance measurements.

Contents

Chapter 1

Introduction

1.1 What is Cloud Computing

In the last decades, Cloud Computing has gathered more and more interest inside the scientific community and, furthermore, it also started to affect the life of many people around the world. Services used every day by million of people such as *Dropbox*, *Google Drive*, but also social networks as *Facebook* and *Instagram* rely on Cloud Computing since they allow users to upload and, for example storage, their photos or documents. Storage is one of the Cloud Computing's goals but it is possible to exploit it also to run, among other things, scientific experiments, even belonging to very different fields [32]. These are some of the motivations why Cloud Computing is becoming one of the most prominent paradigm in ICT (Information and Communication Technology) field: for instance it will have a prominent role also in Mobile Edge Computing inside 5G networks [29]. However, it is still difficult to understand deeply how the technology and the infrastructure behind works. The name "Cloud" is a perfect metaphor that reflects well what is the main characteristic of this technology: is not important how the infrastructure is physically build, which hence could remain nebulous to the user that exploits it, but only the type of service it could offer.

The term *Cloud Computing* becomes popular when the Web division of Amazon, called *Amazon Web Services* or more generally *AWS*, released its own public Cloud service in 2006 [7] with its first data center deployed in North Virginia. Back in 2006, it was the first company to release a public Cloud service. Nevertheless, the term *Cloud* was already used in the late 60s, referring to the first ARPANET network, where all the infrastructure at the center of network was depicted within a cloud. In Figure 1.1 we shown an original ARPANET draw from the US Patent office. Later, in 1996, this term appeared also on a Compaq Document [15], which

has been desecrated just few years ago.

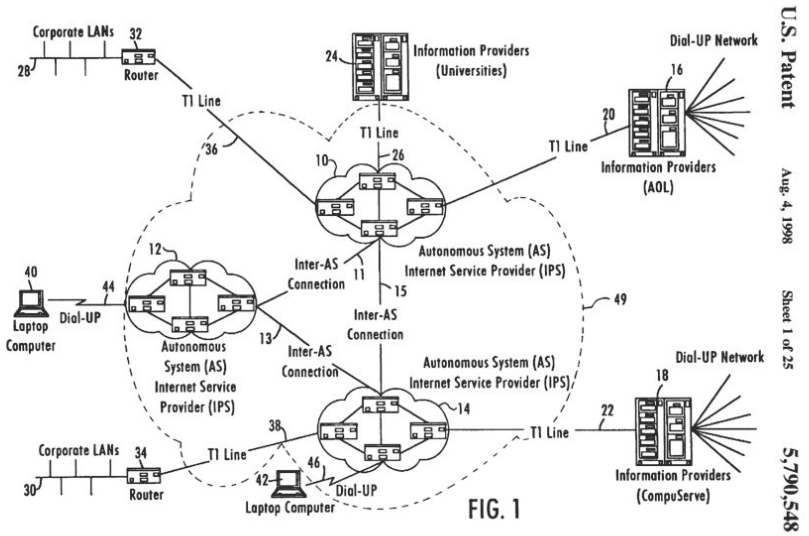

Figure 1.1: Draw of Arpanet, US Patent Office. Image taken from [57]

After Amazon, quickly all the other principal IT competitors commercialized their own Public Cloud version: Google started to offer it in 2008, with the name of*Google Cloud*, while Microsoft in 2010, with *Microsoft Azure*.

After having described at very high level what is Cloud Computing, we want to focus now on the official definition proposed by NIST, which gives us some hint of its main features. According to NIST, Cloud Computing is:

"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [42].

In this definition, we could find which are the most important features of Cloud Computing and we could summarize them in few key points (highlighted also in [42]):

- *On-demand self-service*: An user could provision computing capabilities, such as server time allocation or storage, and this operation is done without

the need of human interaction. For instance, once a user is registered in one of the Public Cloud services, it accesses it and provision the resources simply through a console management accessible via a normal Web Browser.

- *Broad network access*: Resources are available over the network and they are accessed by a broad of different client platforms. This means that the user could access them through a laptop, a mobile phone or through a workstation.
- *Pooling of resources*:The resources owned by the Cloud Provider are pooled to serve multiple consumers using, for instance, a multi-tenant model. The latter means that a single instance could serve multiple clients (which are called Tenants). Another way to perform resource sharing is using Virtual Private Clouds, which we use in our work. Moreover, physical and virtual resources are dynamically assigned according to consumer goals.
- *Elasticity*: Resources need to be elastically provisioned and released, and in some cases this operation has to be performed automatically. From the point of view of the user the resources should appear unlimited and accessible at any time. .

A key technology which made Cloud Computing's success is Virtualization [56]. At a glance, it allows to have within the same physical computing resource (which is called *Server*) several virtual resources running on its top, called *Virtual Machine*. Each of them could perform tasks independently from each other, with the physical hardware that is shared among all the virtual instances.

A Key aspect of Virtualization is *independence*: this means that each virtual machine (from now on VM) created on top of the same physical hardware is not aware of the presence of the other VMs and hence every resources have the illusion that they are the only user of the physical resource.

An example, which happens inside Cloud Computing environment (for instance in AWS [4]), is that several VMs with different Operating Systems could be virtualized on top of the same physical hardware. Hence, to share the physical resource, an entity called *Hypervisor* is used. Hypervisor is a software or process that creates and runs virtual machines. There are two types, shown also in Figure 1.2:

- *Bare-metal Hypervisors*: they control the hardware and manage the several guest operating systems. An example is Xen Hypervisor [13].
- *Hosted Hypervisor*: They run on OS just like any other program. An example is VirtualBox [39].

Amazon Cloud environment uses Xen hypervisor [13]. Virtualization is very important mainly for two reasons: it allows to speed up IT operations (i) and

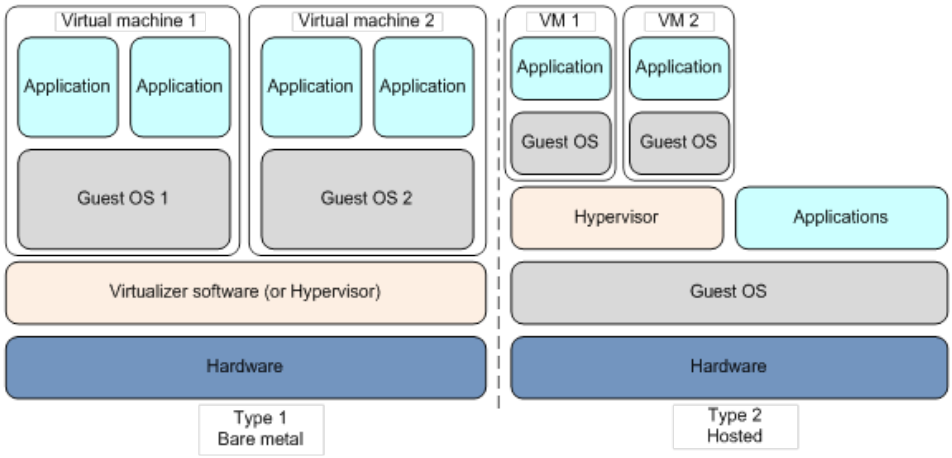

Figure 1.2: Two different Hypervisors in action. Image taken from [2]

reduce cost by increasing the infrastructure utilization (ii). In fact, using a very high level example, it is more convenient for a Cloud Provider to have 3 different OS, one independent from each other, running within the same hardware instead of having just one OS that use all the available physical hardware.

1.1.1 Service Models

Three different service models exist within the Cloud Computing environment, with which the clients and providers could deal with. They have been described by NIST [42]:

- *SaaS*: Software as Service
- *PaaS*: Platform as Service
- *IaaS*: Infrastructure as Service

The first one means that the user have access to applications over the Internet and he has not control on the physical resources (such as servers, OS, hardware) underneath the applications (he can not decide which server could storage his photos for example). He can access those applications through a Web Browser and hence the user does not need to install anything in his computer and this simplifies the maintenance and support. It is usually priced on a pay-per-use base or with a monthly or yearly subscription fee. If we look at a company point of view, SaaS cloud reduce costs by outsourcing software maintenance and hardware.

However, SaaS could have one drawback: it stores the users' data on the cloud provider's server therefore leading to unauthorized access to the data (the most recent example is the Cambridge Analytica scandal [40]) To accommodate a large number of cloud users cloud applications can be multi tenant, meaning that any machine may serve more than one cloud-user organization and we already saw that this is one of the feature of Cloud Computing. Some examples are *Office 360, Google Docs, Gmail, Dropbox*.

In the second one instead, platform layer resources are provided. The provider typically develops toolkit and standards for development and channels for distribution and payment. In the PaaS models, cloud providers deliver a computing platform, typically including operating system, programming-language execution environment, database, and web server. Developers can develop their software solutions on a cloud platform, avoiding the costs and the management of the underlying hardware and software layers. This means that the user can use the framework provided by the Cloud Provider (the company which offers Cloud services) to build and deploy applications. Some examples are *Google App Engine* and *Microsoft Windows Azure*.

Finally, the third one provides on-demand infrastructural resources (Virtual Machines for example). With this option, the user can decide the details of the underlying network infrastructure, choosing for instance which resources instantiate and in what location put them (however all these actions are limited by the degree of freedom imposed by the Cloud Provider). An example, is *Amazon EC2*, which is also the infrastructure used in this study. Here Virtualization, and then the use of an Hypervisor, is very important since it can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements. These resources are inside Public Cloud Providers Datacenters and the users after instantiated the resources can install operating-system images and the application software they need. Normally bill costs depend on the amount of resources allocated and consumed. For Instance, in Amazon EC2, the costs depends on the type of Virtual Machine instantiated (the more powerful it is, the more it will cost) and it is on per hour basis fee. Table 2.1 reports an example of the cost for an Amazon m5 instance type. Furthermore, also Storage could add additional costs.

1.2 Software Routers

Internet architecture is becoming more and more complex. It is vertically integrated [35], which means that the control and data planes are coupled together inside each single network device. The control plane is responsible for handling the traffic, for instance it tells where to route the packets, whereas the data plane just

forward the traffic based on the control plane decision. Nowadays this integration increases the complexity of managing the network infrastructure: in particular because, with distributed control plane, it is difficult, for example, to understand the state of the network and its history. Moreover, every router or switch has a different hardware architecture depending on the Vendor specifications, which reacts only to vendor-specific commands. The last but not the least, the increasing size of the network architecture leads to a more difficult management of reconfiguration and policies in case of loads and faults [35].

Therefore, in the last years, two new technologies have risen to avoid, or at least to decrease, the growing complexity of the network:

- *SDN*, which stands for Software Defined Network
- *NFV*, which instead is Network Function Virtualization.

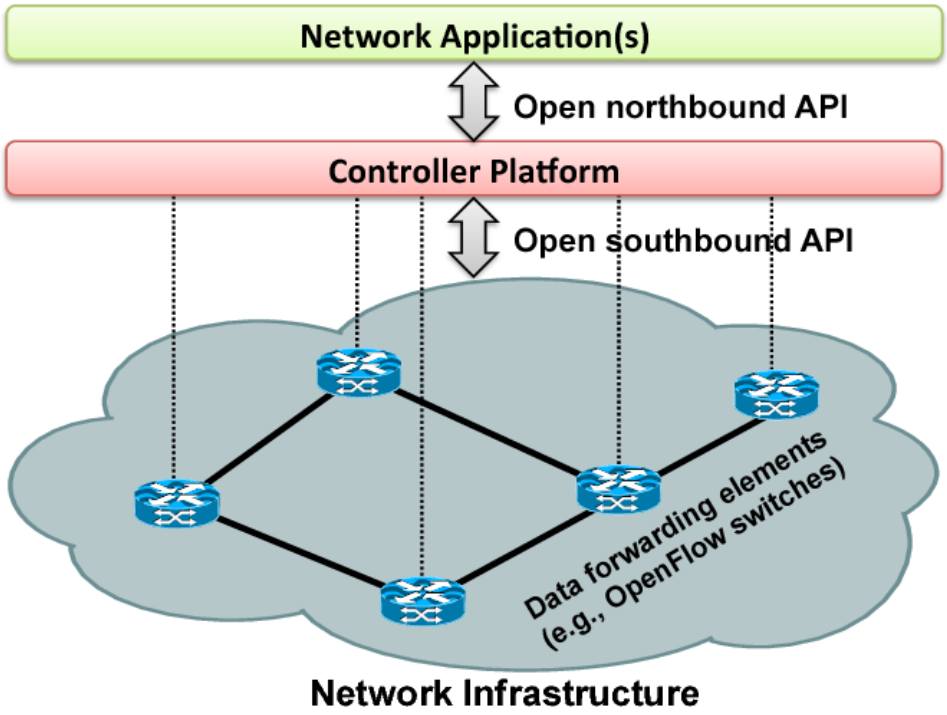

Figure 1.3: A view of a SDN architecture. Image taken from [35]

SDN is an emerging paradigm, which consist on the separation between the control and data plane. At a glance, routers and switches are now devoted only to

forward elements, where instead the control plane is centralized in a SDN controller moved outside of the network, which now controls a set of routers and switches [35]. SDN proposes, as one of the main novelties, a *Flow-based forwarding decision* instead of basing on a destination-based forwarding . A *flow*, in this case, is a set of packet field values acting as a match rule and a set of actions that operate on all packets belonging to the same flow [27]. This definition allows hence to unify the behaviour of routers, switches, firewalls, load balancers and traffic shapers.

Moreover SDN allows to have a unique abstract view of the topology of the network and this control logic is moved to a entity, which is external from the network, called *SDN Controller*. The SDN controller can be controlled by network applications. It is a software platform, which runs on commodity server and it is logically centralized. There are two types of API, which belong to the SDN controller interfaces:

- *Northbound Interface API*: this API is available to developers and allows to abstract the low-level instructions to the forwarding devices. They can be develop with any programming language.

- *Southbound Interface API*: allow to access the switches and send them commands and this means that allow the interaction between the control and data plane.

Figure 1.3 shows at high level a possible SDN structure. A SDN version called OpenFlow [54] was proposed in 2006. The physical structure can be imaged as the one portrait in Figure 1.3 but, when a packet enters in the first router of the network, some messages based on TCP connections are sent to the SDN controller. The first one, called *Packet-In*, is sent from the switch to the controller and it asks the latter what should do with that packet. The second one, called *Packet-out*, is sent from the controller to the switch and tells the action that has to be done (for instance to send the packet to a specified port on the switch). Moreover, a third message is sent from the Controller to the switch, which is called Flow Mod and allows to modify the flow tables of the switch, telling the match-action rule to install to the switch. The flow tables are tables with inside a match-action rule. The latter tells that if a packet matches with or a binary exact match or a ternary match, it has to follow the action connected to that match, that could be drop, forward, modify or goto another table. SDN is used for example in Google's Datacenter [50].

The second paradigm is called Network Function Virtualization (NFV). In network architecture the presence of many different vendor hardware leads in increasing costs of energy, skills to design and integrate different hardware and, in the end, increasing of capital investment [21]. Furthermore, specialized hardware has

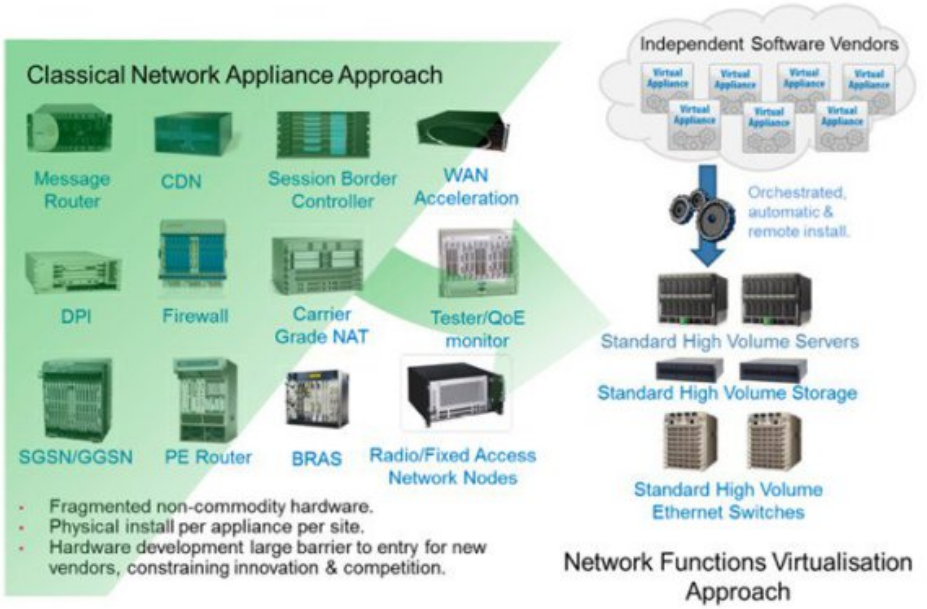

Figure 1.4: A view of the differences between standard network infrastructure and NFV. Image taken from [21]

an high design cost and life-cycles are becoming shorter and shorter. Therefore for network operators is becoming difficult to generate high revenues.

NFV consists on taking advantages of the virtualization technologies and move Network functions such as Firewalls, Load Balancers, that need specialized hardware, into software which can run with underneath a general purpose hardware. As highlighted in Figure 1.4, in the classical network approach we have, for every appliance, a non commodity hardware. However, with a NFV approach, an independent software vendor could orchestrate, automate and remotely install its software (which is just a virtual appliance) into a Standard High Volume Server, which could be then located in Data-centers. This allows to reduce equipment costs and power consumption, increase the duration of hardware and perform multi-tenancy. Moreover, it could increase the speed of *Time to Market* by minimizing the network operator cycle of innovation and the openness of the software. Finally, this allow services to be rapidly scaled up/down and encourage more innovation [21]

These two paradigms, SDN and NFV, are complementary. They could exist

without the other, even though they achieve a greater value combined. For instance, at a very high level, SDN could be used to route the traffic across different NFV services: this could lead to the so called *Service Chaining*. Hence, to summarize, in these days the software implementation of networking stacks is becoming more and more important as demonstrate the rising of development of *Software Routers*. The latter are software-based network elements, which are capable of advanced data plane functions with underneath general purpose hardware. One of the first example of software router is the *Click Modular Router* [33]. Basically, some of the network-related functionalities, performed by specialized hardware, are instead performed by some software functions, allowing the creation and the connection of software functions which can be compiled by a generic hardware. This allows to have a general purpose operating system hardware underneath the software.

However, this technique exploits some disadvantages: for instance, to obtain high performance, most of the high speed functionalities have to be placed close to the hardware, and this means that a kernel module has to be implemented, even though this approach adds overhead to the execution, since a user-space applications needs to perform system calls and hence they need to use the kernel. However, in the last years, recent improvements in transmission speed and network card capabilities were made and now a general kernel-stack could be too slow for processing packets [37] and [6].

In these days, techniques that implement high speed stacks bypassing operating system kernels (also called kernel-bypass techniques) [37] exist. In other words, the hardware is abstracted directly to the user-space. This could be done through different procedures, for example using *netmap* [46], Intel Data Plane Development Kit (DPDK) [24] or modular frameworks for packet processing, like VPP [22].

In literature are present three different types of software frameworks for high-speed packet processing based on kernel-bypass techniques:

- *Low-level building blocks* Some examples are DPDK [24] and netmap [46]. They support kernel by pass features and high speed I/O.

- *Purpose-specific prototype*. These are prototypes based on restrained set of capabilities such as IP routing, traffic classification or name-bases forwarding or transparent hierarchical caching. They could also use GPUs.

- *Full-blown modular frameworks*: They are close to VPP and one example is the Click modular router [33] and its extension called FastClick [6]

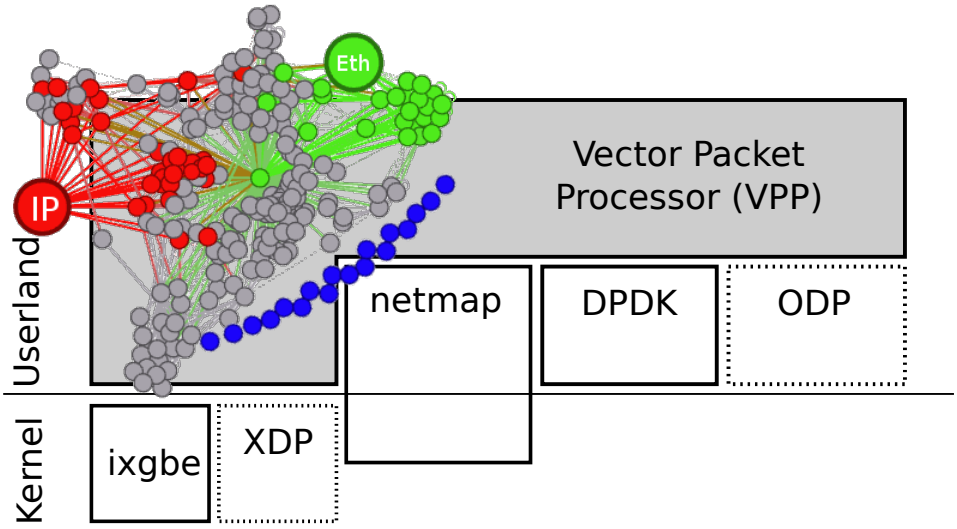

Figure 1.5: VPP Processing tree. Notice that VPP runs on top of DPDK, netmap. Image taken from [37]

1.2.1 VPP

Vector Packet Processor (VPP) is a framework for building *high-speed data plane functionalities* in software, taking advantages of general-purpose CPU architectures [37]. Firstly released under a US patent, in 2011 [16], and then under the context of the Linux Foundation project "Fast Data IO" (FD.io) [25]. VPP is a mature software stack, which could be implemented in several scenarios, ranging from Virtual Switch in data-center to inter-container networking.

VPP's processing holds a *run to completion* model [37]. First of all, it polls a batch of packets using for instance DPDK and then it processes only when the batch is full. VPP is written in C and it comprises a set of low and high level libraries, which are the *main core* of the framework. Some examples are *l2-input* or *ip4-lookup* An user could add extensions, which are called *plugins*, which can add or replace functionalities. The two of them form the *forwarding node graph*, which describes the possible paths a packet can follow during the VPP process. In VPP there are 3 type of nodes:

- *Process* nodes are simple software functions running on the main cores and reacting to timers and events.
- *Input* nodes manage the initial batch of packets.

- *Internal* nodes are only traversed after an explicit call.

Figure 1.5 shows the VPP position in the Operating System. It is worth notice that VPP operates completely in the User Space and it leverages of the kernel-bypass blocks such as DPDK and netmap, using them as Input/Output nodes.

However, VPP's main novelty is the processing of vector of packets, instead of processing them one by one. As seen before, Input nodes produce a vector of packets to process, then the software pushes the vector through the directed node graph, subdividing the vector when needed, until it has been completely processed [37]. Figure 1.6 portrait our statement.

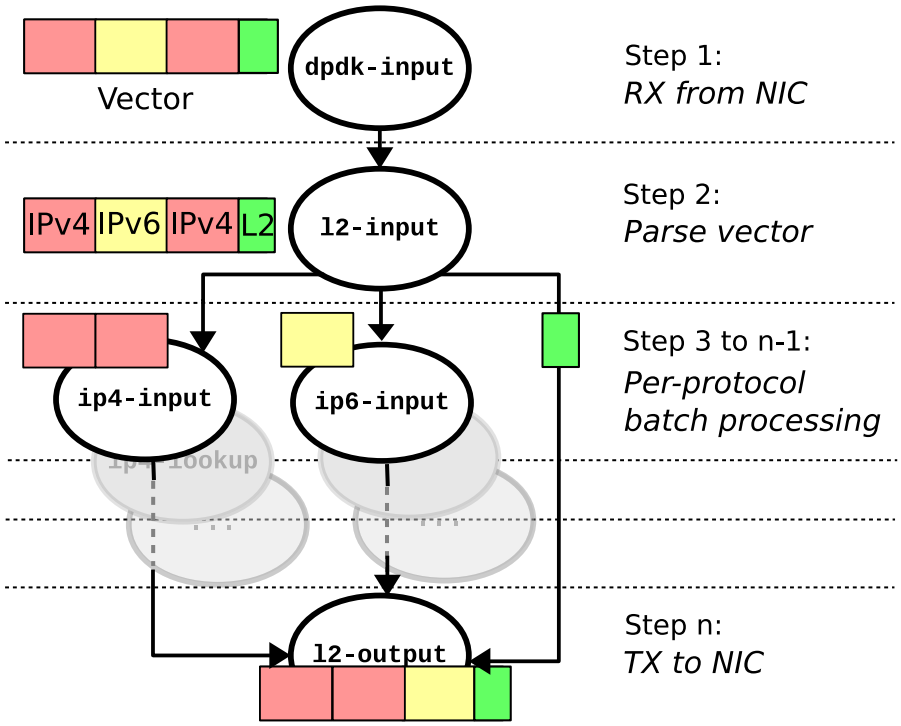

Figure 1.6: VPP Processing graph. Image taken from [37]

Furthermore, as also highlighted in Figure 1.6, not all packets follow the same path. The advantage of using batch of packets are that, firstly, the framework overhead is shared among all the packets inside the same vector, and secondly, the CPU is used more efficiently. In fact VPP optimizes the CPU's instruction cache, with only the first packet that heats up the cache, while the others not.

In our work, we use VPP version 18.04 and 18.07.

1.3 Segment Routing version 6

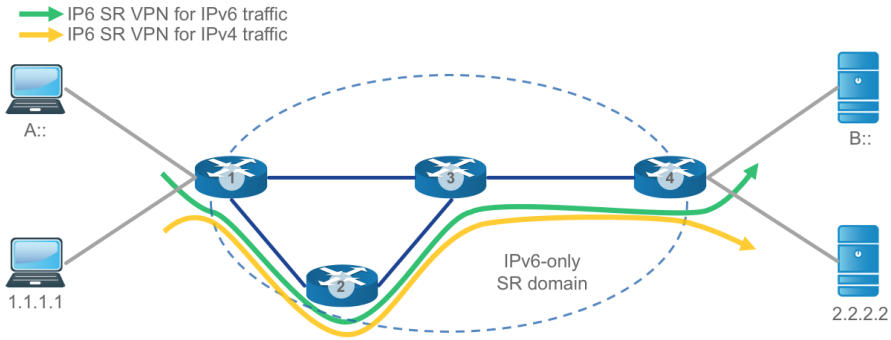

Figure 1.7: Example of SRv6 utilization. Not necessary the shortest path is applied. Image taken from [23]

Since Cloud Computing is becoming more and more important, without forgetting the growing presence of SDN and NFV, new services started to rise up: among them, Service Chaining appeared. The latter rose up thanks to SDN and, basically, it creates a chain of network services, connecting them within a virtual chain. It is very useful since it enables to use a single network connection.

Service Chaining is becoming popular because it changes how to handle traffic flows, automating the setting up of virtual network connections [12]. For instance, an SDN controller. depending on the different traffic flows, could "*take a chain of services and apply them to different traffic flows depending on the source, destination or type of traffic*" [12]. Service chaining could be used for instances in Data-centers.

Service Chaining is performed in many ways. This work use *Segment Routing version 6* [9], which is the IPv6 version of Segment Routing (from now on we refer it as SRv6) since it is supported by VPP 18.04 and Amazon AWS is one of the few public cloud provider, together with Microsoft *Azure*, that allows to allocate IPv6 addresses. It is worth noticing that SRv6 could be applied to either MPLS or IPv6 protocol. Figure 1.7 portrait an example of how SRv6 works and how it is different from the other protocols: for instance, instead of choosing the shortest path, the packets follow a different path, based on the commands present in their header. Basically, at a glance, SRv6 tells a source node to *"steer a packet through a list of instructions, called segments"* [9]. Each one of these instructions is a

function (for instance, decapsulation into a IPv4 packet, some specific IP table lookup functions or just a forwarding instruction) which are performed when the packets arrive at the destination node node. Then, the instruction is deleted and the packet is forwarded to its next destination.

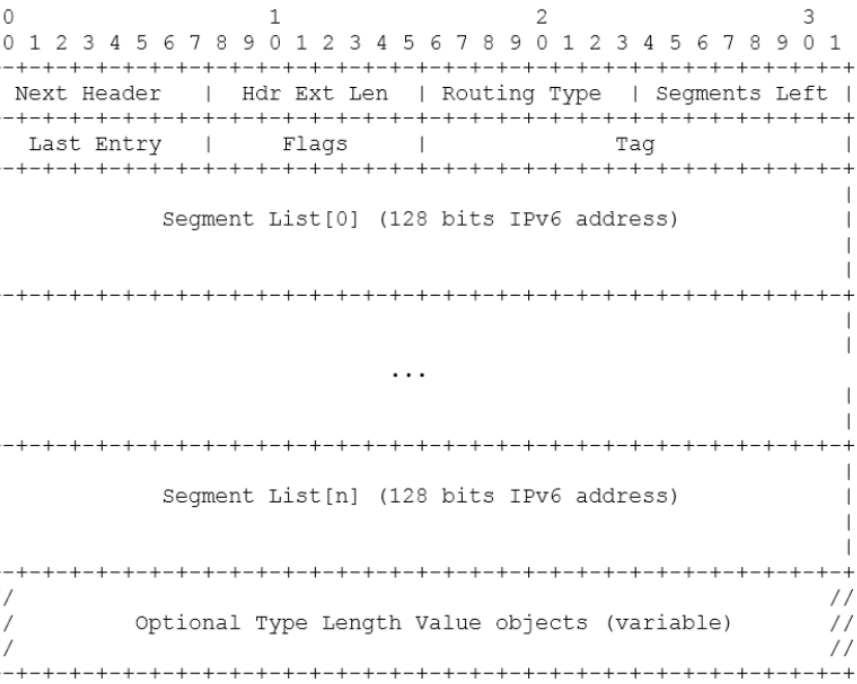

Figure 1.8: Segment Routing packet header. Image taken from [10]

Going deeper in details, Figure 1.8 shows the new SRv6 packet header, which it is comprised inside the IPv6 Routing Extension header and it is called *Segment Routing Header (SRH)*. Inside the latter the SRv6 instructions called *segments* lay, encoded as IPv6 addresses. The current segment processed, in Figure 1.8 *Segment List[0]* is the IPv6 Destination Address of the packet, while the following segment to be processed is indicated in the Segments Leg field of the SRH. After a segment is completed, the header is updated to point to the next segment and the new active segment is copied in the Destination Address field of the IPv6 header. On an SRv6 node, the segments are called *LocalSIDs*. The latters are associated with

a processing function on the SRv6 node, which may range from advancing to the next SID in the SRH (forwarding) up to user-defined behaviors [9].

Segment Routing version 6 allows hence to have Service Chaining but also multi-cloud overlays, which means that it is able to interconnect several public cloud-provider regions between each other, having control on the IPv6 transit and this is important for our work since after automating VPP deployment we would like to perform some experimental measurements.

1.4 Our contribution

Amazon Web Services offers in its own cloud infrastructure [4], a new service called *Virtual Private Cloud* [17] (from now on *VPC*). We are going to explain it deeper in the following section but for now we can say that with this service, at a glance, each user could have its own private cloud, which is completely separated from the other ones and accessible only by the user who create it. It would be interesting to investigate if VPP could be implemented inside a VPC and its deployment automated. This implementation is motivated principally for three reasons:

- *Performance*
- *Security*
- *multi-cloud connection through SRv6*

The first one because since VPP is an high speed software router [37] it should have higher performances than the router which is automatically attached in every VPC. Instead the second and third motivation because, with SRv6 and VPN as a further layer of security, we could steer the packets where we want in a multi-cloud environment. The first part of the work is to install VPP software into a virtual machine inside a VPC and try to set it as main router of the VPC using, among other things, DHCP protocols [19] and the route tables provided by the Amazon Console Management. After the creation and deployment of a first VPC, we create a second specular VPC and then we connect them together using SRv6. Our second step is to automate the VPP deployment inside the VPC with a Terraform script [28], chosen among other competitor for its flexibility and simplicity.

Afterwards, we perform some Experimental Measurements, both intra-cloud and inter-cloud, to see the benefits of the VPP deployment. At a glance we notice that the performances obtained with or without the VPP presence are similar. Before starting the measurements we decided the methodology and the metrics to evaluate, together with the tools and it is worth noticing that the passage from paper to actual experiments led to few changes in the structure of the experiments.

In literature we found some works regarding measurements, even though most of them are now old and show results which are not compatible with which we found, and this allow us to see that, for example, the Cloud environment changes rapidly.

Furthermore, most of the the previous work, such as in [41] [38], concentrates only with intra-cloud measurements and as far as we know, we did not find a work which contains the same amount of different scenarios considered.

Hence, to summarize, the novelties of this work are (i) deploy an high performance software router inside one of the largest Cloud Provider infrastructure, with all its new features which were not present before in an Amazon environment, such as Segment Routing version 6. Afterwards we automate its deployment (ii) and we performed several experimental measurements. The latters are a complete novelty in the literature and, as far from our knowledge, no one performed such steps.

Chapter 2

Automating VPP Deployment

In this chapter we describe the processes to deploy and automate VPP inside an Amazon's *Virtual Private Cloud*, and then how we connect VPP together with another VPP router, deployed in a different VPC. In the first section we review in details what is a VPC, then in the second one we overview several previous works on automation in the Cloud. Afterwards, we focus on what was the preliminary work, which means when VPP was firstly installed and configured, and then in section 4 we describe how the automating took place and why and how we used Terraform. Finally, in the last section some issues, risen up during the script development are highlighted.

2.1 Virtual Cloud Private

A Virtual Private Cloud [17] is, citing the Wikipedia page which summarize well its most important features:

> *"an on-demand configurable pool of shared computing resources allocated within a public cloud environment."* [62]

This means that, in other words, every AWS user could define its own Virtual Network and using it like it is his data-center, but with the advantage of using a Public Cloud Provider infrastructure with all of its benefits. VPCs make their first appearance in Amazon's infrastructure in 2009 [4]. However they are present also in other Competitors such as *Google Cloud Platform* and *Microsoft Azure*. A VPC is logically isolated from the other VPCs and this isolation is achieved through the allocation of private IP addresses. In every VPC it is possible to specify a set of

characteristics: among the others, the IP addresses range of the VPC,the presence of sub-nets, deploy security groups, which are basically the firewall of the VPC and finally an user could define the route table to forward the packets.

The VPC IP addresses range is very important since it defines how many resources an user could instantiate inside it. The subnets are range of IP addresses assigned inside the VPC, with their addresses range that are a subset of the VPC's IP address. Subnets are necessary to instantiate AWS resources and depending on the goal of the user, they could be *public*, where *Elastic IP address* can be allocated if the resources are connected to the Internet, or they could be *private*.

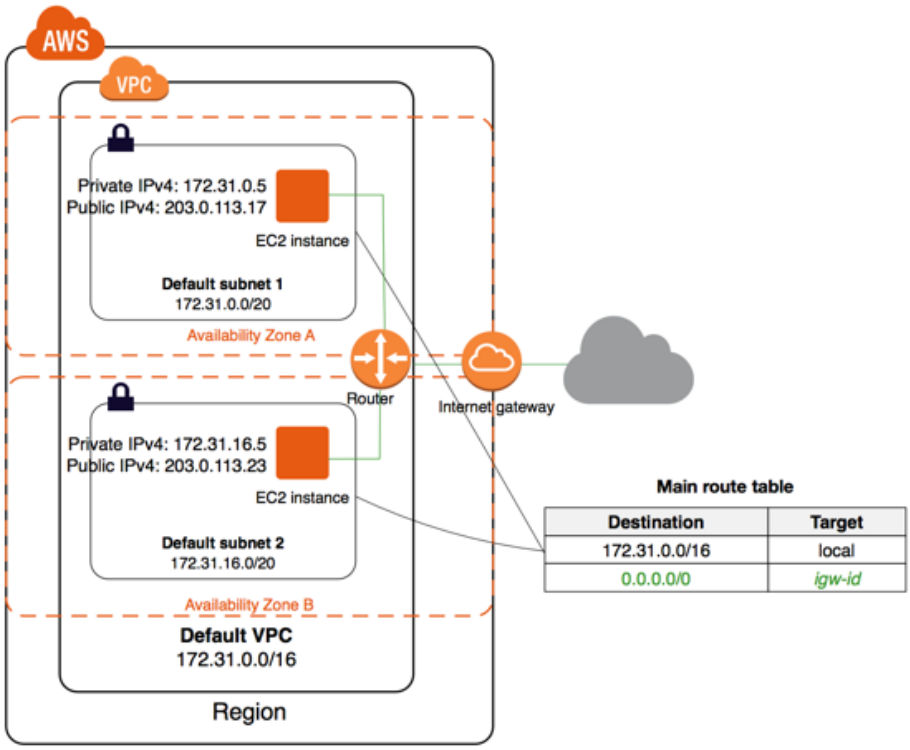

Figure 2.1: An example of VPC's. Notice that the same VPC could be deployed across multiple Availability Zone. Image taken from [17]

In Figure 2.1 we show the VPC structure with all of its characteristics. As main features, highlighted in the previous paragraphs, we see that the VPC has its own IP address' range, called *CIDR* (Classless Inter-Domain Routing), based on IPv4.

However, it is possible to define a range of IPv6 addresses, if necessary. Inside the VPC, an user could instantiate as many sub-nets he wants and it is worth notice that the sub-nets could be placed in different Availability Zones (in Figure 2.1 only two subnets are present, belonging to two different AZ). In our configuration we use 3 subnets, where one of them has IPv6 addresses. Each instance could also have two type of IP addresses, depending if it is connected to the Internet or not. When we define a sub-net, the first 4 IP addresses can not be used, since they are reserved for Amazon's purposes. Furthermore also the broadcast address can't be used since Broadcast traffic is not allowed inside a VPC. The sub-nets are connected within the VPC with a a router which also contain a route table, to forward the packet depending on the destination. Summarizing, with the VPC model, an user gain the ability to:

- Assign private IPv4 addresses to the AWS resources, which remain constant even though the instance is turned off
- It is possible to associate an IPv6 CIDR block to the VPC and therefore assign IPv6 addresses to the instances
- Assign multiple IP addresses to the AWS resources (see below)
- Define network interfaces, called *ENA*, and attach one or more network interfaces to the instances, assigning furthermore IP addresses
- Control, with security groups (name that Amazon gives to firewalls), the outbound and inbound traffic from the AWS resources
- Add an additional layer of security, with network access control lists (ACL)
- Run the instances on single-tenant hardware, avoinding hence multi-tenancy

Every VPC comes with an Internet Gateway but only the user decides whenever a resources could reach the internet. In this case, every instance has two IP address, one private and one public, where the latter is called *Elastic IP*. Another interesting feature of VPCs is that they could be shaped among several Availability Zones (AZ). Amazon has its data-centers hosted in multiple locations across the world (some of them are for example in *Oregon, London, Paris, Sidney*). Inside each regions, different Availability Zones are deployed, which are connected together (as shown in Figure 2.2). This redundancy is especially helpful against fault tolerance and a VPC belonging at the same to different AZs.

2.2 Related Work on automation

In literature cloud orchestrator tools have gathered only recently attention, since they rose up together with the DevOps paradigm. However, in literature only a

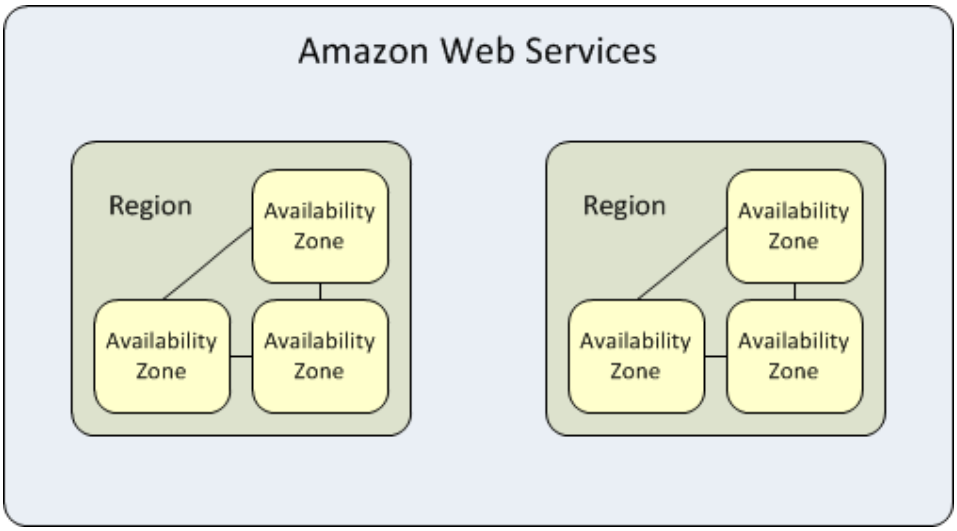

Figure 2.2: An example on the differences between Region and Availability concept. Spawning the same instance in two Availability Zones allows to be covered in case of fault issues. Image taken from AWS website [4]

few works discuss Cloud Orchestrator tools such as Terraform. In [58] for instance, the authors investigate how many DevOps artifacts exist, and how their differences lead to several problems into the deployment and automation of cloud applications. For the authors the solution is to create a framework with *TOSCA* (Topology and Orchestration Specification for Cloud Applications). The latter is a standard language which describes a topology of cloud-based services with their components and relationships.

Instead in [34], the authors have created from scratch an open source cloud orchestration and management framework for heterogeneous multi-cloud platform called *Occopus*. The main goal of *Occopus*, as highlighted by the authors, is to create a complete Cloud Orchestrator, with also looking at many indicators such as *portable descriptors, flexible extendable architecture, health-monitoring, life-cycle management, scaling and error handling.* With Comparison, Terraform [28] is limited by the author's opinion, even though is considered very powerful, since it focuses only on the deployment of the infrastructure, and it does not support features such as *life cycle-management,scaling and error-handling.* They also have noticed that from all the cloud Orchestrator tools, Terraform is the most difficult one to learn, with a non easy steep curve. The same missing aspects are highlighted also Draxer et al work [20], where they notice that Terraform do not focus on

network function-specific needs, using as an example the *flexible forwarding rule*.

Terraform is also used in Callanan et al paper [11], where its role is to provide Infrastructure as Code code to their new Orchestrator, called *Environment Migration Framework*. Finally, in Astahna et all work [3] they use a data-driven approach to dynamically generate Orchestration Engine plugins, which could be later deployed with Terraform.

2.3 Preliminary work: first deployment

In this section we focus on the preliminary work done before automating the VPP deployment and hence how VPP is firstly installed and configured. In figure 2.3 we show the final configuration of our work scenario.

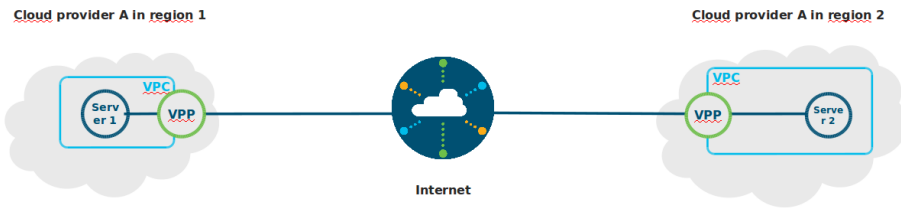

Figure 2.3: An example of our final configuration. Notice that inside the VPCs two Virtual Machine are present: one is our client/server for the experimental measurements and the other one contains VPP

As a disclaimer, all the following steps are also described in this wiki page created by author in the VPP wiki page [51].

Firstly, we need to define the VPC. In the creating process, we choose among several features such as the IPv4 block of the VPC (for instance *10.0.0.0/16*) and/or an IPv6 block, which would be instead provided by Amazon (an example could be *2a05:d01c:440:b000::/56*. Afterwards, we create two subnets (in total

there will be three subnets, since one automatically come with the VPC). Having three subnets will help to have separate work flow, since the virtual machines will have an network interface inside each subnet. We will use one subnet for management purposes and it will have an IPv4 range addresses (*10.0.2.0/24*): this also means that the network interface (from now on called NIC) attached to the Virtual Machines and belonging to this subnet, will have a private and public IPv4 address. This will allow the user to gain control of the virtual machines through SSH. The latter (which stands for Secur Shell [60]) is a network protocol used, for example, to log in remotely to a computer/server, providing a secure channel in a client/server architecture. The second subnet will be used to connect VPP with the client/server VM inside the VPC, and it will only have IPv4 addresses (the IPv4 address range could be *10.0.1.0/24*). Finally, the third one will be used to connect VPP towards the internet. Since inside it we use Segment Routing version 6 [9] to communicate with the other VPP machine belonging to a different VPC (as shown in Figure 2.3), this subnet will have also an IPv6 range addresses (for instance *10.0.3.0/24* and *2a05:d01c:a74:9c01::/64*).

After defined the subnets, we can now instantiate the Virtual Machines. In the console management we enter in the EC2 section, where we can manage our resources. Here we define the characteristics of the Virtual Machine we want to instantiate. Firstly, we choose the type of the AMI (Amazon Machine Image). An AMI [49] is a template that contains the software configuration (operating system, application server, and applications) required to launch the instance. One could choose from the standard ones provided by Amazon itself, for example *Amazon Linux 2, Amazon Linux AMI 2018.03.0, Red Hat Enterprise Linux 7.5*. In our work we use the standard Ubuntu distribution, *Ubuntu Server 16.04 LTS*. However, an user could also choose both a AMI created by himself (it is what we did later using Terraform), or an AMI sold in the AWS marketplace (like for example *pfsense Gate, Cisco Router CSR 1000v*). After performed this step, the user have to decide which instance type use. They are different, optimized to fit different use cases and they have varying combinations of CPU, memory, storage, and networking capacity. Of course, depending on the type of instances different costs are associated with. An user could choose free to use Instances type, such as the *t2.micro* or others which the cost vary depending on the hardware characteristics (such as the m5 type), remembering that their costs is per hour basis. In Table 2.1, we show how different types of Virtual Machine could lead to different costs.

Afterwards, an user could define some configuration details of the Instance. It is possible to choose the number of instances to create, in which VPC and subnets instantiate them. Moreover, if more subnets are present, an user could also create more NICs (Network Card Interfaces) to deploy inside the several subnets. An user could also decide how much storage to launch, the tags of the virtual machine and the setting of the security group [4]. A security group is a set of firewall rules

Table 2.1: An example of Amazon instances type. The more they have increasing performance, the more their cost per hour basis increases

Instance Name	Memory	vCPUs	Network Performance	Cost (hourly)
m5.large	8.0 GiB	2 vCPUs	High	$0.096
m5.xlarge	16 GiB	4 vCPUs	High	$0.192
m5.2xlarge	32 GiB	8 vCPUs	High	$0.384
m5.4xlarge	64 GiB	16 vCPUs	High	$0.768
m5.12xlarge	192 GiB	48 vCPUs	10 Gigabit/sec	$2.304
m5.24xlarge	384 GiB	384 vCPUs	25 Gigabit/sec	$4.608

that control the traffic for the instance. For example, if the user want to set up a web server and allow Internet traffic to reach the instance, it add rules that allow unrestricted access to the HTTP and HTTPS ports. In all of our configurations and experiments, the settings used allows to send and receive any type of traffic (*all traffic* option) Before launching the instance, an user must define a private Key (with .pem extension), which will form a key pair with the public key that AWS creates.

A key pair consists of a public key that AWS stores, and a private key file that final user store. Together, they allow to connect to the instance securely. Finally, the instance is launched. To connect to a instance via SSH an user should write on the terminal of his workstation:

```
sudo chmod 400 "keypair.pem"
ssh -i "$HOME/Terraform/Paris_Key/VM_Paris.pem" ubuntu@ec2
    ↪ -52-47-47-248.eu-west-3.compute.amazonaws.com
```

In our configuration scenario, inside the VPC we create two virtual machines, both with Ubuntu Server 16.04 LTS, but one with 3 NICs, each for every subnet (this would be the one where VPP will be) and the other one with only two (the virtual machine that will be our client/server). Let now focus on the second Virtual Machine, in which VPP is installed. We access the VM and in the command line we then type the following commands to install some initial packages. (As a disclaimer, these commands could be also found in a wiki page created by the authour [51]):

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
sudo apt-get install python-pip
sudo apt-get install libnuma-dev
```

Then the DPDK package has to be installed:

```
sudo wget https://fast.dpdk.org/rel/dpdk-18.02.1.tar.xz \textit{to
    ↪ get the DPDK package}
sudo tar -xvf dpdk-18.02.1.tar.xz
```

After performed these steps, we install the VPP software. The following steps are performed always in the VM's CLI:

```
sudo mkdir /vpp
sudo chmod 777 /vpp
 cd /vpp
 git clone https://gerrit.fd.io/r/vpp ./
 sudo make install-dep
 sudo make build
 cd build-root
 make V=0 PLATFORM=vpp TAG=vpp install-deb
 export UBUNTU="xenial"
 export RELEASE=".stable.1804"
 sudo rm /etc/apt/sources.list.d/99fd.io.list
 echo "deb [trusted=yes] https://nexus.fd.io/content/repositories/fd
    ↪ .io$RELEASE.ubuntu.$UBUNTU.main/ ./" | sudo tee -a /etc/apt/
    ↪ sources.list.d/99fd.io.list
 sudo apt-get update
 sudo apt-get install vpp vpp-lib
 sudo apt-get install vpp-plugins vpp-dbg vpp-dev vpp-api-java vpp-
    ↪ api-python vpp-api-
lua
```

However we still miss the NICs binding. The NICs have to be bounded directly to VPP, where instead now they are still attached to the Linux Kernel. These steps will allow VPP to work properly.

```
 cd /vpp/build-root/build-vpp\_debug-native/dpdk/dpdk-stable-18.02.1
 sudo modprobe uio
 sudo make install T=x86\_64-native-linuxapp-gcc
 sudo insmod /vpp/build-root/build-vpp\_debug-native/dpdk/dpdk-
    ↪ stable-18.02.1/x86\_64-native-linuxapp-gcc/kmod/igb\_uio.ko
```

In the following steps the two NICs are attached:

```
sudo /vpp/build-root/build-vpp\_debug-native/dpdk/dpdk-stable
    ↪ -18.02.1/usertools/dpdk-devbind.py --bind igb\_uio
    ↪ 0000:00:06.0
```

```
sudo /vpp/build-root/build-vpp\_debug-native/dpdk/dpdk-stable
    ↪ -18.02.1/usertools/dpdk-devbind.py --bind igb\_uio
    ↪ 0000:00:07.0
```

The third NIC remains in the linux kernel to allow the communication via SSH. Finally, the last step is to restart VPP:

```
sudo service vpp stop
sudo service vpp start
```

After performed these steps, VPP should be operative and working. To check if it is working, in the terminal of the Virtual machine we write:

```
sudo vppctl
```

To enter in the VPP CLI and then, to see if the NICs have been correctly binded:

```
show interfaces
```

However, we need to modify some features of the NICs: in the section regarding the network interfaces inside the Amazon Console Management, all the NICs have the *Source/Dest. check* field disabled because each EC2 instance performs source/destination check by default and that is because the instances should be either the source or the destination of the traffic it receives or send. After installed VPP inside the VPC, it has to become the main router. In order to achieve this, we have to:

- create manually route tables with the console management
- use Segment Routing version 6 [9] to connect VPP with the other VPP machine inside the second VPP

For the first part, we go in the console management section regarding the route tables and then we modify them in order to tell the packets that if they have to go to the other VPC they should pass through VPP. VPP, when it will receive the packets directed to the other VPC, will encapsulate them in IPv6 packets and then forward them outside of the VPC. Table 2.2 contains an example of route table:

In the target column, local means that all the traffic that has the IP address inside the destination range is routed within the VPC. In the other cases, the IGW corresponds the Amazon Internet Gateway and, as the destination tells, all the traffic is routed towards the Internet. The last field belongs only to the route table of the subnet where there are the VPP's and the client/server's NICs and tells that the specific traffic with IP address 10.0.0.0/16 (addresses belonging especially the other VPC) has to pass towards the VPP's NIC.

Table 2.2: Example of a route table, inside the Amazon Console management. *igw* stands for Internet Gateway and eni for network interface

Destination	Target
10.10.0.0/16	local
2a05:d012:79e:b500::/56	local
0.0.0.0/0	igw-52a8293b
::/0	igw-52a8293b
10.0.0.0/16	eni-128d2b39

In the second case, we define the connections via SRv6. In order to achieve the connection, we repeat the passages describe earlier and create a new VPC with two new Virtual Machines. The new VPC could be or in the same region or in a different one. Afterwards, we connect them with SRv6. We enter in a VPP CLI , and we insert the Segment routing Commands:

```
set int state VirtualFunctionEthernet0/6/0 up
set int state VirtualFunctionEthernet0/7/0 up
```

With these two commands we set up the interfaces inside the VPP environment. If we do not type these commands, VPP will not see these interfaces

```
set int ip address VirtualFunctionEthernet0/6/0 10.1.4.117/24
set int ip address VirtualFunctionEthernet0/7/0 2600:1f14:e0e:7f00:
    ↪ f672:1039:4e41:e68/64
```

Now we are setting the IP addresses of the NICs, writing them manually. Notice that the IP addresses are assigned automatically by Amazon's DHCP server inside the subnet.

```
set sr encaps source addr 2600:1f14:e0e:7f00:f672:1039:4e41:e68
sr localsid address 2600:1f14:e0e:7f00:8da1:c8fa:5301:1d1f behavior
    ↪ end.dx4 VirtualFunctionEthernet0/6/0 10.1.4.117
sr policy add bsid c:1::999:1 next 2600:1f14:135:cc00:43c1:e860:7ce9
    ↪ :e94a encap
sr steer l3 10.2.5.0/24 via bsid c:1::999:1
```

Here we are defining the SRv6 commands. In order we create a localsid, which basically means that the IPv6 address is associated with a SRv6 function. In this case the command is telling that the IPv6 packets coming with that IPv6 destination address should be decapsulated into a IPv4 packet and then they have

to be forwarded through the interface *VirtualFunctionEthernet0/6/0*, which will send the packets towards the destination *10.1.4.117*. Then a policy is created, saying that all the packet belonging to a certain bsid should be encapsulated and transmitted towards the next node with that IPv6 address, which belong the other VPP machine inside the other VPC. Finally the last command say that all the packets which destination address belongs to that subnet has to follow the policy previously created.

```
set ip6 neighbor VirtualFunctionEthernet0/7/0 fe80::84f:3fff:fe2a:
    ↪ aaf0 0a:4f:3f:2a:aa:f0
ip route add ::/0 via fe80::84f:3fff:fe2a:aaf0
    ↪ VirtualFunctionEthernet0/7/0
```

Finally, these commands tells the IPv6 packets where they have to go to reach the Amazon Internet Gateway and to leave the VPC.

Instead, in the Virtual Machine used as client/server we need to type these commands:

```
sudo /sbin/ip -4 addr add 10.1.2.113/24 dev ens6
sudo ifconfig ens6 up
sudo /sbin/ip -4 route add 10.2.0.0/16 via 10.1.4.117
```

Basically, we are setting up a NIC and creating an ip route.

In the first Appendix, it is possible to find the code for the other VPP and VM belonging to the second VPC. After performed this step, the VPP machine is ready. For the future use, we create a snapshot of this machine, which would be later useful with our Terraform script.

Afterwards, the automating work begin. Automating the deployment means to avoid to repeat every time the configuration we did previously by hand and, to reach our goal, three steps are highlighted:

- *Plumbing*
- Use of *DHCPv4* and *DHCPv6*
- *Terraform* Script

The last option will be described more in details in the following section. Plumbing [53] is a term that describes the connections between resources in a cloud environment and therefore we are referring to the the interconnection components that form the cloud environment. It is an analogy of the term "Plumbing" used in the water systems. In our work Plumbing is important because we wanted to avoid to

touch the Virtual Machines. We want that when a Virtual Machine is spawned, it has to automatically know the presence of VPP machine and that it has to forward packet through it. We analyzed different solutions, also looking at the competitors like *Pfsense network gate* [18] or *Cisco CSR router 1000v* [14]:for example we investigate the use of mDNS or others protocol. In the end the simplest solution is to modify the route table in Amazon console management adding a route telling that all the packets with destination address belonging to other VPCs have to pass firstly through VPP.

The second part of the automating process is to use DHCP to automatize the process of gathering the IP addresses, instead of write them manually in the configuration as we did in the previous configuration. The Dynamic Host Configuration Protocl (DHCP) is an internet protocol that is based on a client-server model, where a DHCP serve allocate network addresses and deliver configuration parameters to dynamically configured hosts [19]. When creating a VPC, inside every subnet, Amazon automatically configure a DHCP server [4]. This means that the user can assign a private address (the public addresses are always assigned by default by Amazon) only when creating the Network Interfaces, otherwise Amazon will automatically assign one.

Therefore, since the addresses are already assigned we need only to retrieve them automatically. In order to perform such operation, we use several VPP commands, to avoid writing every time the specific address. In the case of IPv4 addresses, we used some commands referring to DHCPv4, which are already present in the library of VPP 18.04:

```
sudo vppctl set dhcp client intfc VirtualFunctionEthernet0/6/0
```

where *sudo vppctl* enter in the command line of VPP software, *set dhcp client* asks to retrieve the IP address *VirtualFunctionEthernet0/6/0* is the name of the interface

Instead, retrieving the IPv6 addresses is a little more difficult, since VPP 18.04 does not have any implementation of DHCPv6. Therefore, we install the new version of VPP, VPP 18.07, where the implementation of DHCPv6 IA_NA is present. The latter is a specification of the DHCPv6 protocol, and means that An IA (identity association) that carries assigned addresses that are not temporary addresses [26]. In this case, the role in this work was firstly to beta testing its implementation in the Amazon environment and then use it to the automation purposes. In the end, the final command typed was:

```
sudo vppctl dhcp6 client VirtualFunctionEthernet0/7/0
```

where *sudo vppctl* enter in the command line of VPP software, *dhcp6 client* aks to retrieve the IPv6 address and *VirtualFunctionEthernet0/7/0* is the name of the NIC.

However, the biggest step performed in the automating deployment is through Terraform, and its scripting language, which is described in the following section.

2.4 Implementation

In this section, we show why and how Terraform is implemented to automatize the deployment of VPP inside a Virtual Private Cloud. Firstly, we describe why Terraform is chosen among the others competitors. Then, we focus on Terraform's features and how the code is developed, inserting also some example from our code . In the second Appendix, we show the complete and working code, which was used to deploy VPP. Finally, we quickly overview some of the issues encountered, focusing on Terraform limitations and future works.

2.4.1 Tools

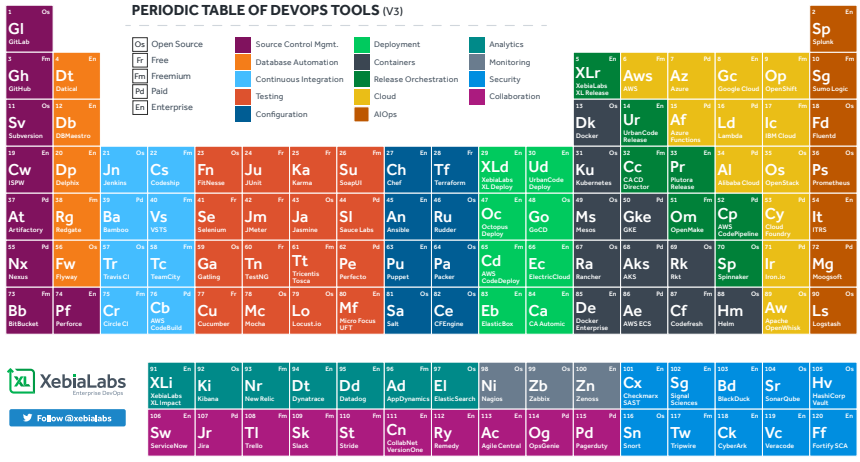

Figure 2.4: DevOps Periodic Table: here there are some of the tools published under the DevOps paradigm. Terraform is present inside the configuration slot. Image taken from [63]

In 2009 Patric Debois coined the term "DevOps" [59], which is the short way to write "Development Operations". The latter is a software engineering culture which intends to unify software development (Dev) and software operation (Ops). The main feature of the DevOps movement is to strongly allow automation and

monitoring at all steps of software construction, from integration, testing, releasing to deployment and infrastructure management. One of the field where DevOps tools are focusing is inside the cloud orchestration tools, where it is possible to deploy infrastructure using coding languages: this process is called *Infrastructure as Code*. Infrastructure as code (IaC) is the process of deploying resources through machine-readable text files. [30]. IaC approaches are hence promoted for cloud computing, and they usually go together with the infrastructure as a service (IaaS).

IaC grew as a response to the difficulty exploited by technology-utility computing and second-generation web frameworks. In 2006 Amazon launched Amazon Web Services‘ Elastic Compute Cloud and in the same time the first version of Ruby on Rails came out and with new tools emerging to handle this ever growing field, the idea of IaC was born. It seemed appealing to model infrastructure with code, and then having the ability to design, implement, and deploy applications infrastructure with known software best practices.

We overview three tool, with which the automating deployment could be performed:

- *AWS CloudFormation*
- *Cisco CloudCenter*
- *Terraform*

AWS CloudFormation [5] provides a programming language to provision all the infrastructure resources in Amazon cloud environment. It allows to use a text file for the modelling and provisioning, in an automated and secure manner, of resources needed. CloudFormation is divided into two parts: JSON text file, which is a template, and a stack, which are the resources instantiated and running. The first one is declarative and non scripting file, which defines what AWS resources need to be instantiated for the application the user want to create. JSON, which stands for *JavaScript Object Notation*, is a text-based data format fo representing simple data structures and objects in Web browser-based code. When the JSON is developed and run, it automatically creates the resources described in the code, putting dependencies and data flows in the right order. The drawback of using AWS Cloudformation is that it can be only used in Amazon's cloud environment.

The second option is called Cisco CloudCenter [52]. It uses a single platform to deploy and manage workloads and cloud resources without going through cloud-specific management tools. Cisco CloudCenter could support public clouds such as *AWS, Azure* or either Private clouds, such as *Kubernetes, OpenStack*.

However at the end we choose Terraform by Hashicorp [28] since one of the proposed future work is to re use the same code with a different cloud provider,

hence it needs to be *Cloud Agnostic* as much as possible, what Cloudformation does not give and in mean time it should give the maximum flexibility to the developer without any constraint and, on the other hand, Cisco CloudCenter is not to be very flexible.

2.5 Terraform

Terraform is a *"tool for building, changing, and versioning infrastructure safely and efficiently"* [28] which can deal with public cloud provider as well as custom in-house solutions. It is written with Go language, a programming code created by Google. Terraform is based on configuration files with a .tf extension and it supports two formats: JSON or a proprietary language called HCL Terraform syntax.

Developers create several configuration files with an .tf extension, which describe to Terraform the components needed to instantiate inside the cloud environment. After writing the files, we launch Terraform, which generates an execution plan: in this phase Terraform summarize the steps it will perform to achieve, and then, if it does not find any errors, it executes, building therefore the described infrastructure. Moreover, Terraform is able to detect if the previous configuration instantiated has changed and if the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied. Finally, Terraform can manage resources such as compute instances, storage, networking, DNS entries, SaaS features and so on

Some of Terraform key features are:

- *Infrastructure as Code*: The infrastructure that the user needs can be described using a high-level configuration syntax, called HCL in the Terraform case.

- *Execution Plans*: After run the configuration files, Terraform generates a so called execution plan. The latter shows on the terminal output what steps Terraform will perform.

- *Change Automation*: Complex change sets can be applied to the user's infrastructure with minimal human interaction, thanks also to the *execution plan*.

More over, Terraform is a very versatile tool. In the following, we list, also present in [28] some of the field where it has an application. Due to its extensible nature, providers and provisioners (blocks inside Terraform script, which enable the possibility to perform bash commands or scripts) can be added to further extend Terraform's ability to manipulate resources:

- *Heroku App setup*: It is a PaaS for web app, where developers create an app and then attach adds on.
- *Software Demos*: Software writers can use Terraform to create a demo on Public Cloud Providers.
- *SDN*: Terraform can codify the configuration for software defined networks. This configuration can then be used by Terraform to automatically setup and modify settings by interfacing with the control layer. An example are Amazon's VPC [17].
- *Multi-Cloud Deployment*: Normally it is difficult to realize multi cloud deployment but since Terraform is CLoud Agnostic hence it allows to use only a configuration file, for multiple Cloud Providers.

2.5.1 Terraform Script

In the next paragraphs we describe and show some examples of Terraform code.

The files inside Terraform environments are called *configuration* and they are simply text files, with *.tf* format.

The configuration files are written in two formats: Terraform format and JSON. The Terraform format, which is also called HCL is more human-readable,it supports comments, while instead the JSON format is meant for machines. Terraform format ends in .tf while JSON format ends in .tf.json.

As seen before, the syntax of Terraform is called HCL (*HashiCorp Configuration Language*). It is a trade off between readability and machine-friendly. Below there is an example of HCL, taken directly from Terraform Website [28]:

```
# An AMI
variable "ami" {
  description = "the AMI to use"
}
/* A multi
   line comment. */
resource "aws_instance" "web" {
  ami = "${var.ami}"

  source_dest_check = false

  connection {
    user = "root"
```

```
  }
}
```

The syntax is quite simple but in order to understand better, we report some comments:

- comments are made with # or /* and */ (for multi-line ones)
- Values are assigned with the syntax of key = value.
- Strings, which are inside a double quote, can interpolate other values using syntax wrapped in , *suchas*var.foo.
- Boolean values: true, false.

As highlighted before, Terraform also supports reading JSON formatted configuration files. Always taking the example from the Terraform website [28], here is an example of JSON file:

```
{
  "variable": {
    "ami": {
      "description": "the AMI to use"
    }
  },

  "resource": {
    "aws_instance": {
      "web": {
        "ami": "${var.ami}",
        "count": 2,
        "source_dest_check": false,

        "connection": {
          "user": "root"
        }
      }
    }
  }
}
```

On the other hand JSON files lack of human readability and comments.

Let now have a look of a very simple Terraform script, which is able to create an instance inside the AWS cloud environment. Notice that instead the example below is taken from the code we use to deploy VPP.

```
provider "aws" {
  access_key = "${var.access_key}"
  secret_key = "${var.secret_key}"
  region = "eu-west-1"

}

resource "aws_instance" "vpp-London-2" {
  ami = "ami-ea9b728d"
  instance_type = "m5.2xlarge"
  key_name = "vpp-2_London-2"
  vpc_security_group_ids= ["${aws_security_group.allow_all.id}"]
#security_groups = ["${aws_security_group.allow_all.id}"]
        subnet_id="${aws_subnet.management-2.id}"
        availability_zone = "eu-west-2a"

tags {
    Name = "vpp-2-London-2"
  }

}
```

The *provider block* is used to configure the Cloud Provider, which in our case is *aws*. In our case inside the block there are two variables. They are the Access and Secret Key to allow Terraform to enter inside Amazon's Cloud and they are easily recognizable as variables for the name *var.access_key*, where instead the presence of the $ gives the actual value of the variables. The dot creates a dependency, saying that there it exists somewhere, in the same configuration file or in other .var file, some variables with those names which have the values we are looking at. Instead the field Region tells in which region Terraform has to operate.

Instead The *resource* block defines a resource, which could be an EC2 instance for example.

The resource block has two strings before opening the block: the resource type and the resource name. In our example, the resource type is "aws_instance" and the name is "VPP-London-2." The prefix of the type maps to the provider. In our case "aws_instance" automatically tells Terraform that it is managed by the *aws* provider. Moreover at its inside there can be seen many interesting features. For

instance, it is possible to choose the the type of AMI (Amazon Machine Image) to install on the instance. In our case the AMI is directly taken from the snapshot of VPP we created before. Hence means that when this instance will be created by Terraform, inside it will already have VPP installed. This is useful since one of the future goal is to create a virtual machine wit already VPP installed and then released into the AWS marketplace. Then it is possible to choose the instance type and the key name we want to associate to enter it (in this case the key has also to be created in the Amazon console Management and separately downloaded). After it in which VPC subnet and availability zone should the machine put. Finally a tag is created.

Then, after the configuration file is created, the resources could be instantiated. In order to perform that, the first command is *terraform init*, which initializes various local settings and data that will be used by subsequent commands and install the Provider binary for the providers in use within the configuration, which in this case is just the AWS provide.

Then , we perform the command *terraform apply* and an output similar to below appear (we show only a part of the all output):

```
$ terraform apply
+ aws_vpc.London-2
     id: <computed>
     assign_generated_ipv6_cidr_block: "true"
     cidr_block: "10.1.0.0/16"
     default_network_acl_id: <computed>
     default_route_table_id: <computed>
     default_security_group_id: <computed>
     dhcp_options_id: <computed>
     enable_classiclink: <computed>
     enable_classiclink_dns_support: <computed>
     enable_dns_hostnames: "true"
     enable_dns_support: "true"
     instance_tenancy: "dedicated"
     ipv6_association_id: <computed>
     ipv6_cidr_block: <computed>
     main_route_table_id: <computed>
     tags.%: "1"
     tags.Name: "London-2"
```

This output shows the execution plan, describing which actions Terraform will take in order to change real infrastructure to match the configuration. If Terraform apply failed with an error, an error message will appear.

If errors don't appear, Terraform complete its task and now the resources are available through the EC2 console management. It is possible to show the current state of the resources using *terraform show*.

After defined how Terraform works in a general way, let see how the actual code is created to automate the deploy of VPP. In the following paragraphs we show only a few example from the code. All the working code is available in the second Appendix section. Several blocks are defined to make everything work in an automated way. We create a vpc block to define the VPC:

```
resource "aws_vpc" "India" {
  cidr_block = "10.7.0.0/16"
  instance_tenancy = "dedicated"
        assign_generated_ipv6_cidr_block = true
        enable_dns_hostnames =true
 tags {
    Name = "India"
  }
}
```

Inside the block, we define a CIDR field, to assign the IPv4 addresses range, we tell to use a dedicated tenancy, which means that every resources will use dedicated hardware. Moreover we generate a IPv6 block and finally a tag name.

Afterwards we create three subnets, and in the example we show only the IPv6 subnet (the one that links VPP towards the internet):

```
resource "aws_subnet" "ipv6" {
  vpc_id = "${aws_vpc.India.id}"
  cidr_block = "10.7.3.0/24"
  ipv6_cidr_block = "${cidrsubnet(aws_vpc.India.ipv6_cidr_block, 8,
      ↪ 1)}"
  availability_zone = "eu-central-1a"
    assign_ipv6_address_on_creation = true
    tags {
    Name = "ipv6"
  }
}
```

In this block, we can see an example of dependency in Terraform environment. In the VPC field, there is a dependency with the ID of the VPC created before with Terraform. An IPv4 and IPv6 addresses range are assigned, notice that the

latter is not flexible as IPv4. Finally, we also choose the availability zone where to deploy the subnet.

Afterwards a internet gateway has to be attached and the route table needs to defined and attached to the subnets:

```
resource "aws_route_table" "r" {
  vpc_id = "${aws_vpc.India.id}"

  route {
    cidr_block = "${var.SUBNET}"
    network_interface_id = "${aws_network_interface.vppIPV4.id}"
  }

route {
       cidr_block = "0.0.0.0/0"
       #ipv6_cidr_block = "::/0"
       gateway_id = "${aws_internet_gateway.vpc_igw.id}"
    }

  tags {
    Name = "India"
  }
}

resource "aws_route_table_association" "association" {
  subnet_id = "${aws_subnet.vpp.id}"
  route_table_id = "${aws_route_table.r.id}"
}
```

This last example is very interesting because this block refers to the route table belonging to the VPP part. In the code we see that there are two sub-blocks called route, which define the routes path. In one block it is defined the gateway address and it is associated with the gateway ID but in the other one it is used a variable to define the receiving subnet and then it is associated to the IPv4 network interface belonging to VPP. the variable is used so an user, when use Terraform Apply command could insert the subnet address he wants, allowing the code to be broadly used. Afterwards, the route table is associated with the corresponding subnet.

Furthermore, we create a security group:

```
resource "aws_security_group" "allow_all" {
  name = "allow_all"
  description = "Allow all inbound traffic"
  vpc_id = "${aws_vpc.India.id}"

  ingress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }

  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }

 tags {
    Name = "allow_all"
  }

}
```

The ingress and egress block told which ports and which traffic should enter and leave the instances. In this case, since we don't have any particular constraint, we just allow any traffic in input and output. Later, the instance resource block we create, which is the same shown before but also with a new field that allows the creation of *user data*. The latter could be a series of commands, of a text file that is run during the bootstrapping of the instance. In our case it is used to attach NICs to VPP and retrieve automatically their IP addresses

Finally we vuild some blocks regarding the assigning of an Elastic IP address and of the Network Interfaces:

```
resource "aws_network_interface" "vppIPV6" {
subnet_id = "${aws_subnet.ipv6.id}"

  security_groups = ["${aws_security_group.allow_all.id}"]
   source_dest_check = false
```

```
  attachment {
    instance = "${aws_instance.VPP-India.id}"
    device_index = 2
  }
}
```

Notice a field which disable the Source/Dest check is present and also a attachment block, which tells with which instance it should be attached.

2.6 Issues

However, even though Terraform is useful for the purposes of the work, it does not achieve a full automation. For instance, every different user has to insert manually its Access and Secret Key, to let the script to deploy the instances, hence Terraform does not lead to a 100% of automation. Moreover, it is very difficult to automate some VPP commands, especially the ones which refer to the network interfaces since the only way to insert the commands in an automated context is during the script deployment process, when the instance is firstly instantiated, using the so called "User Data" [17]. The latter allows to perform scripts or bash commands inside the instance, even though it works only during the booting time and, at that point, using the script, the network interfaces are not still attached to the instance.

The best solution would be therefore to create a bash script with all VPP commands, send it to the machine through Terraform via ssh and then access the machine and use it through the terminal. A way to develop more automation could be through the usage of *Boto3* library and see if it possible to use them in conjunction with Terraform. Boto3 [8] is the Amazon Web Services (AWS) SDK for Pyhton, and it should automate more processes, such as the getting of Access and Secret Key.

Chapter 3

Measurements Methodology

This chapter describes the methodology and the assumptions made, after VPP deployment, to perform some experimental measurements. The first section highlights which performance metrics we decide to measure and afterwards which tools we use to evaluate them. The second section presents an overview of some of the Related Works regarding experimental measurements in AWS cloud and we find very interesting to highlight how much the performances are changed over the years. Then, Section 3 describes the experimental scenarios in which the measurements took places and finally in Section 4 we focus more on some technical aspects of instances.

3.1 Metadata

After implementing VPP inside the Virtual Private Cloud, the work shifted to see if and how Amazon's Cloud Network would benefit of the VPP presence. To evaluate the performances, we have defined several metrics, which could help us to have a better insight. Therefore we chose four different metric:

- *Round Trip Time (RTT)*
- *Throughput*
- *Time to Live (TTL)*
- *Packet Losses*

The first metric represents the delay of a packet transmitted into the network and it is evaluated as the time the packet takes to reach the destination and come back (plus the time to be processed by each node of the path but normally this one

is significally low). It is usually measured, as also in the following experiments, in milliseconds (ms). In formula:

$$\text{RTT} = 2\frac{\text{Packet Dimension}}{\text{Bandwidth}} \tag{3.1}$$

We evaluate the RTT two times: in a configuration with VPP and in another without VPP, to see the differences due to the presence of the software router.

The throughput is the rate of successful bits delivered over a channel. The unit of measure is bits per second but generally bigger units such as Megabits/sec or Gigabits/sec are used. In the following experiments, we decide to evaluate the throughput in two different way: one performing a measurement every 20 second of data sent (hence avoiding that the TCP slow start and congestion avoidance algorithm would affects too much the measurements) for 30 times. We choose to perform the same experiments 30 times to gather relevant statistical experimental data. With the second choice we send work load of different sizes: *1Mbyte, 10 Mbytes, 100Mbytes, 1Gbyte, 5Gbytes and 10 Gbytes (with the latter only for intra-cloud)*. We decide to have different work size to see how the Amazon's cloud infrastructure would react. This help to notice, for example, that a shaper is actually used to shape the traffic ejected from an instance. As before, the throughput is evaluated in the case with and without VPP.

The Time To Live counter (from now on TTL) is a field in the IPv4 packet header, which counts the number of hops that a packet does in the Internet Network. It has a value that ranges from 1 to 256 (typically in Linux system and in our experiments it is setted as 64). If the value reach 0, the network will discard it. It prevents, since IP is a datagram and best effort protocol, that a packet wanders in the network forever causing congestion and problems, if it does not reach the destination.

Finally, the Packet Losses ratio instead indicates the number of packets which have not reach the destination with respect to packets sent. It is an important parameter to evaluate because, in a TCP environment, a loss correspond to a retransmission of the packet and a decreasing in the window transmission size, which leads to throughput reduction.

Afterwards, we decided in which scenario performs our measurements. However, even if we will be deeply describe it in the following sections, for now we can say that ww want to focus on two aspects: the presence of two VPP machines inside the same region and, if not, how much the physical distance (in kilometers) between the two machines is. In formulas:

$$\text{O,D pair} = O, D \in \{Region_{\text{i}}\}, \quad \text{where } \; i = 1{,}2 \tag{3.2}$$

$$\text{VPC}_{\text{distance}} = \{\text{Same } \textit{Datacenter},\ \text{Same } \textit{Availability Zone},\ \textit{Same Region}\} \tag{3.3}$$

3.1.1 Tools

To evaluate the metrics, we propose three different tools but, after several experiments, we decide to use only two of them. The tools are:

- *Ping*
- *Traceroute*
- *Iperf3*

Ping [43] is a computer network software used, among many things, to probe the presence of a host inside a network and therefore it is used also to evaluated the RTT of a packet sent. It leverages on the Internet Control Message Protocol (ICMP), which is a protocol used to send error messages and operational information. Hence Ping, to perform measurements of the RTT, sends echo request packets to and wait for a reply. The output of the program shows useful informations such as packet loss, TTL and the minimum/maximum/mean RTT with the standard deviation of the mean. Moreover Ping has many command-line features: for instance it is possible to define the wait interval seconds between sending each packet, to set the count on how much packet could be sent and finally if the packets should go to a specific interface. More it can be found in the *man* page of the command [43].

Traceroute [61] is instead an tool for displaying the path and measuring the delays of the packets inside the internet. It is different from Ping, since the latter computes only the final RTT in ms whereas Traceroute detects the path followed to reach a target host, sending packets with increasing TTL. In our experiments, Traceroute role should have been the evaluation of the TTL. However during our first experiments, we notice that the tool was not helpful. In fact, in the measurements without VPP, the tool was able to evaluate the path but with VPP presence not. Our guess is that Segment Routing version 6 protocol has some compatibility problem with the tool. Hence we decide to use Ping tool also to evaluate the TTL.

Finally, we use Iperf3 to evaluate the throughput. The latter [55] is a tool used to measure network performances. It supports both TCP or UDP and IPv4/IPv6 protocols, and it creates a data-stream to measure the throughput. The output generally reports the bandwidth, with the amount of data transferred, the packet loss and other useful parameters. In our experiments we used the iperf version 3.

3.2 Related Work

The interest in monitoring the cloud performances has risen only in the last years, as also highlighted in [1]. This gaining interest is motivated for several reasons: first of all, normally Public cloud provider does not disclose which are actually the real performances of their network but they just give a qualitative informations. For instance, in the Amazon Case [4], the network performances are described with a generic *Low*, *Medium*, *Large* comment or when more performant machine are instantiated, Amazon tells they can reach "*Up to 25 Gigabit*" performances [4]. Therefore it would be interesting to investigate if and how the network performances meet the information advertised by the cloud provider and see how much the Amazon infrastructure changed all over the years. Moreover, it is worth noticing that the literature do not give a clear image of the Public cloud performances. The works report in most cases conflicting results (since they exploit several different methodologies and tools). Furthermore, most of the Measurements were taken with the instances belonging in the same Region. Our work proposes hence some new novelties in this scenario.

First, most of the previous works refer only to intra cloud measurements, like in [41], [38] and [56]. Instead in our work we focus on both cases: intra-cloud but also with the VPC belonging in different regions. Finally, our work is the first in the literature, almost that we know, to insert a high speed Software Router inside a cloud environment and then to measure its performances.

However, we highlight that the measurements in the cloud suffer of unreliability, caused by:

- The large number of possible scenarios (number of Regions, Instance Types). In fact, for example, every measurements taken by the previous papers are evaluated with different intrinsic characteristics, such as the Instance type and size, leading to intrinsic differences in the network performances and results.
- The network resource allocation which could differ from time to time, impacting hence the reliability of the Network performance Measurements.

In Table 3.1, we show an overall picture of the Cloud Network performances of the previous works.

Wang et al [56] paper focus on the virtualization impact in the networking performances inside a cloud environment. They use tools such as *ping* and *ad hoc* ones such as CPUTest [56] to define the intra-cloud network performances, evaluating throughput TCP/UDP, packet losses and delay. At the end they find a significant delay variation and throughput instability and according to their results, this is not decided by a shaper but instead by the virtualization process performed

Table 3.1: Previous works measurements inside AWS environment

Paper	Year	Region	Throughput (Mbps)
Wang et al	2010	North California, Ireland	400-800 Mbps (small) 700 - 900 Mbps (medium)
Shad et al	2010	North California, Ireland	1.6 - 6.4 (US) 3.2 - 7.2 (EU)
Li et al	2010	North California, North Virginia Ireland	600-900 Mbps
Raiciu et al	2012	NA	1000 - 4000 Mbps
LaCurts et al	2013	NA	296 - 4405 Mbps
Persico et al	2015	North Virginia, Ireland, Singapore, Sao Paulo	500 Mbps medium) 750 Mbps (Large) 950 Mbps(xlarge)

by Amazon. Depending on the size of the Virtual machine, they have achieved a different throughput.

Shad et al [48] focus instead on the performance unpredictability of AWS since for them it this would be a issues for database researchers and for database applications. They use *iperf* to evaluate TCP and UDP performances and they propose several benchmarks to evaluate VM deployment time together with memory, CPU, disk I.O performance, storage service, access and network bandwidth.

Also Li et al [38] focus on different clouds to compare them in term of costs, VM deployment time and storage. For the network performances, within the same Region, they use *ping* and *iperf* for evaluating Throughput and Ping. However, in [38] they declare that their results can not be considered reliable generally because they have obtained them in few specific scenario.

Afterwards, in Raiciu et al [**rai**] they use *Ping* and *Traceroute*, *iperf*) to perform the performance measurements. An interesting result they find is that the throughput measured depends on the VM position inside the region.

Instead, LaCurts et al [36]'s work focus on the the deployment of the resources. Since the user is not able to decide where to deploy its application and, the VM position will influence its network performances, they exploit *Choreo*, a system that decide the VM position depending on the measurements of network performances between a VM pairs using UDP packet trains. They use *netperf*, instead of *iperf*, and their measurement are very variable, ranging from 900 Mbps to 1.1 Gbps.

Persico et al [41] is probably the most complete and deep work for now in Intra Cloud Measurements. They proposed a non-cooperative experimental approach,

taking in account the geographic position, resource prices, and size. As tool they used *nuttcp*, since in their experiments they found that it impose a more precise bit rate. They obtained different throughput values, depending on the type of the instances, ranging from 500 Mbps for Medium size VM to 1000 Mbps for xLarge.

Finally, all these work have an active approach to the network measurements. In Bermudez et al [31] they instead propose a passive approach, finding that the data-center located in Dublin, in the context of Amazon S3 , could achieve a 2Mbps of throughput.

3.3 Experimental Scenarios

Our scenario consists in two different VPC located or in the same region or in two different regions. Inside them, as highlighted in the previous sections, we installed VPP and to connect the two VPCs together we use the new protocol Segment Routing version 6. As a reminder, we decide to use this new protocol since Amazon EC2 is one of the few public cloud provider, along with Microsoft *Azure*, that allows to allocate IPv6 addresses. Then we create two VMs, one for VPC, and these ones will be our client and server machine.

The measurements will hence take place on the client side. The traffic will go though the first VPP where the SRv6 protocol will steer the packets towards the second VPP. As said before, we decided to perform measurements within the same region and with the VPC belonging in two different regions. The choice of the regions was driven by the distance and for age of construction of the data-center, to see the differences due to the different age of deployment. More over we also choose a small group of data-center deployed during the same year to see if this could lead to some significant analogies. Our main hub is in Paris Region (code eu-west-3), which therefore is always the client sending data.

In Table 2.1 reports the distance of the selected regions: when the same name is written twice it means that the measurements are performed intra-cloud (same region). Furthermore, for the Intra-cloud measurements the distance is reported as *same Availability Zone* or *Different Availability Zone*. We would like to highlight this difference since in an Amazon's region are present several Availability Zones 2.2. An Availability zone is simply a data-center located physically in other part of the region taken in consideration and the purposes of these AZ are to exploit redundancy in order to help recovering from fault tolerance. Hence a user could deploy the same VM in two different AZs and if one data-center stops to work the other one, located on the other AZ, assures the contents of the VM are still reachable.

Now we will focus on some technical configuration details:

Table 3.2: Physical distance of the combination of regions choosen

Combinations of choosen Regions	Distance	Combinations of choosen Regions	Distance
Paris (2017) - Paris	Same AZ	Paris - Oregon (2011)	∼8200km
London (2016) - London	Different AZ	Paris - Sao Paulo (2011)	∼9300km
Tokyo (2011) - Tokyo	Different AZ	Paris - Tokyo (2011)	∼9700km
Paris - London	∼500km	Paris - Sidney (2012)	∼17000km
Paris - Ireland (2007)	∼1000km		

Firstly, we highlight the type of instances selected. We decide to use m5 instances type, since they are:

- General Purposes hardware
- Allow to have high network performances (up to 25 Gbps network bandwidth for the most powerful one)
- they are versatile for many applications

As reported in Table 2.3, several types of m5 instance exist, with *m5.24xlarge* considered as the most performing one. In our experiments we want to maintain the same hardware for every instance (also for the VM containing VPP). The type used is the *m5.2xlarge* one which as hardware characteristics, reported in Amazon EC2 website are: *31 ECUs, 8 vCPUs, 2.5 GHz, Intel Xeon Platinum 8175, 32 GiB memory*, with a maximum network speed of 10 Gigabit [4].

Below we report some numbers of our experiments to give just a quick overview of effort and time spent to perform these experiments. In order, we:

- used more than 30 Virtual Machine (m5.2xlarge type) to cover all the cases
- sent more than 300 Gigabytes of data for our performing Measurements
- spent over $6000 totally to perform all the experiments,
- perfomed More than 720 hours of experiments

Finally, the experiments were performed during the period between June and July 2018.

Chapter 4

Experimental Results

In this section we describe some of the most meaningful results obtained in our measurements. Firstly, in the following section, we show the results regarding the TTL. Then, in the second section we describe the Round Trip Time evaluated and finally we dwell on the throughput results. In the last section we also quickly overview, at high level, the shaper behaviour noticed during our measurements, also highlighted in [41].

4.1 Time To Live

The first metric evaluated is the Time To Live. At a glance, it is a counter which tells the number of nodes that a packet cross. We propose to analyze this metric since it is interesting to see if and how the Segment Routing version 6 could change the path of a packet. Unluckily, since the TTL field belongs to the IPv4 Header, which is itself encapsulated inside the IPv6 header (where Segment Routing version 6 instructions lay) the counter is not decreased until the the packet reaches the destination VPP. This is why in all the measurements we did, in the VPP part the TTL has a constant value of 2 (which means that the packet has crossed two nodes before reaching its destination: basically the two VPP machines). Notice that the value is evaluated by subtracting 64, which is the starting number in the Linux machine, by the number of hops. Instead, in the No VPP case, like in Figure 4.1 and Figure 4.2, the packet is a classic IPv4 packet and therefore, depending on the distance of the destination VPC, it has a different value (the more the destination is distant, the smaller the value becomes).

It is interesting to notice that when the two VPCs are in the same region, the number of hops is equal to one in the case without VPP (like in Figure 4.1), even though they can be in different Available Zone (as a reminder, it means that the

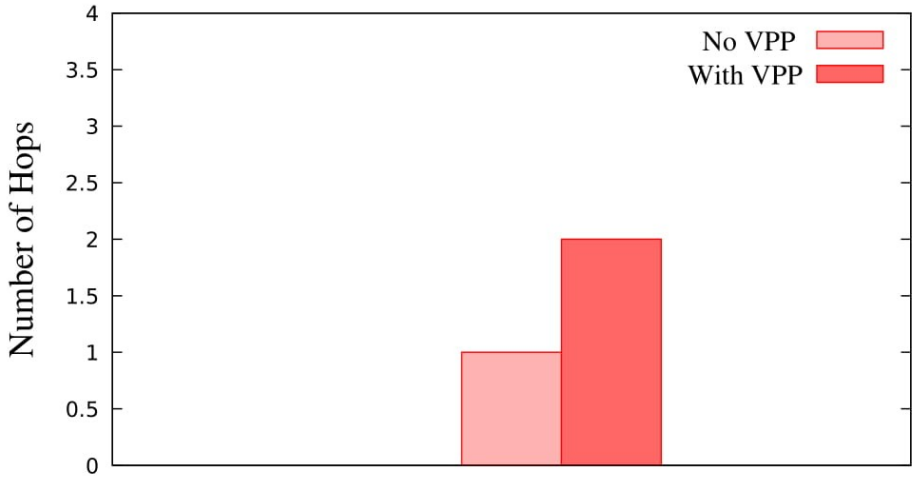

Figure 4.1: Number of Hops within the same region (Paris - Paris).

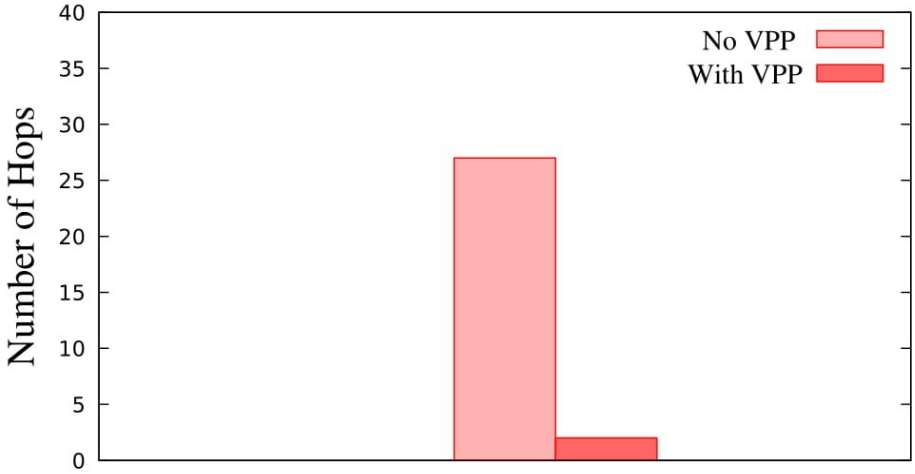

Figure 4.2: Number of Hops with two different regions (Paris - Sao Paulo).

two VPCs are in two separated Data-center located in two distant parts of the Region for instance ≈ 100km). It means that the packet has crossed only 1 Router (which decrements the counter), even though the two VPCs belong to different

AZs.

4.2 Round Trip Time

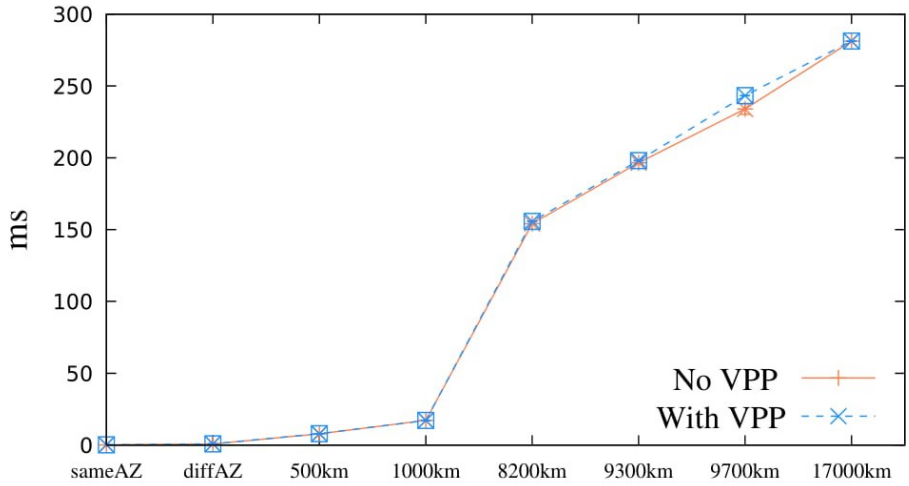

Figure 4.3: RTT of all regions evaluated with a normal scale

Figures 4.3 and 4.4 show the RTTs behaviour and how they change accordingly to the distance of the two VPCs in both VPP and not VPP case. When the VPCs are closer, the RTT is minor with respect of when the two VPCs are very distant from each other and this is obvious since the Round Trip Time depends on the length of the physical link. Further more, the two RTTs increase linearly, even though in the figures this increasing is not very much highlighted since in the horizontal scale there is big a jump between 1000 km and 8000km and this hence does not respect a linear proportion. We use a linear and a logarithmic scale to see better the differences between the two RTTs, since the latter scale works better with small values. We notice that the two RTTs have not the same values but they slightly differ. We can see this difference for instance in Figure 4.3 in the Paris-Tokyo connection and in Figure 4.4 in the first points (hence this means same region and Availability Zone).

To point out better the differences, in Table 4.1 we show which are the values of the two RTTs and in figure 4.5 we show what actually is the difference in terms of μs between the two RTTs:

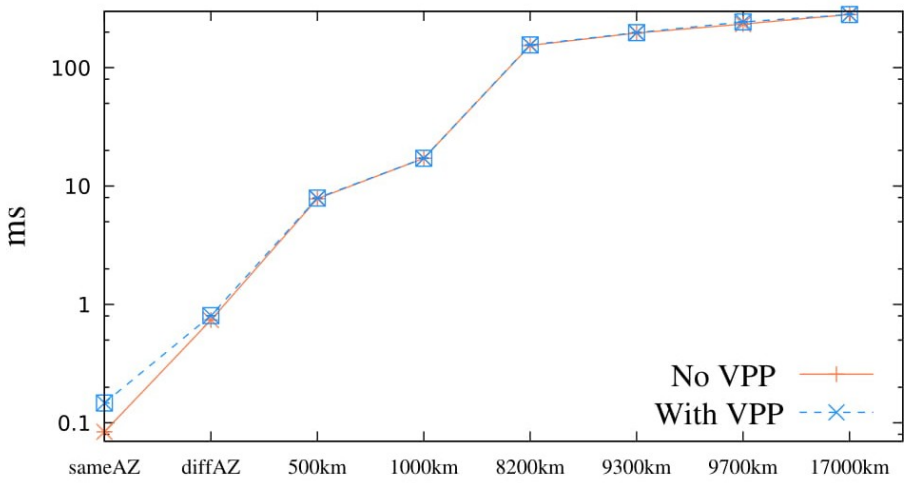

Figure 4.4: RTT of all regionsevaluated with a logaritmic scale

Table 4.1: Values in ms of the two RTTs

	Lon-Lon	Par-Par	Par-Lon	Par-Ire	Par-Ore
Not VPP	0.084	0.74	7.82	17.1706	154.14
VPP	0.147	0.80	7.9	17.1705	155.7
	Par-San	Par-Tok	Par-Sid		
Not VPP	196.54	233.97	281.20		
VPP	198.1	243.32	281.24		

Evaluating this difference we find an interesting pattern, which also will influence the throughput behaviour. With the exception of few cases, the difference between the two RTT is constant, reaching a peak in the Paris Tokyo connection. A justification of this latter behaviour is due to the fact that the packets have followed a very different path during the two measurements. A difference of ≈ 9 ms turns out to be a difference of $\approx$ 1000km, which is huge (basically Italy's size from North to South) and maybe this difference is due to congestion issues and hence the packets were steered into a different path. More over, another reason could be that the Tokyo Data-center is a crowded one. However, we don't have proof or evidence that could support our statement, even though Bermudez et al [31] i highlights that a crowded Data-center could lead to worse performance.

In the other cases, where the difference is constant, we think that with VPP the packet has to make a longer journey. First it has to pass through Amazon

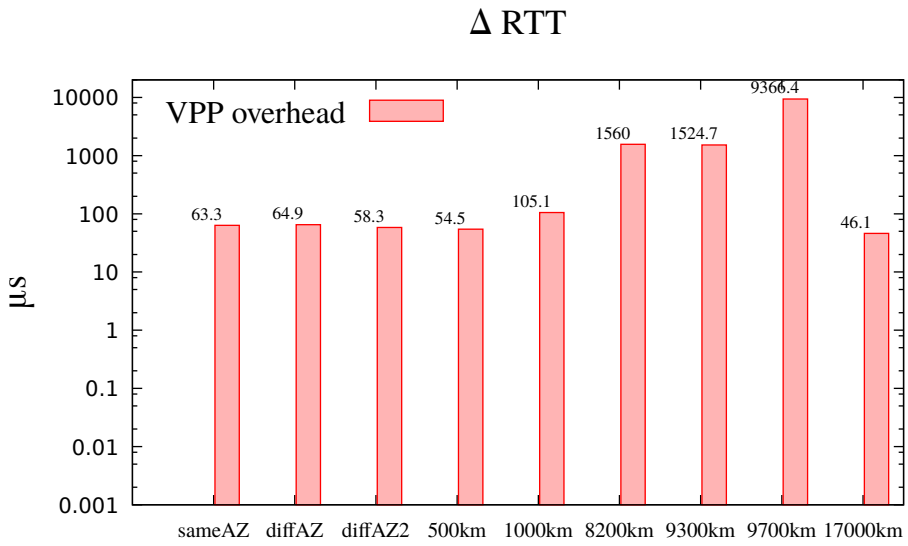

Figure 4.5: Difference in ms of RTT with VPP and RTT without VPP

IGW, which will send the packet towards VPP to get the Segment Routing version 6 header and only then the packets are steered outside the Virtual Private Cloud. Moreover the packet, once arrived in destination VPC, has to pass the VPP destination, before reaching the server. Hence, this add a constant delay and that is why the RTT with VPP is always constantly bigger. This will have an huge impact of the Throughput, as we will see further but, for now, we can say that the throughput is inversely proportional to the Round Trip Time and this difference will play an important role especially in the closer connection, where this difference is more appreciable. In Figure 4.6 we show our assumption previously made.

The Figures shown are however just a summary of what we evaluated during the experiments session. Moreover, to evaluate the RTT we take 10 experiments, with each of them that sent 1000 ICMP packets, to have statistical accuracy and to evaluate the 95% Confidence Interval (CI) of the mean. At a glance, a 95% confidence interval tells the reader that the value evaluated (in this case the mean) has the the 95% of probably to fall in the interval. the CI is evaluated using the formula 4.1:

$$95\% \ \ \text{CI} = \text{E[x]} \pm \text{z}\frac{\sigma(X)}{\sqrt{\text{N}}} \tag{4.1}$$

where:

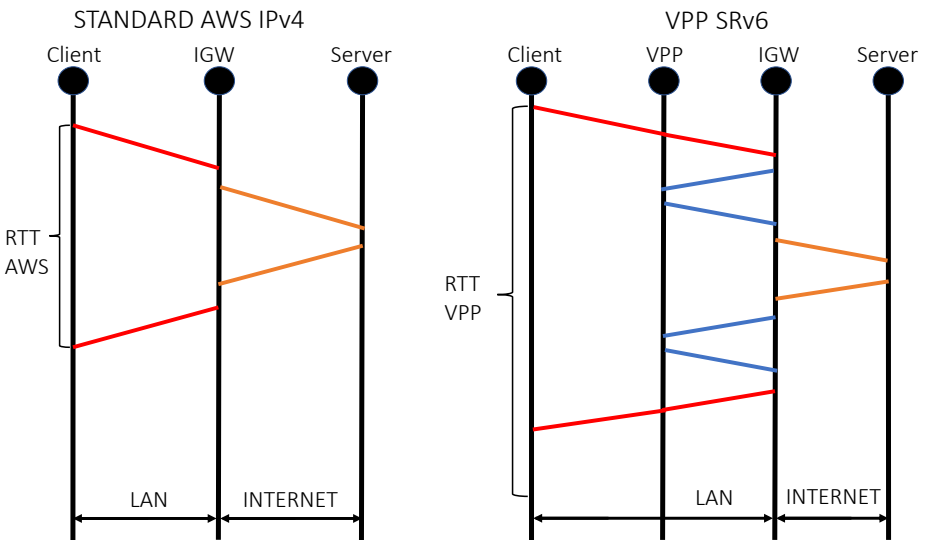

Figure 4.6: Different path performed by the two packets.

- $E[x]$ is the empirical average.
- z is the value of the Student Distribution, if the samples (N) are < 30, otherwise has constant value of z = 1.96.
- $\sigma(X)$ is the standard deviation.
- N are the number of samples.

We also use the confidence interval in the evaluation of the Throughput.

4.3 Throughput

In this section, we show the results obtained by measuring the throughput. We evaluate it in two different way: sending different file size from the client to the server and performing measurements of 20 seconds, both of them repeated for 30 times to obtain relevant statistical results. Before showing our results, we would like to highlight one of the main feature of the throughput, which is the inverse proportionality with Round Trip Time, as shown in formula 4.2:

$$\text{Throughput} = \frac{\text{cwnd}}{\text{RTT}} \tag{4.2}$$

where:

- *cwnd* is the congestion window dimension: it tells of many packet could be sent inside the same Window (the larger it is, the more packets are sent).
- *RTT* is the Round Trip Time.

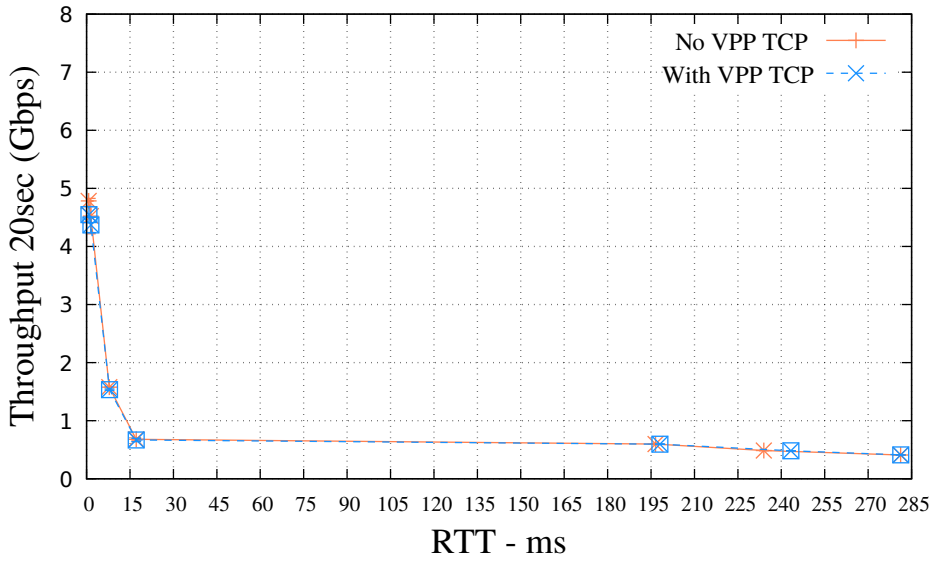

Figure 4.7: Throughput against RTT

In Figure 4.7 we clearly see that for small RTT the throughput (in this case we show only the TCP traffic, since UDP is not constrained by the Round trip Time) is higher, reaching values around 5Gbits/sec. However, the more the RTT becomes bigger, the more the throughput decrease. Actually the throughput curve decrease very fast and for RTT bigger than 15ms, it is stable under 1 Gigabit/sec. Moreover always in Figure 4.7 we see that the throughput values are not the same in both cases but they are different. In Figure 4.8, this difference is better highlighted.

In Figure 4.8, we show the throughput with measurements lasting 20 seconds and repeated 30 times, giving hence statistically accuracy (since we are able to evaluate the 95% confidence interval (CI) of the mean) and in table 4.2 we portrait the actual values obtained from the 20 seconds measurements case.

We see that TCP and UDP flows have two different behaviours. The latter saturates the link, reaching a maximum of 10 Gigabit/sec. UDP's behaviour is justified due to the missing presence of a closed loop-control protocol [45] (which

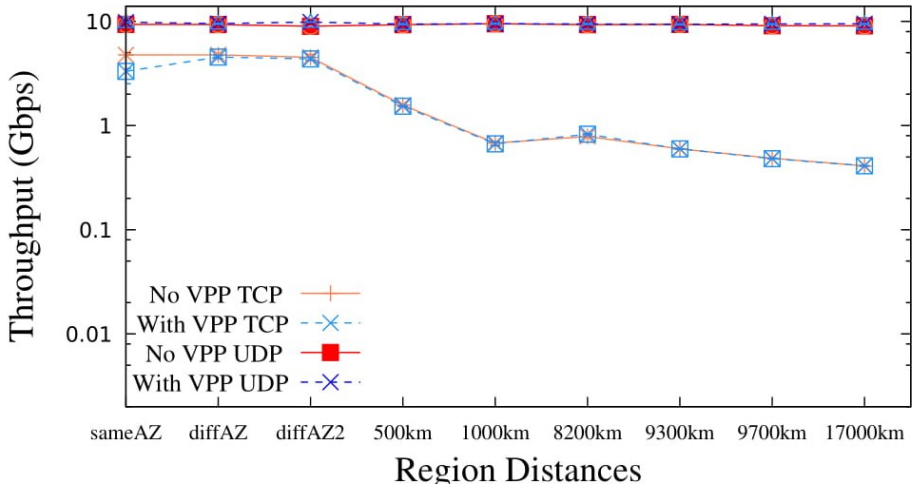

Figure 4.8: Throughput against RTT using a logarithmic scale with 20 seconds measurements

Table 4.2: Throughput values of 20 seconds measurement - all data are expressed in Gigabit/sec

Protocol	VPP/not VPP	sameAZ	diffAZ	diffAZ2	500km	1000km
TCP	No VPP	4.77	4.78	4.52	1.58	0.68
	VPP	3.32	4.55	4.37	1.53	0.68
UDP	No VPP	9.35	9.32	8.98	9.30	9.35
	VPP	9.83	9.51	9.84	9.41	9.50
Protocol	VPP/not VPP	8200km	9300km	9700km	17000km	
TCP	No VPP	0.079	0.0597	0.049	0.041	
	VPP	0.082	0.596	0.048	0.041	
UDP	No VPP	9.31	9.35	9.10	9.10	
	VPP	9.44	9.37	9.43	9.46	

means that it does not have a mechanism to recovery losses and to control the flow of data) and therefore it is not reactive to losses.

Instead TCP has the a different behaviour since it is a closed loop algorithm governed by flow and congestion control, as highlighted in [44]. More over the throughput, as described in 4.2, is inversely proportional to the RTT.

Therefore, the TCP behaviour is justified. When the VPCs are closer, the throughput is higher. Instead, when they are distant, the throughput becomes

smaller. Furthermore, TCP measurements do not reach the same values as UDP ones and we think because Amazon implements inside its cloud environment a shaper, avoiding therefore the links to get congested. This behaviour is also highlighted in [56] and [41].

In Figures 4.9, 4.10, 4.11 we show the behaviour of the two protocols managing different file size. In our experiments also we set up a file size of *1MB,10MB* but we find that those measures are not sufficient to have a clear understanding of the behaviour of the two protocols (for shaper and TCP slow start issues) and therefore we decided to not include the results in this section. However, they are integrally portrayed in the final appendix. The behaviours are similar between each other in all the cases and tables 4.3, 4.4, 4.5 show the actual values.

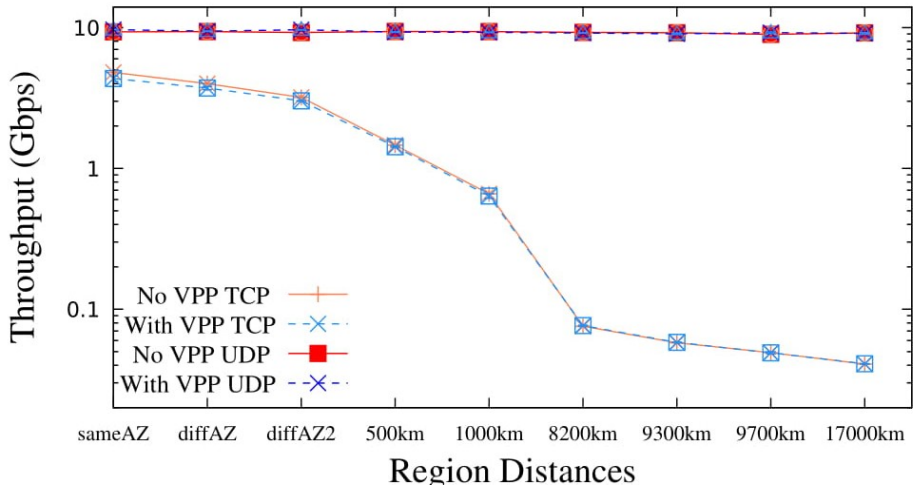

Figure 4.9: Throughput with work size: 100MByte

Regarding the benefits due to the presence of VPP, we notice that with TCP we have few cases in which VPP is better (and this is what we expected before performing the measurements) but in general we observe that we have obtained mixed performances on overall. We notice that the difference between the two TCP throughput is more perceived when the two VPCs are closer. In Tables 4.6 and 4.7 we show the differences in percentages between the two throughput (with VPP and without VPP) in the case of the 20 seconds measurements and with a work load of 5Gbps:

To justify this difference on the TCP throughput measurements, we consider three factors that could affect the TCP performances:

Table 4.3: Throughput with work size 100Mbyte - all values are in Gigabit/sec

Protocol	VPP/not VPP	sameAZ	diffAZ	diffAZ2	500km	1000km
TCP	No VPP	4.80	3.99	3.19	1.46	0.66
	VPP	4.34	3.72	3.02	1.43	0.63
UDP	No VPP	9.33	9.37	9.22	9.38	9.40
	VPP	9.67	9.41	9.64	9.28	9.25
Protocol	VPP/not VPP	8200km	9300km	9700km	17000km	
TCP	No VPP	0.076	0.058	0.0491	0.0408	
	VPP	0.077	0.0582	0.0490	0.041	
UDP	No VPP	9.24	9.18	8.59	9.17	
	VPP	9.13	9.01	9.20	9.07	

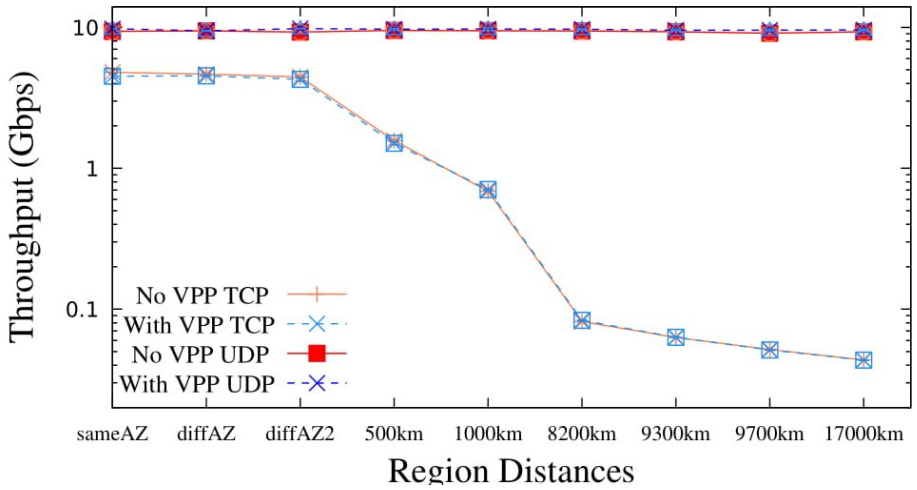

Figure 4.10: Throughput with work size: 1GByte

- packet losses.
- the differences on the header of the packets.
- the RTT.

We start looking at the first option. The losses could affect the throughput since, when a loss happens, TCP, thanks to its congestion control algorithm, set its rate accordingly to the entity of the losses. The more losses we have, the more the throughput will decrease its value. In this case we notice that without VPP the

Table 4.4: Throughput with work size 1Gbyte - all values are in Gigabit/sec

Protocol	VPP/not VPP	sameAZ	diffAZ	diffAZ2	500km	1000km
TCP	No VPP	4.80	4.66	4.44	1.57	0.69
	VPP	4.50	4.52	4.26	1.50	0.70
UDP	No VPP	9.35	9.44	9.26	9.52	9.47
	VPP	9.78	9.43	9.79	9.70	9.73
Protocol	VPP/not VPP	8200km	9300km	9700km	17000km	
TCP	No VPP	0.082	0.062	0.052	0.044	
	VPP	0.083	0.063	0.051	0.043	
UDP	No VPP	9.30	9.31	9.07	9.3	
	VPP	9.69	9.56	9.55	9.60	

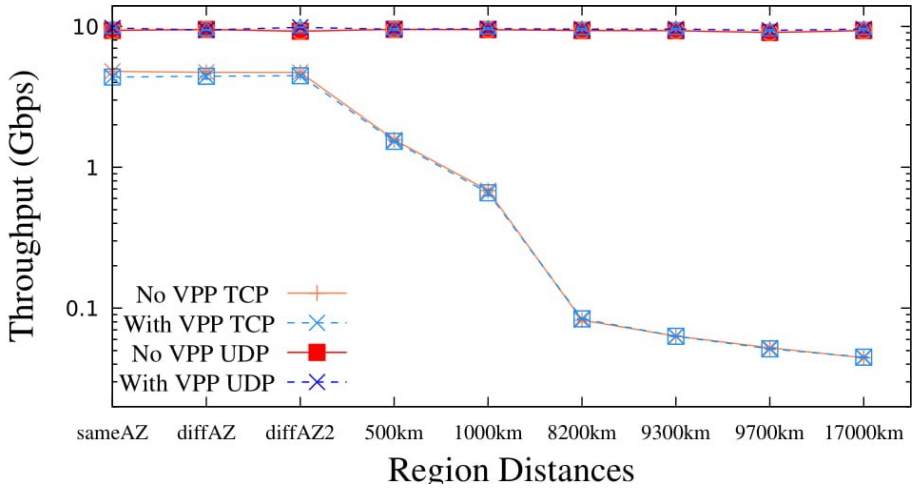

Figure 4.11: Throughput with work size: 5GBytes

number of losses was quite small (in most cases the Retr field in the Iperf3 output in the console, which highlights the number of segments transmitted, is equal to 0) where instead with VPP we experienced some losses. However, we decide to not take measurements because we find them very unreliable. Since the losses depend of various factors, it is hard to us to define which are the causes of the losses and, secondly, to quantify them. They varied a lot during the experiments without following any pattern and further more the losses noticed where not too many to justify a decreasing of the throughput.

Table 4.5: Throughput with work size 5Gbyte - all values are in Gigabit/sec

Protocol	VPP/not VPP	sameAZ	diffAZ	diffAZ2	500km	1000km
TCP	No VPP	4.80	4.72	4.71	1.56	0.68
	VPP	4.36	4.42	4.46	1.53	0.65
UDP	No VPP	9.36	9.50	9.25	9.52	9.49
	VPP	9.72	9.42	9.84	9.53	9.64
Protocol	VPP/not VPP	8200km	9300km	9700km	17000km	
TCP	No VPP	0.082	0.063	0.052	0.044	
	VPP	0.084	0.063	0.051	0.044	
UDP	No VPP	9.34	9.32	9.04	9.35	
	VPP	9.55	9.58	9.36	9.57	

Table 4.6: Differences in percentage between the Throughput without VPP and with VPP in the case of 20 seconds measurements

Protocol	sameAZ	diffAZ	diffAZ2	500km	1000km	8200km
TCP	31%	4.9%	3.4%	3.2%	0%	0%
Protocol	9300km	9700km	17000km			
TCP	2%	2%	0%			

Therefore, since losses in this case are not a reliable evidence, we shift our attention in what, in our opinion, affect most of the throughput difference between the two cases, which are: the differences on the MTU of the packets caused by different packets header size and, secondly, the RTTs. In other words, we could say that the first one is a fixed difference, whereas the second is variable one. We

Table 4.7: Differences in percentage between the Throughput without VPP and with VPP in the case of 5Gbps of work load measurements.

Protocol	sameAZ	diffAZ	diffAZ2	500km	1000km	8200km
TCP	9.2%	6.4%	5.4%	2%	4.5%	0%
Protocol	9300km	9700km	17000km			
TCP	0%	2%	0%			

can write these assumptions in formula (4.3, 4.4):

$$\rho = \frac{\text{MSS'} + |\text{hdr'}|}{\text{MSS} + |\text{hdr}|} \tag{4.3}$$

$$\Delta = \text{RTT}_{\text{VPP}} - \text{RTT}_{!\text{VPP}} \tag{4.4}$$

where:

- ρ tells how much is the difference between the VPP packet *(MSS' + hdr')* and the normal IPv4 packet *(MSS + hdr)*(explained in the next paragraph)
- RTT_{VPP} is the RTT measured using VPP
- $RTT_{!\text{VPP}}$ is the RTT measured without using VPP

Let now check how much these features influence actually the throughput, focusing first to the difference on packet size 4.3.

At the beginning of our experiments we notice that probably, since VPP uses Segment Routing Version 6, an IPv6 header should be located inside the packet, leading to a different packet header with respect to an usual one. We have a first encounter with this assumption when, starting the measurements, we had the constraint to set a maximum MSS equal to 1400 byte in the VPP case, otherwise we would have had several issues in the measurements. The MSS (Maximum Segment Size) is basically the size of the payload (useful data that can be transmitted).

After performed the measurements, we want to see how much the differences between the header of the packets belonging to the two different flows are. We know that for normal TCP packets we have a MTU of 1500 bytes. The MTU (Maximum Transmission Unit) is the whole size of a packet, considering payload and header. The payload is therefore 1460 bytes and as header the packet has 20 bytes of $\text{IP}_{\text{header}}$ and 20 bytes of $\text{TCP}_{\text{header}}$, reaching hence totally 1500 bytes. We can write in formula 4.5:

$$\text{TCP}_{\text{MTU}} = 1460\text{bytes} + 20\ \text{TCP}_{\text{header}} + 20\ \text{IP}_{\text{header}} \tag{4.5}$$

Instead the VPP packet has a different header size and we checked this using Wireshark. In Wireshark we notice that the full header length of the VPP packet is of 80 bytes: 20 bytes of IP header, 20 of TCP header and 40 of IPv6 (which also contains the Segment routing version 6 header). Moreover, we use payload of 1400 bytes to make the experiments work (since we did not know in advance the IPv6 header size). Therefore, in total we have 1480 bytes. In formula 4.6:

$$\text{TCP}_{\text{MTU}} = 1400\text{bytes} + 20\ \text{TCP}_{\text{header}} + 20\ \text{IP}_{\text{header}} + 40\ \text{IPv6}_{\text{header}} \tag{4.6}$$

Hence, let now see how much this difference of packet payload and overhead could affect the performances, without considering the RTT for the moment. We define the formula 4.7:

$$\mathrm{TH}_{\mathrm{VPP}} = (1 - \epsilon)\mathrm{TH}_{!\mathrm{VPP}} \tag{4.7}$$

where:

- TH_{VPP} is the Throughput obtained using VPP
- $TH_{!\mathrm{VPP}}$ is the Throughput obtained without using VPP
- ϵ is a factor, evaluated by dividing the two packets' MTU. It has a constant value of 0.986 (1.4% of differences)

Using formula 4.7 we are interested to investigate how much the difference of the packets size could affect the throughput. Hence we have multiplied the throughput measured without VPP with a factor that exploits the difference on the two packets size. For this moment we put the constraint to use the same RTT for the two throughput, since we want to exploit only the packet size difference. In Table 4.8 we show 3 different throughput, based on the case of 20 seconds measurements: in order, the first row is the throughput evaluated without VPP, the second row is the one evaluated with our theoretical formula 4.7 where instead the third row is the actual throughput with VPP.

Table 4.8: In this table we portrait the throughput results without using VPP, wih VPP and the ones evaluated with the formula proposed above. Measurements all in Gbps.

Throughput TCP	Same AZ	Diff AZ	Diff AZ2	500km	1000km
Not VPP	4.77	4.78	4.52	1.58	0.68
Theoretical VPP	4.70	4.71	4.45	1.56	0.67
with VPP	3.32	4.55	4.37	1.53	0.68
Throughput TCP	8200km	9300km	9700km	17000km	
Not VPP	0.079	0.0597	0.049	0.041	
Theoretical VPP	0.078	0.0590	0.048	0.040	
with VPP	0.082	0.0596	0.048	0.041	

Furthermore, in table 4.9 we put which are the differences in percentage between the theoretical throughput and the throughput evaluated with VPP.

Looking at tables 4.8, 4.9, we can say that the difference in percentage between the measures with the theoretical VPP, which is the one we evaluated from our

Table 4.9: Percentage of the differences between the theoretical throughput and the throughput evaluated with VPP.

Protocol	Same AZ	Diff AZ	Diff AZ2	500km	1000km
TCP	29.4%	3.4%	1.8%	2%	0%
Protocol	8200km	9300km	9700km	1700km	
TCP	0%	0%	0%	0%	

formula 4.7, and the actual VPP, evaluated from our real measurements, is still huge when the two VPCs are closer. Instead, the more the latters become distant, the more the differences became meaningless and, for our purposes, unreliable. Hence the difference of the packet size has not a meaningful impact as we expected because, since we are not considering the RTT role in this evaluation, we would have expected that the difference in percentage between the two throughput when the VPCs are closer, should have been smaller (highlighting the role of packet size difference).

More over to demonstrate how much RTT could affect the performance and that the size difference is meaningless, we did an experiment setting the MTU to 1300 in both cases, with and without VPP, to see how much was the differences. In Figure 4.12 we show the performances measured where instead in Table 4.10 the actual values obtained. As we expected within the same Region we still have a huge gap between the measures but with different regions we have instead similar ones, due to almost the same RTT. Therefore it would be interesting to focus on our second assumption.

Table 4.10: Values obtained with same MTU (in Gbps)

TCP	Intra Cloud	Inter Cloud
Without VPP	4.75	0.662
With VPP	4.55	0.660

In Figure 6.1 and 6.2 we have a deeper look of the differences of the throughput versus the RTT. We notice that in both cases we have that the curve belonging to VPP measurements has always an higher RTT value rather then the not VPP one but at the same time the latter has always an higher throughput on the Figure 6.1 while on the Figure 6.2 the throughput, expect for one measurements, is the same:

Hence, since previously we said that the difference of packet size does not play a so important role to justify this behaviour and, looking also at 1.5 where we show the difference in μs between the RTT with VPP and the RTT without VPP, we could highlight that this difference between RTTs values play an huge role in

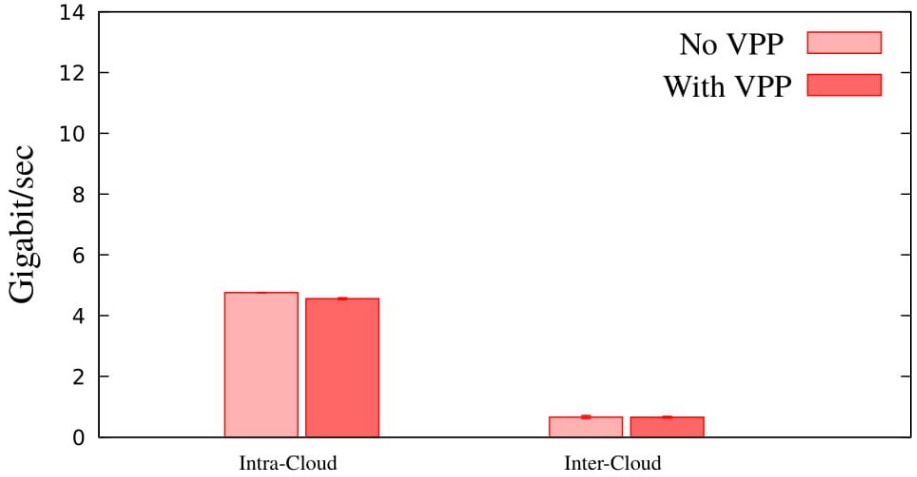

Figure 4.12: Throuhput using the same MTU

determining the throughput of the two flow, especially when the two VPCs are close. In fact, even though on average the difference of RTT values is ≈ 63 μs, this difference clearly play an important role in the throughput evaluation, especially with small RTTs, as also highlighted previously in the formula (4.2) and double checking with Table 4.1 referring to the RTT values. When instead the RTTs have bigger values, the difference is still present but it does not play an important role: in fact the throughput values are almost the same and in some cases VPP outperforms the not VPP measurements.

Previously, in figure 4.5 we highlighted the difference of the two RTT flows with an overall value: ≈ 63 μs. Let now define better what agents enter in the evaluation of the value. Using formula 4.8 we say that:

$$\mathrm{Th}_{\mathrm{VPP}}(\mathrm{RTT}) = \mathrm{TH}_{!\mathrm{VPP}}(\mathrm{RTT}_{!\mathrm{VPP}} + \mathrm{d} + \xi) \tag{4.8}$$

where:

- $RTT_{!\mathrm{VPP}}$ is the RTT of a packet without VPP
- d is the path that the SRv6 has to do more with respect to the IPv4 packet
- ξ is an overhead due to the processing time

Let focus on the last two agents. d is present since it is the path that the VPP packet has to do more. For example when a packet is ejected from the client

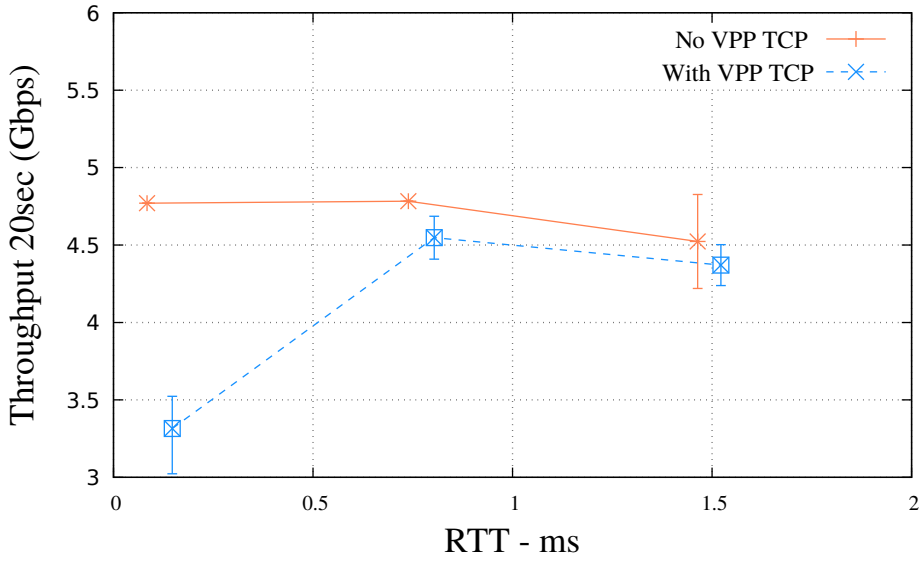

Figure 4.13: Throughput against RTT with same Region

Figure 4.14: Comparison between Throughput and RTT in same/Different Region

Virtual Machine, it has to go firstly to the router of the VPC, then to the VPP machine where it is encapsulated in an IPv6 packet and then again it has to go though the VPC internet gateway. Afterwards we don't know if the path across internet is the same for the two packets belonging to the two different flows but certainly when it arrives at the destination VPC, it has to do again the previous steps, making hence a longer path. Figure 4.6 shows our assumption. Ideally, the best scenario in which there are not other delay, d in on order of ≈ 33 μs, where ≈ 8.25 μs is the time taken to go out from the VPC).

ξ is instead the processing time due to the batch that VPP processes. The batch processing may impact the latency that packets experience in a VPP router, due to the fact that the forwarding operation occurs only after a full graph traversal, and since several new nodes are pushed into the processing graph (related to IPv6 and SRv6) we could say that it could have an impact on the overall processing time [37]. However looking at some statistics inside VPP software, we discovered that the software router process 4 packets per batch at maximum, with an average processing of one packet per batch. Hence VPP is not working at its optimum capacity, since it could process a batch of 256 packets. As future work we would like to investigate further on why this happened inside the Amazon Cloud environment.

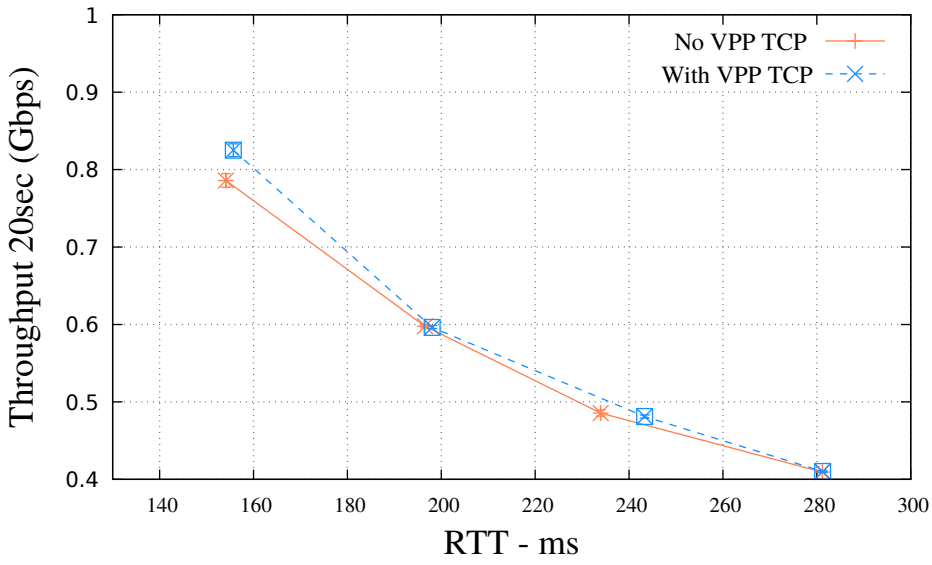

Figure 4.15: Throughput and RTT with different Regions

Figure 4.16: Comparison between Throughput against RTT in same/Different Region

Finally, in figure 4.17 we show which are the clock cycles dedicated per node inside VPP, in both case when we are in the same region or not and with two different packet size.

From 4.17 we can say that on average the cost clock cycle performance are higher when the measurements are taken inside the same region (first and third column) and when the packets have a size of 512 bytes.

Instead, with UDP we are free from the RTT constraint and we observe different behaviours. When the packets use UDP protocol we notice that, referring on figures 4.9, 4.10, 4.11, in both cases (20sec measurements and high loads) UDP saturates the link, even though with VPP presence the latter has slightly better performance (on average it is in the order of $\approx 3\%$). We also discover that UDP, inside VPCs, use Jumbo frame to carry data. A Jumbo frame [47] is an Ethernet frame which has a playload that could be bigger than 1500 bytes, carrying up to 9000 bytes. Instead with small work size, the performances are better without VPP, even though we find these measurements not reliable because the work size is too small to gather significant values.

Finally, we perform some experiments with UDP using also the same MTU,

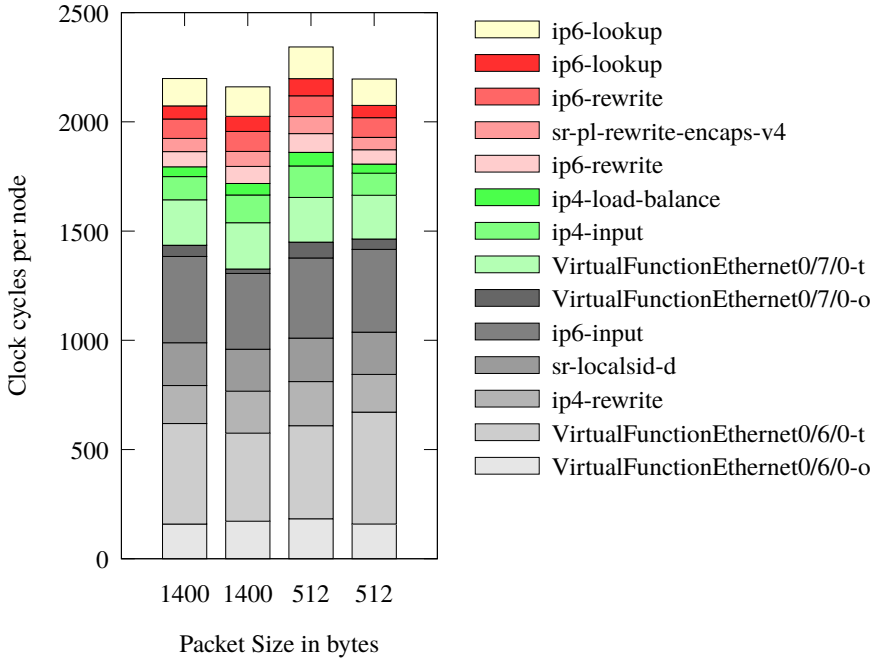

Figure 4.17: Clock cyle per node for different Packet Size. The left columns are intended inside the same region, while the right one with different region

to see if there are any pattern with the other results. Even in this case, UDP with VPP performs better with respect to measure without VPP. In Figures 4.18 and 4.19 we show the two behaviours:

4.4 Shaper

One of most interesting behaviour we find performing our experiments, and which is only mentioned in [41], inside the papers related to Amazon's Cloud measurements, is that all our TCP sources were constrained by a shaper. We know that TCP is a closed loop algorithm [44] which is influenced by the path and by the network congestion, but when we looked at the results, the presence of a shaper was not expected. A shaper is a technique used to perform traffic conditioning, having hence traffic that respect the desired traffic profile. We noticed it because, for example, the instances could support traffic up to 10 Gigabit/sec but with TCP it never reaches it those peaks. At maximum it reaches a speed of 5 Gigabit/s.

Figure 4.18: UDP measurements with different regions with same MTU

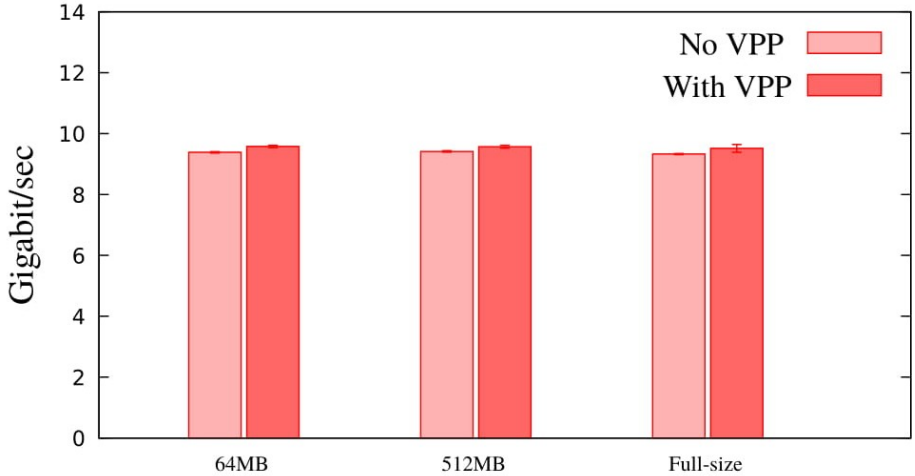

Figure 4.19: UDP measurements within same regions with same MTU

Probably this is due to an internal Amazon policy, to avoid congestion problems and it is a new feature, which was not present in the other years and that could change over time, depending on Amazon's goals. More over, an interesting behaviour is

when we have multiple flows.

4.4.1 Multiple flows

When we have multiple flows directed from our client to the server we notice several different behaviours. Inside the different region scenario, in the UDP case the bandwidth is equally divided by the number of flows. In formula:

$$\text{Bandwidth UDP}_{\text{per flow}} = \frac{\text{Total Bandwidth}}{\text{Number of Flows}} = \frac{9.7 \text{ Gigabit/sec}}{10} \tag{4.9}$$

Instead, with TCP, we have that the maximum bandwidth reached is 5 Gigabit/sec and it is fairly shared among the flows.

$$\text{Bandwidth TCP}_{\text{ per flow}} = \frac{\text{Total Bandwidth}}{\text{Number of Flows}} = \frac{4.8 \text{ Gigabit/sec}}{10} \tag{4.10}$$

VPP has generally worse performance since it has many losses with respect to the TCP flows without VPP. It is interesting to notice that before, in the same scenario but with only one flow passing from the client to the server, we have a peak of 600Mbit/sec. Now instead, with several flows, we can reach a maximum of 5 Gigabit/sec (summing all the flows) and this means that, for this scenario, the shaper allows to allocate a maximum of 5 Gigabit/sec bandwidth for each instance.

Instead, if we look at the same region scenario we have that UDP has the same behaviour as in 4.9 but in this case also TCP saturates the links with both flows (VPP and not VPP), even though with VPP we have more losses.

Chapter 5

Conclusion

5.1 Summary

This work is the first, as far as we know, that implements an high speed software router inside a Public Cloud provider (Amazon) and automate its deployment through a Terraform script. Both the VPP machine and Terraform script will be available in the internet: the software router is already publicly available inside the FD.io web page [25] but, in addition, we will create a AMI with already VPP installed, which will be released inside AWS marketplace. Therefore it would be easier to deploy it inside a VPC using our Terraform script developed. Instead the second one will be released publicly in the internet inside a repository. Afterwards, we propose a methodology to measure the performances of VPP inside the complex scenario of Cloud Computing. We were able to gather useful information about the network performance of Amazon's cloud and compare them with some of the previous work. Furthermore, we find out that there is a shaper working,limiting the TCP traffic. Even though at the end we obtain similar performance with the measurements with and without VPP, with hence a differences of 2-3% on average, we insert inside AWS one of the most prominent and new software router with all its novelties, flexibility and wide area usages. One of the novelties is the new protocol *Segment Routing Version 6* which allows, for instance, the possibility to perform Service Chaining in a multi-cloud overlays environment, something which was not present before and which it is becoming more and more important in data-centers. Moreover it exploits IPv6 protocol, which is slowly replacing the IPv4 addresses inside internet. This study represents only the first step on what could be some possible future works.

5.2 Future Work

As future work, we could exploit different paths for improving and refining our work. First of all, it would be interesting to see if the same environment tested inside Amazon would work also in other Public Cloud Providers, such as *Google Cloud* or *Azure*, even though they do not offer the same features of AWS, making the VPP deploment harder but, if it works, we could modify our Terraform script, since it is Cloud-agnostic, to automate also the VPP deployment inside the other clouds and afterwards performs more measurements, comparing also with the results we gathered before. More over we are interested to see if we could connect together two VPCs belonging to different Public Cloud Providers.

Finally, we are interested to investigate further on the results we obtained or maybe to perform more sophisticated measure to get a clearer image on how the shaper actually works and if it possible somehow to improve the TCP performances with VPP presence (maybe substituting the Amazon router with VPP). On the other hand, it would interesting to find out why UDP with VPP have always outperformed and then we would like to perform these same experiments in another cloud environment to find any patterns.

Chapter 6

Appendix

6.1 VPP commands

In this section we show the commands used to configure the second VPP machine inside another VPC and how we connect them together using SRv6.

```
cd /
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
sudo apt-get install python-pip
sudo apt-get install libnuma-dev
sudo apt-get install make

sudo wget https://fast.dpdk.org/rel/dpdk-18.02.1.tar.xz
sudo tar -xvf dpdk-18.02.1.tar.xz

cd /
sudo mkdir /vpp
sudo chmod 777 /vpp
cd /vpp
git clone https://gerrit.fd.io/r/vpp ./
sudo make install-dep
sudo make build
cd build-root
make V=0 PLATFORM=vpp TAG=vpp install-deb

export UBUNTU="xenial"
```

```
export RELEASE=".stable.1804"
sudo rm /etc/apt/sources.list.d/99fd.io.list
echo "deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.
    ↪ io$RELEASE.ubuntu.$UBUNTU.main/ ./" |
sudo tee -a /etc/apt/sources.list.d/99fd.io.list
sudo apt-get update
sudo apt-get install vpp vpp-lib
sudo apt-get install vpp-plugins vpp-dbg vpp-dev vpp-api-java vpp-
    ↪ api-python vpp-api-
lua

cd /vpp/build-root/build-vpp_debug-native/dpdk/dpdk-stable-18.02.1
sudo modprobe uio
sudo make install T=x86_64-native-linuxapp-gcc
sudo insmod /vpp/build-root/build-vpp_debug-native/dpdk/dpdk-stable
    ↪ -18.02.1/x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
sudo /vpp/build-root/build-vpp_debug-native/dpdk/dpdk-stable
    ↪ -18.02.1/usertools/dpdk-devbind.py --bind igb_uio
    ↪ 0000:00:06.0
sudo /vpp/build-root/build-vpp_debug-native/dpdk/dpdk-stable
    ↪ -18.02.1/usertools/dpdk-devbind.py --bind igb_uio
    ↪ 0000:00:07.0

sudo service vpp stop
sudo service vpp start

sudo /sbin/ip -4 addr add 10.2.5.190/24 dev ens6
sudo ifconfig ens6 up
sudo /sbin/ip -4 route add 10.2.0.0/16 via 10.2.5.21

set int state VirtualFunctionEthernet0/6/0 up
set int state VirtualFunctionEthernet0/7/0 up
set int ip address VirtualFunctionEthernet0/6/0 10.2.5.21/24
set int ip address VirtualFunctionEthernet0/7/0 2600:1f14:135:cc00
    ↪ :13b9:ff74:348d:7642/64
set sr encaps source addr 2600:1f14:135:cc00:13b9:ff74:348d:7642
sr localsid address 2600:1f14:135:cc00:43c1:e860:7ce9:e94a behavior
    ↪ end.dx4 VirtualFunctionEthernet0/6/0 10.2.5.190
sr policy add bsid c:3::999:1 next 2600:1f14:e0e:7f00:8da1:c8fa
    ↪ :5301:1d1f encap
sr steer l3 10.1.4.0/24 via sr policy bsid c:3::999:1
```

```
set ip6 neighbor VirtualFunctionEthernet0/7/0 fe80::86a:b7ff:fe5d:73
    ↪ c0 0a:4c:fd:b8:c1:3e
ip route add ::/0 via fe80::86a:b7ff:fe5d:73c0
    ↪ VirtualFunctionEthernet0/7/0
```

6.2 Terraform Script

In this section we portrait the whole Terraform script used to automate the deployment of our configuration:

```
#VARIABLES

variable "root_password" {}
#variable "access_key" {}
#variable "secret_key" {}
variable "SUBNET" {}
#PROVIDER AMAZON

provider "aws" {
  access_key = "AKIAJPT5IFDH2FVODCZA"
  secret_key = "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"
  region = "eu-central-1"

}


#provider "aws" {
# access_key = "${var.access_key}"
# secret_key = "${var.secret_key}"
 # region = "eu-central-1"

#}


#VPC


resource "aws_vpc" "India" {
  cidr_block = "10.7.0.0/16"
  instance_tenancy = "dedicated"
        assign_generated_ipv6_cidr_block = true
        enable_dns_hostnames =true
 tags {
    Name = "India"
  }
```

```
}

#SUBNETS

resource "aws_subnet" "vpp" {
  vpc_id = "${aws_vpc.India.id}"
  cidr_block = "10.7.1.0/24"
        availability_zone = "eu-central-1a"
  tags {
    Name = "vpp"
  }
}

resource "aws_subnet" "management" {
  vpc_id = "${aws_vpc.India.id}"
  cidr_block = "10.7.2.0/24"
        availability_zone = "eu-central-1a"
  tags {
    Name = "management"
  }
}


resource "aws_subnet" "ipv6" {
  vpc_id = "${aws_vpc.India.id}"
  cidr_block = "10.7.3.0/24"
  ipv6_cidr_block = "${cidrsubnet(aws_vpc.India.ipv6_cidr_block, 8,
      ↪ 1)}"
  availability_zone = "eu-central-1a"
    assign_ipv6_address_on_creation = true
    tags {
    Name = "ipv6"
  }
}

#INTERNET GATEWAY

resource "aws_internet_gateway" "vpc_igw" {
    vpc_id = "${aws_vpc.India.id}"

  tags {
    Name = "main"
```

```
  }
}

#ROUTE TABLES


resource "aws_default_route_table" "rt" {

    default_route_table_id = "${aws_vpc.India.default_route_table_id
        ↪ }"

    route {
        cidr_block = "0.0.0.0/0"
        gateway_id = "${aws_internet_gateway.vpc_igw.id}"
    }

    route {
        ipv6_cidr_block = "::/0"
        gateway_id = "${aws_internet_gateway.vpc_igw.id}"
    }
}

#resource "aws_route_table_association" "association1" {

 # subnet_id = "${aws_subnet.vpp.id}"
 # route_table_id = "${aws_default_route_table.rt.id}"
#}

resource "aws_route_table_association" "association2" {

    subnet_id = "${aws_subnet.management.id}"
    route_table_id = "${aws_default_route_table.rt.id}"
}

resource "aws_route_table_association" "association3" {

    subnet_id = "${aws_subnet.ipv6.id}"
    route_table_id = "${aws_default_route_table.rt.id}"
}

resource "aws_route_table" "r" {
  vpc_id = "${aws_vpc.India.id}"
```

```
  route {
    cidr_block = "${var.SUBNET}"
    network_interface_id = "${aws_network_interface.vppIPV4.id}"
  }

route {
       cidr_block = "0.0.0.0/0"
       #ipv6_cidr_block = "::/0"
       gateway_id = "${aws_internet_gateway.vpc_igw.id}"
    }

  tags {
    Name = "India"
  }
}

resource "aws_route_table_association" "association" {
  subnet_id = "${aws_subnet.vpp.id}"
  route_table_id = "${aws_route_table.r.id}"
}



#SECURITY GROUP

resource "aws_security_group" "allow_all" {
  name = "allow_all"
  description = "Allow all inbound traffic"
  vpc_id = "${aws_vpc.India.id}"

  ingress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }

  egress {
    from_port = 0
    to_port = 0
```

```
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }

 tags {
    Name = "allow_all"
  }

}


#INSTANCE WITH VPP CREATED BY ME


resource "aws_instance" "VPP-India" {
  ami = "ami-574d49bc"
  instance_type = "m5.2xlarge"
  key_name = "VPP_India"
  vpc_security_group_ids= ["${aws_security_group.allow_all.id}"]
#security_groups = ["${aws_security_group.allow_all.id}"]
       subnet_id="${aws_subnet.management.id}"
       availability_zone = "eu-central-1a"
user_data = "${data.template_file.user-data.rendered}"


#timeouts {
# create = "60m"

# }

tags {
    Name = "VPP-India"
  }

}

data template_file "user-data"
{
    template = "${file("script_interfaces.sh")}"
}
```

```
#ASSIGN KEYPAIR


#NETWORK INTERFACES

resource "aws_network_interface" "vppIPV4" {
  subnet_id = "${aws_subnet.vpp.id}"

  security_groups = ["${aws_security_group.allow_all.id}"]
  source_dest_check = false
  attachment {
    instance = "${aws_instance.VPP-India.id}"
    device_index = 1
  }
}


resource "aws_network_interface" "vppIPV6" {
  subnet_id = "${aws_subnet.ipv6.id}"

  security_groups = ["${aws_security_group.allow_all.id}"]
   source_dest_check = false
  attachment {
    instance = "${aws_instance.VPP-India.id}"
    device_index = 2
  }
}



#ASSIGN EIP

resource "aws_eip" "ip" {
  instance = "${aws_instance.VPP-India.id}"
}
```

6.3 Other Results

In this section we portrait some of our other figures made during our experiments. Firstly, we show the throughput measured using as work load size 1 Mbyte, with a normal y scale then a logarithmic y scale (figures 6.1 and 6.2). Afterwards, we focus on the measurements with work load size of 10Mbytes, always with both different y scales (figures 6.3 and 6.4). We find both these measurements unreliable since we notice that they are affected by the shaper action and the load small size.

Then we show figures 6.5, 6.6 and 6.7 with the throughput evaluated in the case of 100Mbyte, 1 Gbyte and 5 Gbyte with instead the normal y scale. We decided to show in our work only the figures (4.9, 4.10 and 4.11) with a logarithmic scale since they allow to highlight better the differences.

Finally, we show the TCP and UDP behaviour comparing together all the several work load scenario within the same region and not (figures 6.8, 6.9, 6.10 and 6.11):

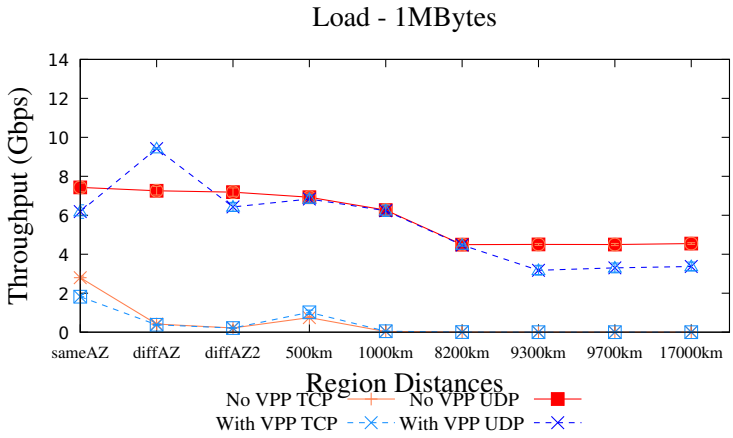

Figure 6.1: Normal Scale

.

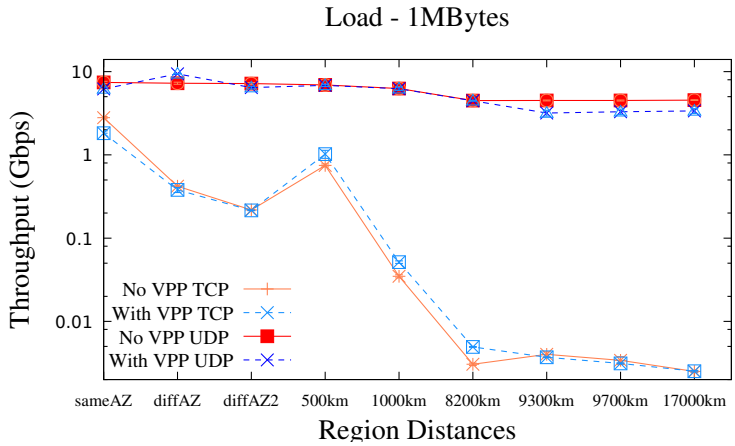

Figure 6.2: Logarithmic Scale

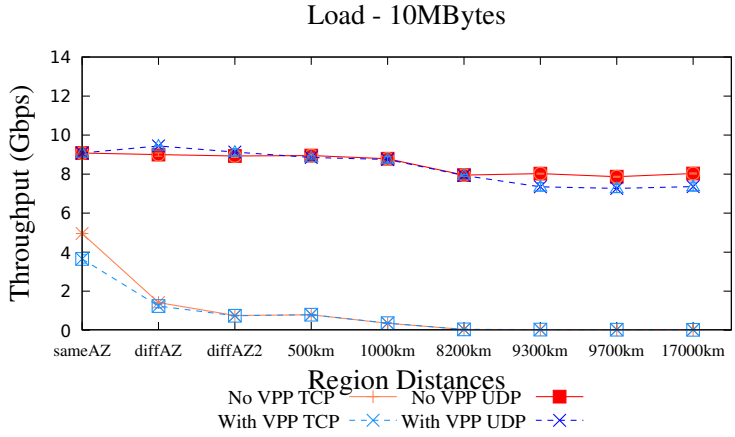

Figure 6.3: Normal Scale

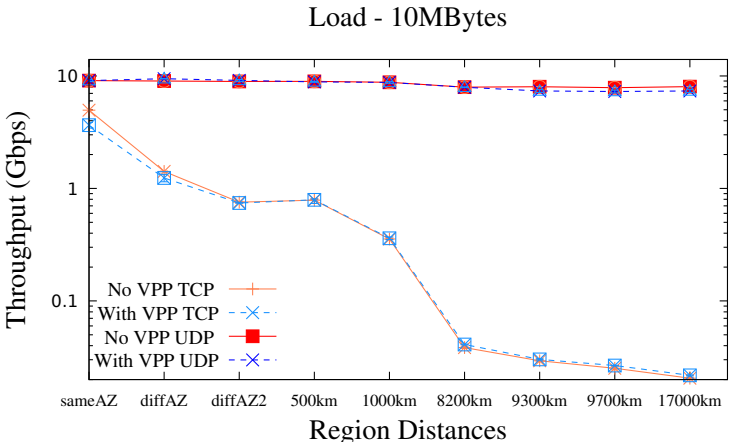

Figure 6.4: Logarithmic Scale

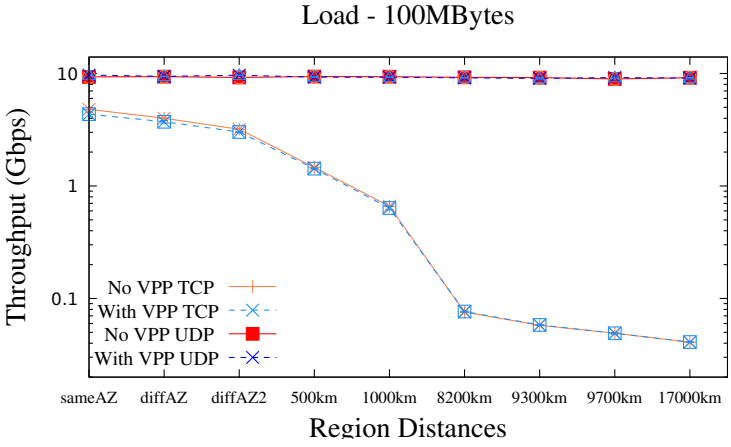

Figure 6.5: Throughput evaluated with work size: 100 Mbyte.

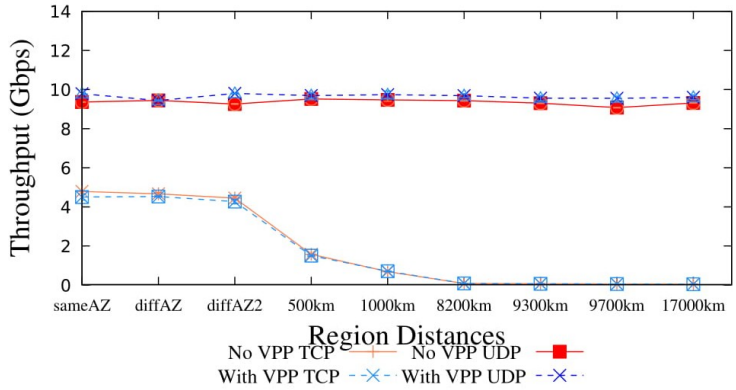

Figure 6.6: Throughput with work size: 1Gbyte

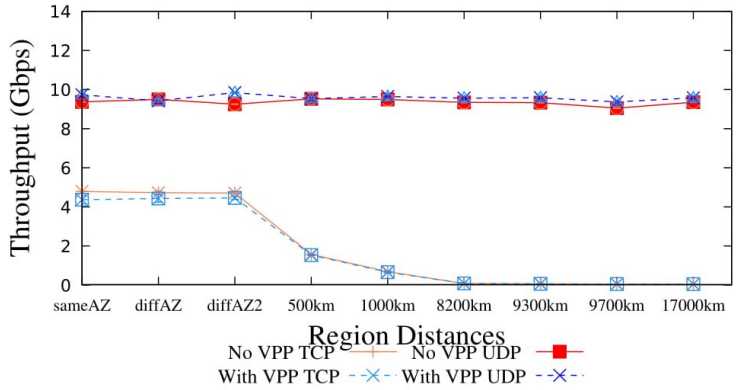

Figure 6.7: Throughput evaluated with work size: 5 Gbyte

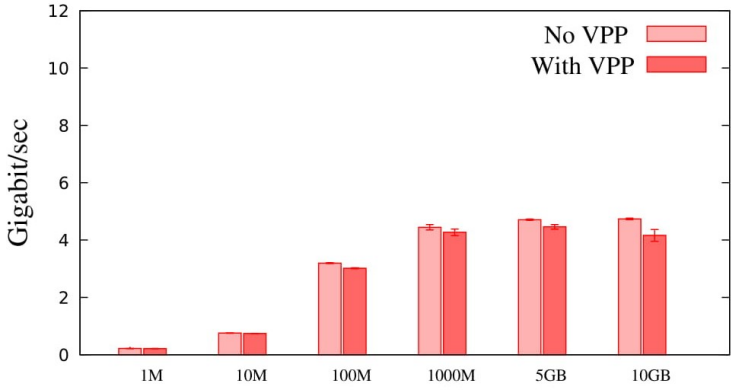

Figure 6.8: TCP throughput evaluated within same Region (Tokyo - Tokyo)

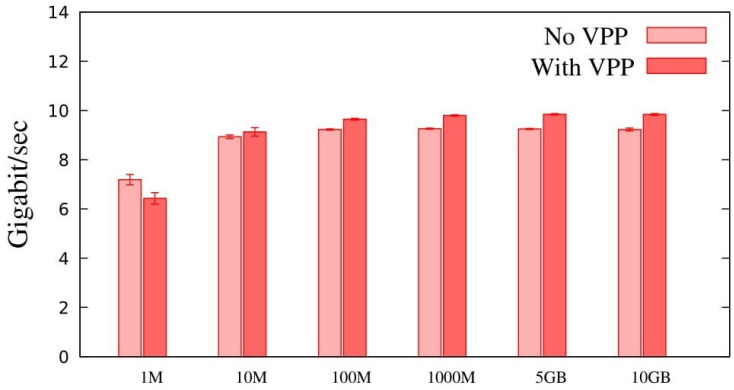

Figure 6.9: UDP throughput evaluated within same Region (Tokyo - Tokyo)

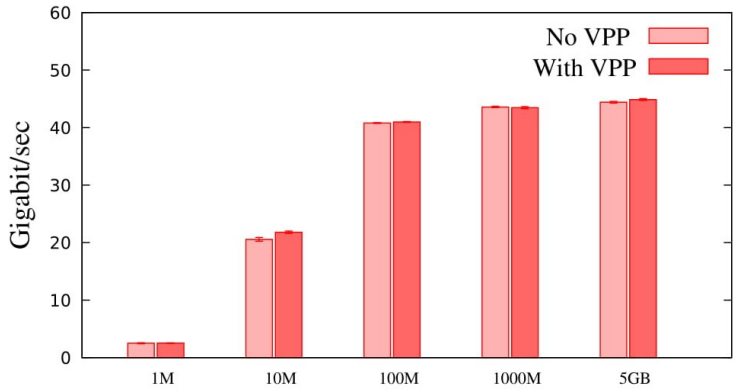

Figure 6.10: Throughput evaluated with work size: 5 Gbyte

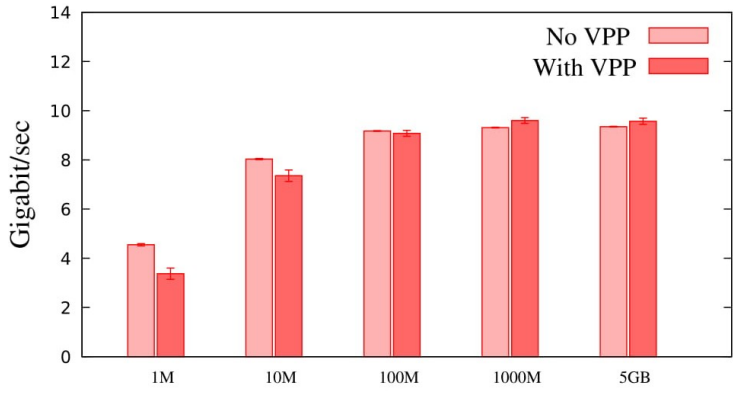

Figure 6.11: Throughput evaluated with work size: 5 Gbyte

Acknowledgements

This work has been carried out during Francesco Spinelli's internship at LINCS (https://www.lincs.fr/) and benefited from support of NewNet@Paris, Cisco's Chair "Networks for the Future" at Telecom ParisTech (https://newnet.telecom-paristech.fr)

I would like to thank Prof. Dario Rossi for giving me the wonderful opportunity to work with his team inside the NewNe@Paris context, Prof. Paolo Giaccone for guiding me during the internship and the thesis drafting and Jerome Tollet, who has been very welcoming and always available to help me during my internship.

Bibliography

[1] Giuseppe Aceto et al. “Cloud monitoring: A survey”. In: 57 (June 2013), 2093â2115.

[2] arm Developer. *AArch64 virtualization*. URL: https://developer.arm.com/products/architecture/cpu-architecture/a-profile/docs/100942/latest/aarch64-virtualization.

[3] N. Asthana et al. “A declarative approach for service enablement on hybrid cloud orchestration engines”. In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. 2018, pp. 1–7. DOI: 10.1109/NOMS.2018.8406175.

[4] AWS. *Amazon Web Series WebSite*. URL: https://aws.amazon.com.

[5] Amazon AWS. *AWS CloudâFormation*. URL: https://aws.amazon.com/it/cloudformation/.

[6] T. Barbette, C. Soldani, and L. Mathy. “Fast userspace packet processing”. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2015, pp. 5–16. DOI: 10.1109/ANCS.2015.7110116.

[7] AWS Blog. *Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta*. 2006. URL: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.

[8] *Boto 3 Documentation*. 2014. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[9] J. Leddy D. Voyer D. Bernier F. Clad P. Camarillo Ed. C. Filsfils Z. Li. *SRv6 Network Programming*. 2018. URL: https://tools.ietf.org/html/draft-filsfils-spring-srv6-network-programming-04.

[10] J. Leddy S. Matsushima C. Filsfils S. Previdi. *D. Voyer*. 2018. URL: https://tools.ietf.org/html/draft-ietf-6man-segment-routing-header-14.

[11] S. Callanan, D. O'Shea, and E. O'Regan. "Automated Environment Migration to the Cloud". In: *2016 27th Irish Signals and Systems Conference (ISSC)*. 2016, pp. 1–6. DOI: 10.1109/ISSC.2016.7528471.

[12] SDX central. *What is Network Service Chaining? Definition*. URL: https://www.sdxcentral.com/sdn/network-virtualization/definitions/what-is-network-service-chaining/.

[13] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. First. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780132349710.

[14] Inc Cisco Systems. *Cisco Cloud Services Router (CSR) 1000V - Bring Your Own License (BYOL)*. 2018. URL: https://aws.amazon.com/marketplace/pp/B00EV8VWWM.

[15] Compaq. *Internet Solutions Division Strategy for Cloud Computing*. 1996. URL: https://s3.amazonaws.com/files.technologyreview.com/p/pub/legacy/compaq_cst_1996_0.pdf.

[16] E. Dresselhaus D. Barach. *Vectorized software packet forwarding*. 2011.

[17] AWS Documentation. *Amazon Virtual Private Cloud*. URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.

[18] Netgate Documentation. *pfSense Firewall/VPN/Router for AWS*. 2018. URL: https://www.netgate.com/docs/pfsense/solutions/aws-vpn-appliance/.

[19] R. Droms. *Dynamic Host Configuration Protocol*. RFC 2131 (Draft Standard). Updated by RFCs 3396, 4361, 5494. Internet Engineering Task Force, Mar. 1997. URL: http://www.ietf.org/rfc/rfc2131.txt.

[20] S. DrÃ¤xler et al. "SONATA: Service programming and orchestration for virtualized software networks". In: *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2017, pp. 973–978. DOI: 10.1109/ICCW.2017.7962785.

[21] ETSI. *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action*. Vol. Introductory White Paper. ETSI.

[22] FD.io. *VPP*. URL: https://wiki.fd.io/view/VPP.

[23] FD.io. *VPP/Segment Routing for IPv6*. 2017. URL: https://wiki.fd.io/view/VPP/Segment_Routing_for_IPv6.

[24] The Linux foundation. *DPDK*. URL: https://www.dpdk.org/.

[25] The Linux Foundation. *FD.io - The Fast Data Project*. 2018. URL: https://fd.io/.

[26] Ole TrÃ¸an Pablo Camarillo fd.io JJBtechopedia Francesco Spinelli Jerome Tollet. *Implement DHCPv6 IA NA client (VPP-1094)*. 2018. URL: https://gerrit.fd.io/r/\#/c/12599/.

[27] Paolo Giaccone. *Software Defined Networking and the design of OpenFlow switches*. 2015. URL: https://pdfs.semanticscholar.org/presentation/f608/e454b709aed8cf47fbd46e3a90d2927de804.pdf.

[28] HashiCorp. *Introduction to Terraform*. URL: https://www.terraform.io/intro/index.html.

[29] Yun Chao Hu et al. "Mobile edge computingâA key technology towards 5G". In: *ETSI white paper* 11.11 (2015), pp. 1–16.

[30] Michael Hüttermann. "Infrastructure as Code". In: *DevOps for Developers*. Berkeley, CA: Apress, 2012, pp. 135–156. ISBN: 978-1-4302-4570-4. DOI: 10.1007/978-1-4302-4570-4_9. URL: https://doi.org/10.1007/978-1-4302-4570-4_9.

[31] Marco Mellia Maurizio Munafo' Ignacio Bermudez Stefano Traverso. "Exploring the Cloud from Passive Measurements: the Amazon AWS Case". In: *Proceedings IEEE INFOCOM*. INFOCOM'13. 2013.

[32] Vahi Karan Mehta Gaurang Berriman Bruce Berman Benjamin P Maechling Phil Juve Gideon Deelman Ewa. "Scientific Workflow Applications on Amazon EC2". In: *2009 5th IEEE International Conference on E-Science Workshops*. IEEE. 2009.

[33] Eddie Kohler et al. "The Click Modular Router". In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: http://doi.acm.org/10.1145/354871.354874.

[34] Jozsef Kovacs and Peter Kacsuk. "Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures". In: 16 (Nov. 2017), pp. 1–19.

[35] Diego Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103 (2014), pp. 14–76.

[36] Katrina LaCurts et al. "Choreo: Network-aware Task Placement for Cloud Applications". In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 191–204. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504744. URL: http://doi.acm.org/10.1145/2504730.2504744.

[37] Salvatore Pontarelli Dave Barach Damjan Marjon Pierre Pfister Leonardo Linguaglossa Dario Rossi. *High-speed Software Data Plane via Vectorized Packet Processing extended version*. Tech. rep. 2018.

[38] Ang Li et al. "CloudCmp: Comparing Public Cloud Providers". In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: ACM, 2010, pp. 1–14. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879143. URL: http://doi.acm.org/10.1145/1879141.1879143.

[39] Oracle. *Virtual Box*. URL: https://www.virtualbox.org/.

[40] Hilary Osborne. *What is Cambridge Analytica? The firm at the centre of Facebook's data breach*. 2018. URL: https://www.theguardian.com/news/2018/mar/18/what-is-cambridge-analytica-firm-at-centre-of-facebook-data-breach.

[41] Valerio Persico et al. "Measuring Network Throughput in the Cloud". In: *Comput. Netw.* 93.P3 (Dec. 2015), pp. 408–422. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2015.09.037. URL: http://dx.doi.org/10.1016/j.comnet.2015.09.037.

[42] Timothy Grance Peter Mell. "The NIST Definition of Cloud Computing". In: NIST Special Publication.800-145 (2011).

[43] *ping(8) - Linux man page*. URL: https://linux.die.net/man/8/ping.

[44] J. Postel. *Transmission Control Protocol*. RFC 793 (Standard). Updated by RFCs 1122, 3168. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[45] J Postel. "User Datagram Protocol". In: *RFC 768* (1980).

[46] Luigi Rizzo. "Netmap: A Novel Framework for Fast Packet I/O". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=2342821.2342830.

[47] Gaurav Chawla Anthony Faustini Robert Winter Rich Hernandez. "Ethernet Jumbo Frames". In: Ethernet Jumbo Frames (2009).

[48] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 460–471. ISSN: 2150-8097. DOI: 10.14778/1920841.1920902. URL: http://dx.doi.org/10.14778/1920841.1920902.

[49] Amazon Web Services. *Amazon Machine Images (AMI)*. URL: https://docs.aws.amazon.com/en_us/AWSEC2/latest/UserGuide/AMIs.html.

[50] Arjun Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *Commun. ACM* 59.9 (Aug. 2016), pp. 88–97. ISSN: 0001-0782. DOI: 10.1145/2975159. URL: http://doi.acm.org/10.1145/2975159.

[51] Francesco Spinelli. *How to deploy VPP in EC2 instance and use it to connect two different VPCs*. 2018. URL: https://wiki.fd.io/view/How_to_deploy_VPP_in_EC2_instance_and_use_it_to_connect_two_different_VPCs.

[52] Cisco Systems. *Cisco CloudCenter Solution*. URL: https://www.cisco.com/c/dam/en/us/products/collateral/cloud-systems-management/cloudcenter/at-a-glance-c45-737051.pdf.

[53] techopedia. *Definition - What does Plumbing mean?* URL: https://www.techopedia.com/definition/31509/plumbing.

[54] The Open Networking Foundation. *OpenFlow Switch Specification*. 2012.

[55] Ajay Tirumala et al. "iPerf: TCP/UDP bandwidth measurement tool". In: (Jan. 2005).

[56] Guohui Wang and T. S. Eugene Ng. "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center". In: *Proceedings of the 29th Conference on Information Communications*. INFOCOM'10. San Diego, California, USA: IEEE Press, 2010, pp. 1163–1171. ISBN: 978-1-4244-5836-3. URL: http://dl.acm.org/citation.cfm?id=1833515.1833691.

[57] Matt Weinberger. *Why 'cloud computing' is called 'cloud computing'*. 2015. URL: https://www.businessinsider.com.au/why-do-we-call-it-the-cloud-2015-3.

[58] Johannes Wettinger et al. "Streamlining DevOps Automation for Cloud Applications Using TOSCA As Standardized Metamodel". In: *Future Gener. Comput. Syst.* 56.C (Mar. 2016), pp. 317–332. ISSN: 0167-739X. DOI: 10.1016/j.future.2015.07.017. URL: https://doi.org/10.1016/j.future.2015.07.017.

[59] Wikipedia. *DevOps*. URL: https://en.wikipedia.org/wiki/DevOps.

[60] Wikipedia. *Secure Shell*. URL: https://en.wikipedia.org/wiki/Secure_Shell.

[61] Wikipedia. *Traceroute*. URL: https://en.wikipedia.org/wiki/Traceroute.

[62] the free encyclopedia Wikipedia. *Virtual private cloud*. 2018. URL: https://en.wikipedia.org/wiki/Virtual_private_cloud.

[63] XebiaLabs. *Periodic table of DevOps tools*. URL: https://xebialabs.com/periodic-table-of-devops-tools/.