# Dynamic Service Chain Composition in Virtualised Environment

by

Wajdi Hajji

A Doctoral Thesis

Submitted in partial fulfilment
of the requirements for the award of

Doctor of Philosophy
of
Loughborough University

28th October 2018

# Abstract

Network Function Virtualisation (NFV) has contributed to improving the flexibility of network service provisioning and reducing the time to market of new services. NFV leverages the virtualisation technology to decouple the software implementation of network appliances from the physical devices on which they run. However, with the emergence of this paradigm, providing data centre applications with an adequate network performance becomes challenging. For instance, virtualised environments cause network congestion, decrease the throughput and hurt the end user experience. Moreover, applications usually communicate through multiple sequences of virtual network functions (VNFs), aka service chains, for policy enforcement and performance and security enhancement, which increases the management complexity at to the network level.

To address this problematic situation, existing studies have proposed high-level approaches of VNFs chaining and placement that improve service chain performance. They consider the VNFs as homogenous entities regardless of their specific characteristics. They have overlooked their distinct behaviour toward the traffic load and how their underpinning implementation can intervene in defining resource usage. Our research aims at filling this gap by finding out particular patterns on production and widely used VNFs. And proposing a categorisation that helps in reducing network latency at the chains.

Based on experimental evaluation, we have classified firewalls, NAT, IDS/IPS, Flow monitors into I/O- and CPU-bound functions. The former category is mainly sensitive to the throughput, in packets per second, while the performance of the latter is primarily affected by the network bandwidth, in bits per second. By doing so, we correlate the VNF category with the traversing traffic characteristics and this will dictate how the service chains would be composed. We propose a heuristic called *Natif*, for a VNF-Aware VNF insTantIation and traFfic distribution scheme, to reconcile the discrepancy in VNF requirements based on the category they belong to and to eventually reduce network latency. We have deployed *Natif* in an OpenStack-based environment and have compared it to a network-aware VNF composition approach. Our results show a decrease in latency by around 188% on average without sacrificing the throughput.

# Acknowledgements

I would like to thank Loughborough University and Liverpool John Moores University for supporting me during my PhD study.

I am grateful to my supervisors, Dr Posco Tso, Dr Iain Phillips, and Prof Qi Shi, for the help and guidance. I cannot forget how collaborative was the administrative staff in the Universities above, Tricia Waterson, Elizabeth Hoare, Bill Atherton, and Judith Poulton.

Thanks to my colleagues Akeel, Raul, Thiago for their kindness. My best wishes to them.

My gratitude goes also to my friend Slim, who has helped me in developing my career.

I would like to thank Ramzi, Riadh, Mohamed, my family, and my parents for their kind words and support.

# Publications

## Conference Proceedings

(i) Jeremy Singer, Herry Herry, Philip J Basford, **Wajdi Hajji**, Colin S Perkins, Fung Po Tso, Dimitrios Pezaros, Robert D. Mullins, Eiko Yoneki, Simon J. Cox, and Steven J. Johnston. "Next Generation Single Board Clusters". In:Network Operations and Management Symposium (NOMS), 2018 IEEE / IFIP (Accepted). IEEE.

(ii) **Wajdi Hajji**, Thiago Lopes Genez, Fung Po Tso, Lin Cui, and Iain Phillips. "Dynamic Network Function Chain Composition for Mitigating Network Latency". In:Computers and Communications (ISCC), 2018 IEEE Symposium on. IEEE. 2018.

(iii) Gabor Kecskemeti, **Wajdi Hajji**, and Fung Po Tso. "Modelling Low Power Compute Clusters for Cloud Simulation". In:Parallel, Distributed and Network based Processing (PDP), 2017 25th Euromicro International Conference on. IEEE. 2017, pp. 3945.

(iv) **Wajdi Hajji**, Fung Po Tso, Lin Cui, and Dimitrios P Pezaros. "Experimental evaluation of SDN-controlled, joint consolidation of policies and virtual machines". In:Computers and Communications (ISCC), 2017 IEEE Symposium on. IEEE. 2017, pp. 13381343.

## Journal Papers

(i) **Wajdi Hajji** and Fung Po Tso. "Understanding the performance of low power Raspberry Pi Cloud for big data". In Electronics 5.2 (2016), p. 29.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

A wide range of data centre applications are sensitive to latency, for example, high-frequency trading [8], high-performance computing, RAM-Cloud [9–11], and Online Data-Intensive (OLDI) [12]. As a result, a network performance degradation seen in these applications can risk user engagement and potential business revenue [13]. It is reported that every 100ms of latency cost Amazon 1% in sales [14].

In data centres, applications are typically communicating through a set network functions (NFs) [15], as demonstrated in Fig. 1.1, which are, according to IETF, *"functional building blocks within a network infrastructure"*, having *"well-defined external interfaces and a well-defined functional behavior. In practical terms, a Network Function is today often a node or physical appliance"*. NFs such as firewall, proxies and WAN optimisers have become a critical part of today's data centres. They are essential in guaranteeing security and improving performance. A survey on Enterprise middlebox (a synonym of network appliance) in 2011 has revealed that the number of middleboxes was on par with the number of layer-3 routers [16]. Also, these NFs are rarely being used in isolation, they are commonly deployed in chains and are typically interposed between the communicating hosts [17, 18]. For instance, to fulfil changing policy requirements, multiple NFs need to be dynamically chained in an ordered sequence for the delivery of end-to-end services. Nevertheless, being in composition, NFs can provoke a mutual interference as they can change the volume of the processed traffic [2], which can cause resource, such as network and CPU, bottlenecks in the chain. As a result, network latency starts to build up and degrade the performance of latency-sensitive applications [19].

NFV has facilitated the transformation of NFs to software applications running

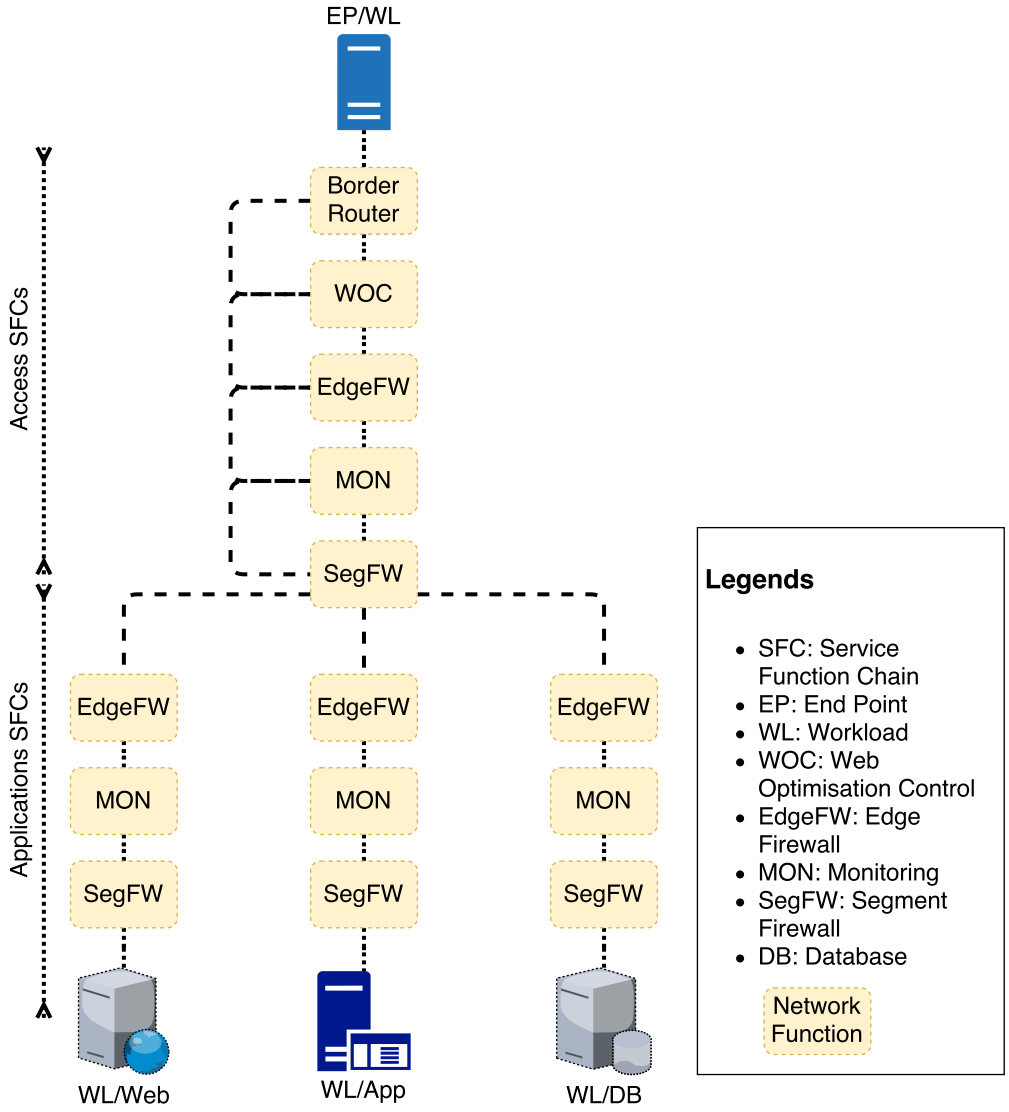



Figure 1.1: Service function chains in data centre [1]

on vendor-independent commodity servers in virtual forms (Virtual Machines or Containers), which provides flexibility in offering dynamic services fulfilling rapidly changing user demands [3].

Whereas, in addition to the performance problem caused by the NF chaining, virtualisation technology leveraged by NFV is considered one of the main reasons behind network congestion at computational and network appliances. Fig. 1.2 shows four latency traps in a simple network comprising communicating hosts (Virtual Machines (VMs)), a virtual network function, and a forwarding device (switch). At point (1) the hypervisor on the physical host schedules the transmission of the packets to the server VMs. This has turned out to be a real problem in Amazon EC2, and it causes large tail latency between EC2 instances [20]. At (2), the physical host network stack has to fulfil I/O requests for multiple VMs, which is another source of excessive latency [21]. Similarly, at (3) both traps described

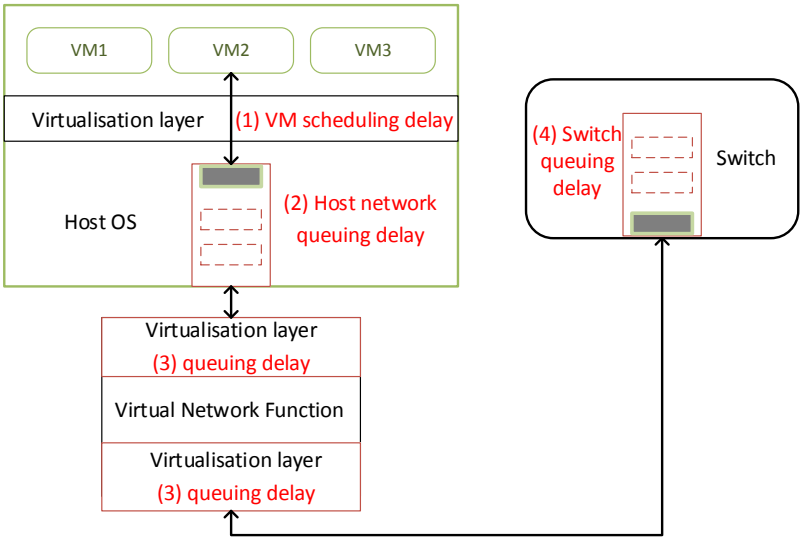



Figure 1.2: Latency traps

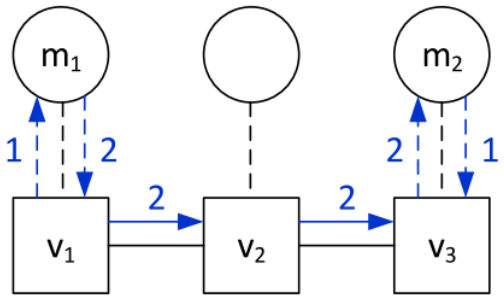
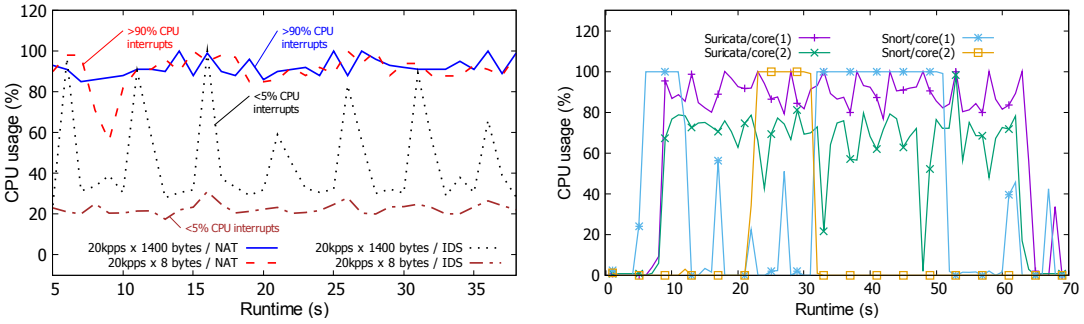



Figure 1.3: Traffic changing effects of network functions. NF $m_1$ doubles the traffic volume while $m_2$ cuts it in half [2]

in (1) and (2) coexist since the network function is running in a virtual host such as VM or LXC (Linux Containers). Lastly, (4) is where traffic can experience queuing at switch. Also, VMs on the same physical host can contend for limited bandwidth available to that host ignoring the specific requirements of each other [21].

In service composition, current studies tackle the latency problem in the service chains by proposing high-level strategies managing the VNF instances without considering their characteristics such as the ways they are handling the traffic load. For example, VNF placement optimisation [22], VNF chaining [23, 24] and VNF parallelisation [25] improve latency by shortening end-to-end paths. However, they see VNFs as black boxes and neglect their internal packet processing characteristics.

VNFs are software applications running inside a virtualised environment. While software performance is bounded by either I/O or CPU or both of them. In gen-




(a) CPU use at pfSense NAT and Snort IDS (b) Multi-threaded Suricata IDS vs. single-threaded implementation in Snort

Figure 1.4: VNF performance bottlenecks

eral, I/O bound application is when the completion time primarily depends on the waiting time for input/output operations to complete. CPU-bound application is when the completion time principally depends on the speed of central processor or widely known as CPU.

We argue that VNFs are no exception. To demonstrate this, we have evaluated the resources usage at three VNFs, pfSense Network Address Translator (NAT), Snort Intrusion Detection System (IDS), and Suricata IDS. Each one of them is deployed on a Virtual Machine (VM) with 1Gbps vNIC, 1vCPU, and 1GB RAM. Fig. 1.4a depicts a distinct CPU usage, and CPU interrupts activities at pfSense NAT and Snort IDS despite being allocated similar resources and handling the same network traffic. This is because pfSense NAT is insensitive to the packet payload since it only handles the packet header. At each packet arrival, it calls its subroutine running in the user space which results in high CPU interrupts (interrupt-driven I/O), hence I/O-bound. We have found that the effective computation of pfSense NAT is only around 2% of the total CPU usage. In comparison, the Snort IDS, using community rules[1], buffers and inspects the data carried out in the packet payloads, a CPU-intensive task, hence CPU-bound. Interestingly, our experiments have also revealed that Suricata IDS, a multi-threaded application, efficiently uses the CPU resources while Snort IDS, a single-threaded implementation, only uses one core at a time, as shown in Fig. 1.4b.

To sum up, we have shown three main challenges in the current data centre environments, namely, virtualisation overhead effect on the application network performance, VNF chaining, and lack of exploiting the VNF proprieties in the existing VNF management schemes. In next section, we, therefore, present our research hypothesis and aims in investigating and addressing the problems above.

[1]https://www.snort.org/downloads/#rule-downloads

## 1.2 Research hypothesis and objectives

We aim to mitigate the network latency at the service chain, ordered or partially ordered sequence of general network functions (NFs). This will have a significant impact of the network performance of latency-sensitive applications. For this purpose, we aim to experimentally understand to what extent the virtualised environment can degrade NFs performance. On the other hand, unlike the existing NF management approaches that neglect the NF characteristics [22][23][24][25], we are looking for studying different types of NFs to find out a particular pattern relating them or an interesting characterisation that can be exploited in the chain composition and traffic steering. To do so, we leverage Software Defined Networks, Service Function Chaining, and NFV paradigms in designing testbed and for preliminary assessment and solution implementation. Lastly, we demonstrate how our research can be an extension of the state-of-the-art through showing the advantage of its outcomes comparing with concurrent works.

We define the following aim and the corresponding objectives.

- Aim: Provide a novel VNFs characterisation that is able, where exploited, to improve the service chain network performance.

  In order to achieve the above aim. We proceed with the following steps.

  - Study a set of open source and production VNFs, namely, pfSense NAT, pfSense firewall, Snort IDS, Suricata IDS, and Open vSwitch as a traffic monitoring VNF.
  - Exploit the experimental study of the above VNFs in a mathematical formulation and modelling of the VNF instantiation and traffic distribution problem.
  - Propose a heuristic for VNF-aware service chain composition.
  - Implement the proposed solution in an OpenStack based testbed to demonstrate its efficiency compared to a network-aware approach.

## 1.3 Original contributions

Contrary to the existing approaches that work on VNF composition, parallelisation and placement. Ours consider the specificity of each VNF. The classification we propose is inspired from the fact that any software application can belong to one or both of these categories I/O- and CPU-bound. This observation is still applicable on VNFs and we aim to exploit it in our chain composition. In this section, we

show how the research questions we pose lead to the main contribution of our work.

- RQ1: How can we understand the virtualisation affect on the network functions performance?
  - Study the network performance of a production VNF, virtual firewall, and compare its performance with its counterpart in bare-metal deployment.
- RQ2: Is it possible to classify the network functions based on their performance bottlenecks?
  - We use the categorisation that applies on any software application, which is I/O- versus CPU-bound. We also experimentally study open source and production virtual network functions regarding 1) the impact of the software implementation on their resource utilisation, 2) their sensitivity to traffic load, and 3) how their order can impact the performance of the service chain.
- RQ3: How does the experimental analysis of the network functions help in improving the service chains network performance?
  - We exploit the experimental knowledge to mathematically model the virtual network function instantiation and traffic distribution problem. Since the problem has been proved NP-hard, we design a heuristic that correlates the network function and the traffic characteristics based on the VNFs categorisation. We validate our approach through conducting testbed experiments in production environment.
- RQ4: How does our research advance the state-of-the-art?
  - Through the literature review, we have identified occasions to improvements in the existing service chain composition schemes. Current studies in service composition neglect the particularities of network functions, they instead treat them as black boxes. We highlight how our research differs from the existing studies. We demonstrate how exploiting the VNF characteristics improve the overall performance of service chains.

## 1.4 Thesis overview

This thesis is organised as follows.

**Chapter 1** (this chapter) illustrates the main motivations of our research work. It highlights the significance of latency in today's data centre networks and its particular impact on latency-sensitive applications. Also, it shows the ubiquitousness of service chains and how the sequential processing of packets at virtual NFs can degrade the network performance of data centre applications. Moreover, it demonstrates the lack of considering the performance bottlenecks at the NFs in the existing approaches. Afterwards, the chapter describes our aims and objectives, and our contribution to the research, then it gives an overview of the thesis structure (this section) and ends up with listing our publications.

**Chapter 2** defines the background of our research such as the leveraged paradigms, namely, virtualisation technology, SDN, NFV, and SFC.

**Chapter 3** describes the literature review, e.g., works achieved in the context of data centre networks, attempts made to reduce network latency, and more related to our research, network function composition, chaining, design, and management studies.

**Chapter 4** shows the methods we have adopted in our research. For example, benchmarking VNFs and setting up an OpenStack based experiment to run multiple service chains. Also, the chapter illustrates other methods such as network simulation used to evaluate joint policy management and VM placement approach.

**Chapter 5** shows how the virtualisation can affect the performance of open source and production VNFs. It also describes the conducted experiments regarding the VNF characterisation, and it ends up with demonstrating how the exploited knowledge on the VNFs can improve the network performance of a sample of service chains.

**Chapter 6** describes the main contribution of our research work which is proposing a dynamic VNF composition that relates the traffic characteristics with the VNF performance bottleneck. We prove that the VNF instantiation and traffic distribution problem is an NP-hard problem which means that it does not have an optimal solution. As so, we formulate and model the problem, we propose our heuristic called *Natif*, and we describe the algorithms behind.

**Chapter 7** describes the experimental setup for the solution evaluation, we show how *Natif* mitigates the latency without sacrificing the network throughput. We also compare it with a well-known network-aware approach for NF orchestration. For instance, we evaluate the algorithms of the two methods as well as the resources utilisation of service chains composed and managed by each one of them.

**Chapter 8** concludes the thesis and shows the proposed approach's limitations and new research directions. It also demonstrates how the acquired knowledge as well as the experimental evaluation can be usable and applicable in the context of edge computing. It illustrates practical ideas regarding such applicability.

# Chapter 2

# Background and Key Concepts

In this chapter, we describe the main concepts and paradigms that have been explored and leveraged. We explain the virtualisation technology as a key enabler of the Network Function Virtualization (NFV) and the broad adoption of virtual network appliances or functions (VNFs) in data centres. We also present the Software Defined Networks (SDN) as a concept calling for the centralisation of the network management and the dissociation of the control plane and the data plane. In the end, we describe the Service Function Chaining, and we highlight how it has enabled the application of our proposed approach in reducing the network latency.

## 2.1 Virtualisation

Virtualisation has been widely leveraged to fulfil the growing user needs associated with the advent of smart-phones and cloud computing service models. A wide range of resources has been virtualised such as computation process, memory, storage, and network [26]. Virtual Machines are the virtual form of the traditional computer units, and they are nowadays ubiquitous in data centres and workplaces. Behind the concept of VM, related components have been consequently virtualised such as NIC (Network Interface Cards) and CPU resources. Open vSwitch, known as OVS, is among the first initiatives to implement virtual network switches. Moreover, the network has been virtualised like VLAN (virtual LAN) and VXLANs (virtual extensible LANs).

Virtualisation has been introduced for numerous reasons. 1) Sharing, which means that the same cable/link can be utilised by two different networks, the same processor or the same hard disk can be shared between more than a VM. As a result, sharing reduces the cost of hardware. 2) Isolation is a critical feature when multiple applications are running on the same server. Virtualisation ensures that

each resource can only be accessible by the right application or the right person. 3) Aggregation, contrary to the resources segregation, is achieved by grouping inexpensive resources to make up reliable resource, e.g., storage or memory. 4) flexibility, for, e.g., dynamic resource allocation, resource management and server provisioning and deployment. Also, VMs can be instantiated, deleted, cloned or migrated within minutes, which increases the productivity and reduces the customer's waiting time [27]. In our research, virtualisation is utilised in virtual NFs and running data centre applications on testbed environments.

## 2.2 Software Defined Networks

SDN is an umbrella term that includes several technologies to facilitate the network management and improve the network flexibility to respond to changing business requirements. For example, network administrators can change network policies for specific network traffic from the controller without directly re-configuring the network devices, e.g., routers or switches. So, OpenFlow network protocol has been proposed to ensure the communication between the controller and the forwarding devices. Also, a network virtualisation approach is an SDN application that allows flexible network provisioning and dynamic flow scheduling [28] due to resources virtualisation. SDN is mainly bringing four innovations [27]:

- **Separation of control and data plane:** The traditional way to perform traffic steering is to configure the forwarding devices individually. This is apparently tedious especially if the traffic characteristics (e.g., source and destination IPs) can change over the time or there are many points on the forwarding path. Hence, SDN comes to resolve the problem above by separating the control plane, where forwarding decisions and policy are set, from the data plane, where packets are forwarded. For example, implementations of the control plane such as Ryu, NOX, Floodlight allow applying customised treatment over the packets and prepare the forwarding tables of the connected OpenFlow switches.

- **Centralisation of the control plane:** Distributed methods in forwarding packets were widely adopted before the emergence of SDN. For example, through network protocols like OSPF, IS-IS, and BGP, routers exchange information about their neighbours and available nodes in a way that the forwarding decision would be shared between the existing routers on the network. This achieves more reliable message transmission since the system can tolerate the failure of some routers. However, it presents performance issues comparing to the centralised approach in a sense that the routing algorithm

convergence depends on the routers' status and consequently the information takes a while to be exchanged between them. Whereas in the centralisation approach, the controller probes the network topology, and since it would have visibility on the network, it can, therefore, decide which path should be used to forward the packers. Centralisation has scalability issues such as the impact of increasing workload with growing traffic on the controller's performance, but this is addressed by setting up a controller cluster. For instance, the set of controllers can share the workload, or standby controllers can be launched in case of failure of the central controller.

- **Programmable control plane:** The control plane can be programmed according to the defined objectives. This has not been feasible in the distributed method. Network administrators can develop and deploy applications on the control plane, which offers high flexibility as well as innovation opportunities for network developers.

- **Standarised APIs:** Control plane needs standardised APIs to facilitate the development and deployment of network applications which are ensuring the traffic management. For example, OpenFlow as a Southbound API; interacting with the forwarding devices. Floodlight and OpenDaylight as Northbound API as it enables the integration with the developed applications, and lastly the East-West API that enables the communication between different controllers. We show in Fig. 2.1 such layers in SDN architecture.

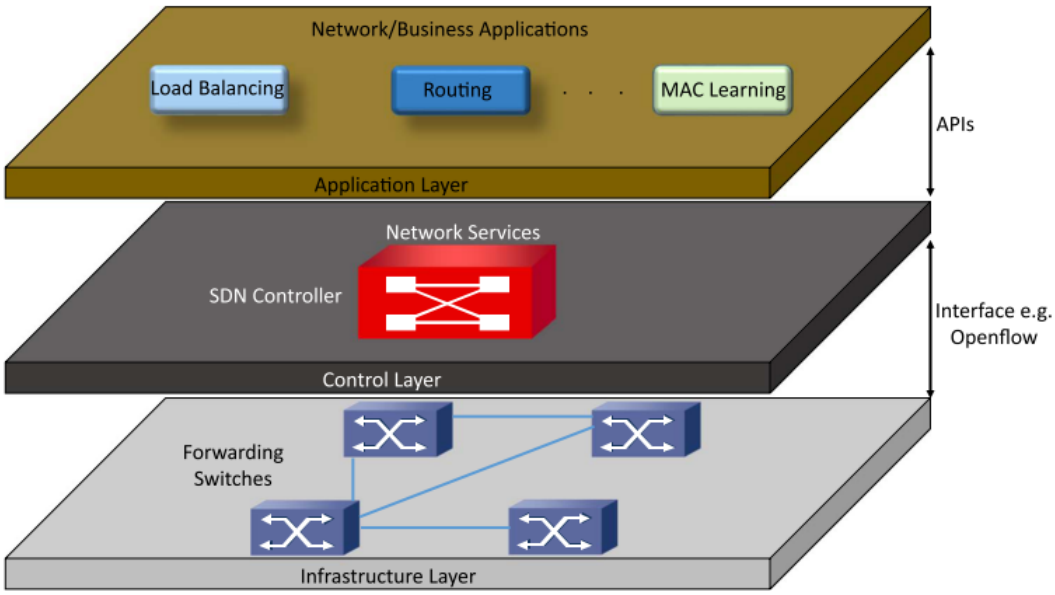



Figure 2.1: Logical layers in SDN [3]

## 2.3 Network Function Virtualisation

Network infrastructure should respond to the growing demands and the performance challenges caused by the increasing workloads, a result of the proliferation of cloud and mobile applications. However, the network functions (NFs) and the computational nodes responsible for handling the incoming requests have been deployed using physical proprietary equipment. Also, NFs usually belong to network chains (sequence of NFs) and are configured within the chain according to a specific order to correctly ensure network services. Because of these constraints, deployment and validation could take weeks and months and require significant human involvement to maintain and manage the network asset [29].

Therefore, Telecommunication Service Provides (TSPs) have been looking for inventing ways and proposing new approaches to reduce their operating expenses (OPEX) and capital expenses (CAPEX). The solution has been to adopt the Network Function Virtualisation (NFV) leveraging the virtualisation technology to break the dependency on proprietary hardware, improve the network management and service agility, and reduce the time-to-market of new services. NFV is defined as an initiative to decouple the physical network equipment from the functions that are running on. NFV achieves that by virtualising the NF to mimic the VM concept in a way, they can run on commodity hardware. For instance, a firewall becomes a piece of software that can be installed on a VM or a container and afterwards deployed on commodity servers, or it is possible to aggregate multiple NFs and spin up them on the same server. As a consequence, managing NFs becomes as simple as managing VMs through hypervisors which means offering automation capabilities such as monitoring, deletion, live migration, instantiation and configuration.

### 2.3.1 NFV considerations

NFV to be convincing to the industry and academia, it should satisfy the following requirements. The first two requirements are exploited in our research while the last ones have been highlighted for informational purpose.

- **Network architecture and performance:** NFV should provide a reliable alternative to the physical deployment of the NFs. Otherwise, its contribution would be questionable. For instance, NFV should address possible network performance issues that can be a consequence of the virtualisation technology.

- **Network scalability and automation:** NFV should scale with the growing number of subscribers and so the increasing computation needs. Auto-

mation can play a critical role to overcome the scalability challenge, e.g., setting up controllers for resources management and usage optimisation (enhancing the throughput or reducing the latency) [3].

- **Security and high availability requirements:** Diverse NFs can be deployed on the same commodity server, but they can belong to different tenants or subscribers. So, there is a need to isolate them from each other and resources allocation should be reliable and consistent to avoid performance interference between the NFs. Also, NF deployment plan should consider the availability requirements of each NF.

- **Support for heterogeneity:** Since NFV is a new approach and not yet totally adopted in production environments, its introduction should be progressive and should take into consideration the legacy support. As a consequence, there should be an orchestration layer that can manage both virtual and physical infrastructure. Another aspect of supporting heterogeneity consists of breaking the dependence on proprietary hardware so that NFs can be deployable on servers from different vendors.

### 2.3.2 NFV architecture

NFV architecture is composed of three main components as shown in Fig. 2.2. 1) Network Function Virtualisation Infrastructure (NFVI), which represents the set of software and hardware needed to set up the environment on which VNFs run. 2) VNFs, which are the deployed virtual network appliances. A VNF can be deployed on VM. Moreover, 3) NFV Management and Orchestration (NFV MANO), which provides tools for monitoring and provisioning of VNFs.

## 2.4 Service Function Chaining

Network service chaining, also known as service function chaining (SFC) is a capability that uses software-defined networking (SDN) capabilities to create a service chain of connected network services (such as L4-7 like firewalls, network address translation [NAT], intrusion protection) and connects them in a virtual chain. This capability can be used by network operators to set up suites or catalogues of connected services that enable the use of a single network connection for many services, with different characteristics [**?**]. However, in this thesis, we refer to the way VNF are instantiated and how traffic is steered within the chain by network service composition.

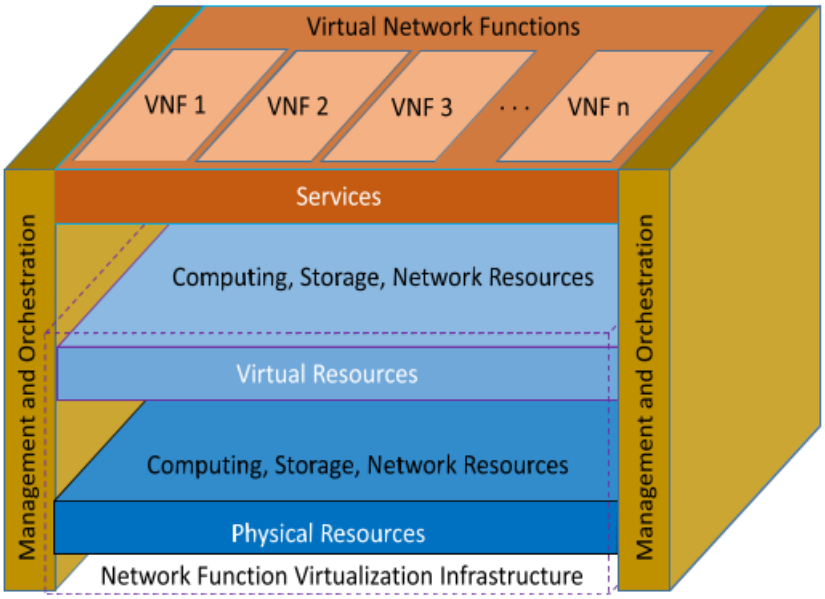



Figure 2.2: NFV architecture [**?**]

### 2.4.1 SFC definitions

NFV has reduced the time-to-market of new network services by virtualising the hardware-based network appliances or functions and thus accelerating their deployment on commodity servers. However, a proper interconnection of the network services is required to ensure the correct implementation of network policies. The mechanism allowing various virtual functions to be connected for a complete end-to-end service is called Service Function Chaining (SFC).

We list below essential definitions related to SFC.

- Classification: Locally instantiated matching of traffic flows against policy for subsequent application of the required set of network service functions. The policy may be a customer, network, or service specific.
- Network Overlay: A logical network built, via virtual links or packet encapsulation, over an existing network (the underlay).
- Network Service: An offering provided by an operator that is delivered using one or more service functions. This may also be referred to as a composite service. The term "service" is used to denote a "network service" in the context of this document.
- Service Function (SF): it is a synonym to virtual functions, a function that is responsible for specific treatment of received packets. A service function can act at various layers of a protocol stack (e.g., at the network layer or other OSI layers). As a logical component, a service function can be realised

as a virtual element or be embedded in a physical network element. One or more service functions can be embedded in the same network element. Multiple occurrences of the service function can exist in the same administrative domain. A non-exhaustive list of service functions includes firewalls, WAN and application acceleration, Deep Packet Inspection (DPI), server load balancers, NAT44 [RFC3022], NAT64 [RFC6146], HTTP header enrichment functions, and TCP optimisers. The generic term "L4-L7 services" is often used to describe many service functions.

- Service Overlay: An overlay network created for forwarding data to essential service functions.

- Service Function Chain: it defines an ordered or partially ordered set of general service functions (SFs) and ordering constraints that must be applied to packets, frames, and flows selected as a result of classification. An example of an abstract service function is a firewall. The implied order may not be a linear progression as the architecture allows for SFCs that copy to more than one branch, and also allows for cases where there is flexibility in the order in which service functions need to be applied. The term "service chain" is often used as shorthand for "service function chain" [30].

### 2.4.2 SFC architecture

The SFC architecture proposed by IETF suggests encapsulating the packets with information describing the service path of a service function chain. This is achieved by adding Network Service Header (NSH) to the packets at the service classifier (SC) located at the data plane, e.g., as shown in Fig. 2.3. The SC determines which packets need treatment and what the service path that should follow. The SF Forwarder (SFF) considers the NSH field in the packet for the traffic steering. An SFC Aware SF can update the NSH header. In particular, the first node in the service chain adds NSH field while the last node removes it [31].

In case the SFC SF is NHS-unaware (e.g., legacy service functions), an SFC-Proxy can be used to ensure the NSH packet encapsulation between the SF and the SFF (Fig. 2.4).

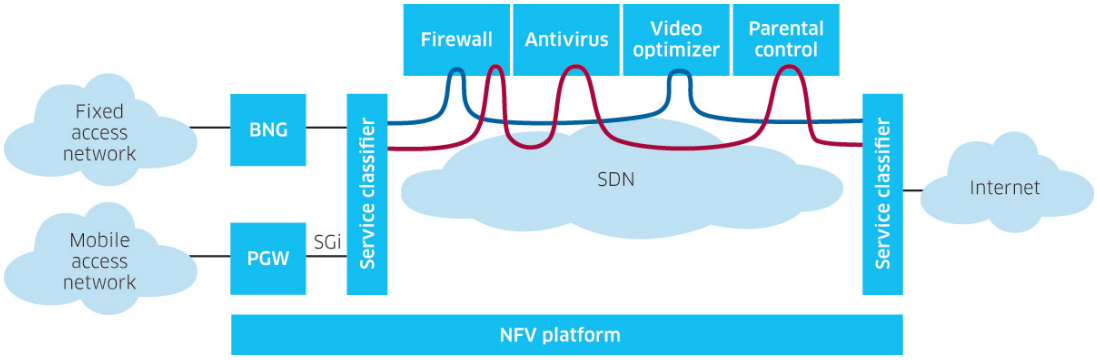



Figure 2.3: Service classifier [4]

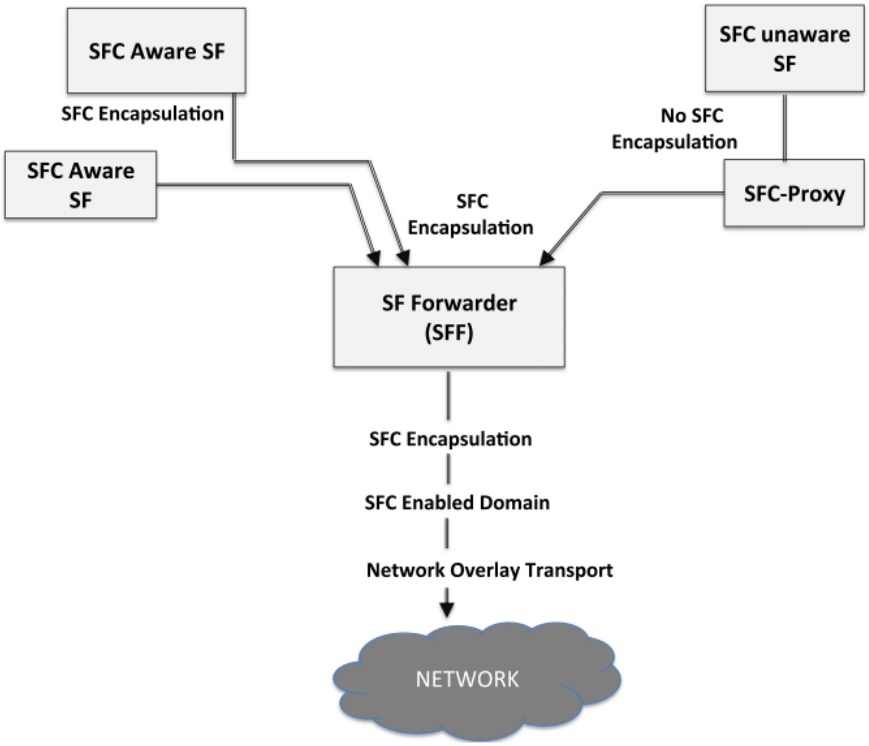



Figure 2.4: SFC architecture [5]

### 2.4.3 A catalogue of middleboxes

The table 2.1 introduces some examples of middleboxes to give an overview on the types of middleboxes used in data centre networks.

Table 2.1: Examples of middleboxes [7]

| **Middlebox** | **Description** |
|---|---|
| NAT | Network Address Translator. A function, often built into a router, that dynamically assigns a globally unique address to a host that doesn't have one, without that host's knowledge. |
| NAT–PT | NAT with Protocol Translator. A function, normally built into a router, that performs NAT between an IPv6 host and an IPv4 network, additionally translating the entire IP header between IPv6 and IPv4 formats. |
| SOCKS gateway | It is a stateful mechanism for authenticated firewall traversal, in which the client host must communicate first with the SOCKS server in the firewall before it is able to traverse the firewall |
| IP Tunnel Endpoints | Tunnel endpoints, including virtual private network endpoints, use basic IP services to set up tunnels with their peer tunnel endpoints which might be anywhere on the Internet. Tunnels create entirely new "virtual" networks and network interfaces based on the Internet infrastructure, and thereby open up some new services. Tunnel endpoints base their forwarding decisions at least partly on their policies, and only partly if at all on information visible to surrounding routers. |
| Packet classifiers, markers and schedulers | Packet classifiers classify packets flowing through them according to policy and either select them for special treatment or mark them, in particular for differentiated services. They may alter the sequence of packet flow through subsequent hops, since they control the behaviour of traffic conditioners. |
| TCP performance enhancing proxies | "TCP spoofer" is often used as a term for middleboxes that modify the timing or action of the TCP protocol in flight to enhance performance. |

| | |
|---|---|
| Load balancers that divert/munge packets | There is a variety of techniques that divert packets from their intended IP destination or make that destination ambiguous. The motivation is typical to balance the load across servers, or even to split applications across servers by IP routing based on the destination port number. |
| IP Firewalls | The simplest form of firewall is a router that screens and rejects packets based purely on fields in the IP and Transport headers (e.g., disallow incoming traffic to certain port numbers, disallow any traffic to certain subnets, etc.) |
| Application Firewalls | Applicationlevel firewalls act as a protocol endpoint and relay (e.g., an SMTP client/server or a Web proxy agent). |
| Application-level gateways | These come in many shapes and forms. NATs require ALGs for certain addressdependent protocols such as FTP; these do not change the semantics of the application protocol but carry out mechanical substitution of fields. At the other end of the scale, still using FTP as an example, gateways have been constructed between FTP and other file transfer protocols such as the OSI and DECnet (R) equivalents. In any case, such gateways need to maintain state for the sessions they are handling, and if this state is lost, the session will normally break irrevocably. |
| Gatekeepers/ session control boxes | Particularly with the rise of IP Telephony, the need to create and manage sessions other than TCP connections has arisen. In a multimedia environment that has to deal with name lookup, authentication, authorisation, accounting, firewall traversal, and sometimes media conversion, the establishment and control of a session by a thirdparty box seems to be the inevitable solution. |
| Transcoders | Transcoders are boxes performing some onthefly conversion of application-level data. |
| Proxies | An intermediary program which acts as both a server and a client to make requests on behalf of other clients. |

## 2.5 Summary

This chapter has illustrated the background and fundamental concepts used in our research. We have illustrated how the virtualisation technology is a key enabler for NFV, SFC, and SDN paradigms. It has also presented the essential definitions in the SFC context and illustrated examples of network functions. In the next chapters, we show use cases of SDN, SFC, and NFV applications. For instance, development of a joint consolidation of policies and virtual machines, VNF deployment and characterisation VNFs, and service chain composition using SFC principles to reduce the network latency.

# Chapter 3

# Literature Review

Many cloud applications in multi-tenant data centres are distributed in nature and require guaranteed latency and bandwidth to afford an acceptable user experience [12]. Providing such guarantees has been challenging for several considerations; queues the most disturbing factor of network performance is an additive end-to-end property [12], and it is developing in many points in data centres, e.g. Virtual Machines, Switches and Virtual Appliances. Moreover, low latency and high throughput are two contradictory goals and prioritising one of them leads to the regression of other [11, 13]. Recent studies have proposed approaches that reconcile between these two ends, whereas after the emergence of NFV technology and with the broad adoption of network appliances in data centres, those approaches have still been limited to improve network performance at the level of the forwarding devices with no particular consideration of the network functions performance. The existing studies are worth to explore as they present relevant techniques to our current research. Thus, one of the aims of the chapter is to give an overview of these works, what challenges they have been addressing and how they have overcome.

We also cover the renewed focus on network latency metric and the efforts that have been made to mitigate it in data centre environments. We also highlight the noticeable degradation of network performance caused by the virtualisation technology and how such an overhead has been addressed.

Lastly, we describe the studies that have been looking at improving the network chains performance regarding the end-to-end delay and the throughput. We illustrate the different approaches to tackling the problem, such as NF chaining and placement, NF parallelisation, and NF resources requirement provisioning and prediction.

## 3.1 Data centre networks

Several works have proposed techniques to assess the overall performance, understand network topology, and detect network degradation within the data centre environment. For instance, Everflow [32], ECHO [33], Pingmesh [34] and SNAP [35] have presented ways for debugging faults in data centre networks, measuring and analysing latency of servers in large multi-tenant data centres and providing network administrators with performance monitoring interfaces. They have contributed to mitigating servers' downtime, improving the quality of services, helping developers and operators to identify and diagnose network performance problems in the reasonable and acceptable time frame.

Others have focused on improving and renewing the architecture of data centres following the emergence of NFV and SDN. For instance, work in [36] has pointed out the usefulness of the SDN approach to facilitate and enable network management and programmability and allow more control of the underlying infrastructure. Authors of [37] have proposed Open Network Operating System (ONOS) prototype for global network view on the network topology and state. Authors in [38] have proven how VM placement can be efficient and improving the overall data centre performance if network traffic and topology information have been taken into account

Researchers have also drawn attention to the network policy issues in data centres and have proposed mechanisms to tackle them. Work presented in [39] has developed a high-level Policy Graph Abstraction (PGA) to describe network policies clearly and independently, and by using graph theory, it has been proven possible resolving conflicts between policies in a compelling way. The same reconciliation objective has been considered in [40] for network policies, and [41] for routing rules and [42] for network updates.

In a virtualised environment, Virtual Machines placement has been thought of as one of data centres management knobs that could improve general network performance. Net-Cohort [43] has aimed at reducing the bisection bandwidth by proposing VM ensembles detection and placement based on information collected about VM network interactions. Cloud Mirror [44] has used as well the VM placement technique and what called Tenant Application Graph (TAG) to set guarantees for network bandwidth. AppAware [45] also has come up with an application-aware VM migration algorithm that, in simulation, has led to a vital network traffic reduction.

There have also been application-aware approaches to improve the application and network performance. For instance, [46] and [47] have implemented an application-aware data plane processing and packet forwarding mechanisms in

the light of SDN paradigm. Also, work in [48] has demonstrated how YouTube application performance can be enhanced if application recognition is leveraged while serving web requests. Table 3.1 summarises the above works and tells if the described technique can be applied in the perimeter of service chains.

Table 3.1: Summary of efforts made in the area of data centre networks

| **Ref.** | **Context** | **Affected object** | **Applicable on service chains?** | **Summary of the main technique** |
|---|---|---|---|---|
| ECHO [33] | Network workload modelling | Data centre applications | Yes | Markov Chain model trained on real traces to capture temporal and spatial network patterns |
| Pingmesh [34] | Network latency measurement and analysis | Latency and packet loss | Yes | Create a latency graph based on the probes of servers between each other, then data are visualised and analysed in Data Storage and Analysis. |
| SNAP [35] | Performance problem detection | Data centre applications | Yes | Collect and correlate TCP statistics and socket-level logs across shared resources and connection to locate performance problem |
| [36] | SDN | Data centres | Yes | Survey |
| [37] | SDN | Data centre network and applications | Yes | Collect data from network devices (ONOS instance) to construct a global network view which in turn updated by applications |
| [39] | Network policies modelling | Network appliances | Yes | PGA (Policy Graph Abstraction) specifies the packet processing behaviour of service function, identifies overlapping endpoint membership, and adds composition constraints to avoid policy violation |
| [40] | Network control conflict detection | SDN controllers | Yes | Model the controller function as a deterministic finite-state transducer |

| | | | | |
|---|---|---|---|---|
| Fibbing [41] | Routing and traffic steering | Network traffic | Yes | Inject fake network nodes to the existing topology to change the routing behaviour |
| Net-Cohort [43] | Dependency analysis | Application throughput, bi-section bandwidth | Yes | Monitor traffic exchanged between VMs in order to use it for VM placement |
| CloudMirror [44] | Application performance | Bandwidth and High-Availability | No | Propose TAG (Tenant Application Graph) to allow applications to precise their network requirements which defines the workload distribution scheme to guarantee bandwidth and high availability requirements. |
| AppAware [45] | Application-aware VM placement | Network traffic | No | Model the VM placement problem based on the dependency of applications running on VMs, the underlying topology, and the capacity of hosting physical servers |
| [46] | SDN | Data plane | Yes | Augment the programmable Open vSwitch with stateless app processing capability (app table), similar to the typical OpenFlow flow table. |
| Atlas [47] | Application classification | Data centre applications | Yes | Implement a Machine Learning consuming traffic data from OpenFlow switch to recognise applications |
| [48] | Quality of Experience (QoE) | Data centre applications | Yes | Improve the QoE for a YouTube user by using application signature to define a custom network behaviour particularly regarding path selection |

## 3.2 Network latency

Researching in reducing network latency has gained momentum because of the application strict requirements and the significance of latency metric to the user experience satisfaction. We aim in this section to give an overview on the techniques addressing network latency in data centre environment.

Work in [21] has introduced a novel host-centric solution to mitigate latency in a virtualised environment, it tackles three latency traps induced by VM scheduling delay, host network queuing delay, and Switch queuing delay. It relies on applying the Shortest Remaining Time First (SRTF) scheduling policy at the level of end-host to control traffic to reduce latency in respect to the throughput requirement.

Fastpass [49] has proposed a data centre network architecture that in a centralised arbiter schedules all network traffic at a fine-grained level so that packets queuing would be reduced. Silo [12], QJUMP [13] and HULL [11] have aimed at improving the network performance of two types of applications; latency-sensitive and throughput-intensive. The idea in QJUMP consists of recognising these applications and set up a sort of packet-level prioritisation at network Switches. Silo relies on VM placement and end-host packet limiters to lower congestion and also to improve throughput. However, another work like [50] shows how the software rate limiters can increase the latency by order of magnitude in cloud networks. CONGA [51] and Presto [52] have proposed mechanisms for load balancing that aims at reducing congestion and so mitigating latency. The above works have been summarised in Table 3.2.

Table 3.2: Summary of some works tackling latency problem

| **Ref.** | **Context** | **Location of latency** | **Applicable on service chains?** | **Summary of the main technique** |
|---|---|---|---|---|
| [21] | GuaranteedLatency | VMs, hosts, and switches | Yes | Use Shortest remaining time first to schedule traffic at the end-hosts |
| Fastpass [49] | Data centre network architecture | Switches | Yes | Schedule packets transmission in specific timeslot and network path |
| Silo [12] | Guaranteed Latency | Switches | Yes | VM placement + rate limiter at end-hosts |
| QJUMP [13] | Guaranteed Latency | Switches | Yes | Packets are prioritised at the switches according to their application sensitivity to latency |
| HULL [11] | Data centre network architecture | Switches | Yes | Define a bandwidth headroom that triggers a phantom queues signal which invokes the application of DCTCP algorithm used for congestion control |
| CONGA [51] | Load balancing | Switches | Yes | Split TCP flows into flowlets and use switch feedback to balance traffic to reduce congestion |
| Presto [52] | Load balancing | Switches | Yes | Fine-grained flowcells (defined portion of flow) load balancing at virtual switches |

## 3.3 Service chains

SFC and NFV have facilitated the introduction of network services implementing network policies for security and performance purposes. Also, service chains have become ubiquitous in data centre environment which explains the significant growth in a number of studies on this topic.

In this context, Stratos [53] has investigated a network-aware orchestration layer for middle-boxes. For this purpose, it has come up with three mechanisms; 1) elastic scaling, to determine how many middle-boxes needed to be deployed. 2) middle-boxes Rack-aware placement algorithm that considers the bandwidth availability on network links, and 3) network-aware flow distribution that is triggered once a scaling decision is taken to improve the network utilisation. Sync [54] has proposed a synergistic middle-boxes and VMs placement that reduces the end-to-end delay and the communication cost.

Other studies issued from the telecom industry have proposed architectures and models to manage the virtual appliance instances. VNF-P [55] has presented a model for efficient placement of virtualised network functions, [56] and [57] have offered ways for efficient dynamic placement of chains of virtual network functions, other general works like [58] and [59] have aimed at improving the performance of middle-boxes and facilitating its management. Even though the context and the environment to which these studies belong are partly different from those of data centres, they are worthy to be investigated for possible reuse.

The number of studied on the optimal and efficient middle-boxes placement has increased significantly in the recent years. For instance, the authors in [22] have proposed a middle-boxes placement algorithm to decrease the number of rules implemented on SDN switchs responsible for traffic steering. Work in [60] has presented an incremental solution that seeks to reduce the utilisation of the links and the available CPU cores. Quokka [61] schedules the deployment of middle-boxes according to the changing traffic in a bid to reduce the transmission latency. Work in [62] has introduced a pre-planned placement scheme that considers the tenants' requirement for network bandwidth to reduce the migration cost due to dynamic middle-boxes placement. [63] and [64] have also proposed heuristic middle-boxes placement algorithms to reduce the end-to-end delay and the bandwidth consumption. Moreover, [65] has described and formulated the problem of placing the network functions with minimal resource consumption. Other approaches proposed in [23, 25] focus on the placement problem to reduce the end-to-end delay at the chains without considering the VNF performance. Instead, they model and manage the VNFs as identical entities.

SIMPLE [66] uses both an online and offline formulation to keep limiting the

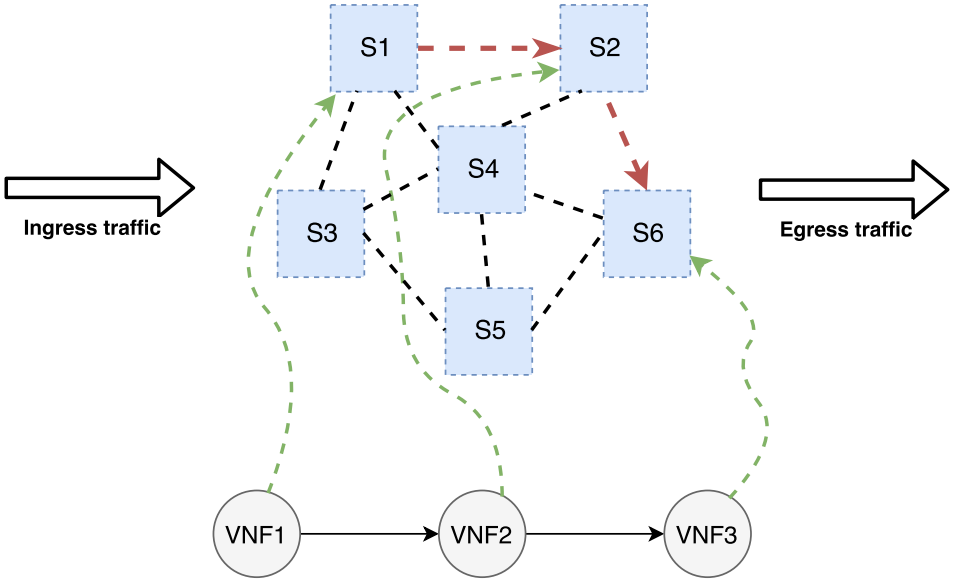



Figure 3.1: VNF placement problem: find a location of each VNF in the chain on the available servers according to a specific goal, e.g., latency, bandwidth, resource utilisation, and number of forwarding rules

size of forwarding rules due to the limits in the TCAM memory of SDN switches. The online formulation is for online load balancing on the available switches. OpenBox [67] proposed an SDN-based framework for developing, deploying, and managing network functions. The platform is composed of data plane entities called OpenBox instances (OBIs) and logically-centralised control plane, called OpenBox controller (OBC). One of the interesting ideas presented in this work is how to move a typical core logic (usually computationally-intensive) of multiple network functions to the control plane so that that logic will be executed once (instead of executing it each time the packet goes through an NF). ClickOS [68] is a runtime platform for virtual NFs based on the Click modular router as the underlying packet processor and running on Xen MiniOs. ClickOS provides I/O optimisations for NFs and reduced latency for packets that traverse multiple NFs in the same physical location. CoordVNF [69]: coordinate the resolution of resource allocation problem on network substrate by formulating the chain composition and VNF-FG embedding (VMs hosting NFs) sub-problem to reduce the bandwidth utilisation. Research presented in [15] has demonstrated how poor CPU scheduling can lower throughput of NF chains by 50%, and inefficient NF placement is causing service chains to cross sockets can triple latency and reduce throughput by 60%. The work has presented a way that allows the service chains to share the same CPU cores rather than spread them across multiple cores, despite fewer resources being available.

VNF-VITAL [70], close work to ours dealing with the VNF performance, has

presented a framework for VNF characterisation. The study has been based on Clearwater IMS VNF and two IDS VNFs (Snort and Suricata). It examines the horizontal and vertical scaling impact on the VNF performance regarding the CPU and memory utilisation. However, the study shows no consideration of the possible performance interference incurred by the VNFs since the evaluation did not cover performance of a sequence of VNFs where chained, which we have achieved in this paper. Work presented in [71] has proposed a graph neural network-based algorithm that predicts future VNF components (VNFCs) resource requirements based on the collected CPU and RAM utilisation data. So, it would be possible to proactively allocate necessary resources to the VNFCs even before they could experience performance degradation. However, we believe that in some cases only considering the CPU or memory utilisation to understand the VNF performance would not be sufficient since the CPU usage metric can be misleading such as in the case of pfSense NAT. In the experiments presented in Section 5.2 of Chapter 5, we show how the high CPU usage at pfSense NAT does not reflect the real computation need but it is a consequence of high CPU interrupts caused by the high rate of incoming packets (only 2% effective computation of the total CPU usage). The authors in [2] have focused on an interesting aspect in the service chains. They have illustrated the traffic changing effects of middleboxes and formulated the middleboxes placement problem to reduce the maximum link load ratio. Their proposed VNF chaining is relying on the gain/drop factor of each VNF but neglecting the fact that such chaining should fulfil well-defined requirements such as the order of the VNF within the chain. Otherwise, this could probably lead to breaching the network policies that should have been correctly implemented by the chains. Table 3.3 summarises the service chain related works.

Table 3.3: Summary of service chains related works

| **Ref.** | **Context** | **Affected metric** | **Summary of the main technique** | **Consider VNF performance?** |
|---|---|---|---|---|
| Stratos [53] | Middlebox orchestration | Number of MBs and network utilisation | MBs rack-aware placement + network-aware traffic distribution | Only CPU utilisation |
| Sync [54] | Chain performance optimisation | communication cost and end-to-end delay | A joint consolidation of policy migration and VM placement + shortest path for the policy implementation | No |
| VNF-P [55] | NF placement | Resource utilisation | NFs placement based on their resources requirements (mathematical formulation) | Generic capacity assigned to VNFs |
| [56] | Service Chain placement | Resource utilisation | Service chains based on the physical host limited resources and NF requirements (mathematical formulation) | Generic capacity assigned to VNFs |
| [57] | NF placement | Resource utilisation | Dynamically placing VNFs based on the behaviour of the resources | No |
| [58] | NF design | State of the NF | Store the state of an NF in a separate backend store using DRAM technique | No |
| [59] | MB management | MB state | Explain the MB state management and representation | No |
| LightChain [22] | VNF placement | Flow rules in the switches | Place VNF in a way to reduce the number of rules on the switches using Directed Acyclic Graph (DAG) | No |

| | | | | |
|---|---|---|---|---|
| [60] | NF placement and routing | Resource utilisation and end-to-end delay | Formulate the problem of network function placement and routing as a mixed integer linear programming (MILP) problem | Generic capacity assigned to VNFs |
| Quokka [61] | NF placement | Latency | Define two model of latency at NF based on its packet processing and places NFs to reduce latency | Yes in latency modelling |
| [62] | VNF placement | Bandwidth | Pre-planned allocation of NFs based on changing workload to reduce network resources utilisation including VM migration overhead | No |
| [63] | MB placement | End-to-end delay and bandwidth | Formulate the MB placement problem as 0-1 programming problem | No |
| [64] | VNF placement and chaining | Resource utilisation and end-to-end delay | Formulate the VNF placement and chaining as Integer Linear Programming (ILP) model | No |
| [65] | VNF placement and chaining | Resource utilisation | Formulate the VNF placement as an Integer Linear Programming problem | No |
| [23] | VNF embedding | Cost of VNF mapping on physical network | Use VNF decomposition in resolving the Virtual Network Embedding Problem (VNEP) | No |
| [25] | VNF parallelisation | End-to-end delay and throughput | Run NFs at the same level of the chains for packet parallel processing | No |
| OpenBox [67] | MB design | NA | Provide platform to develop MBs with the possibility to run computationally-intensive tasks shared between NFs at the controller | Yes |

| | | | | |
|---|---|---|---|---|
| ClickOS [68] | MB design | Latency and I/O | Locate MiniOS-based VNF on same physical server | Yes |
| CoordVNF [69] | Resource allocation | Bandwidth | Formulate the chain composition and VNF-FG embedding problem | Generic capacity assigned to VNFs |
| [15] | Resource allocation | Service chain throughput | Allow service chains to share the same CPU cores rather than spread them across multiple cores | Yes |
| VNF-VITAL [70] | VNF characterisation | Resource utilisation | Experimental VNF resource utilisation assessment | Yes |
| [71] | Resource allocation | Resource utilisation | Predicts future VNF components (VNFCs) resource requirements based on the collected CPU and RAM utilisation data | Yes |
| [2] | VNF placement | End-to-end delay and throughput | Place VNFs based on their gain/drop factor | No |

## 3.4 Summary

In the literature review, we have classified the works related to our research in three main categories. 1) works on data centre networks that aim to improve the quality of service and the application performance in the data centres. For example, monitoring network activity and optimising the resources utilisation and workload distribution. 2) works for reducing the network latency. These studies have tackled the latency in two main locations: VMs/hosts and switches. techniques such as optimising traffic scheduling, fine-grained traffic management, and traffic rate limiters at end hosts have improved both latency and throughput. 3) Category of works that aims at improving the performance of service chains provides. For instance, schemes for service chaining and composition, VNF design, and VNF optimal placement. These studies show a lack of considering the VNF performance sensitivity and bottlenecks in the chain composition. They consider the VNFs as black-boxes that all of them in the chains have the same behaviour toward network traffic. Our idea captures the fact that any application can be either I/O- or/and CPU-bound. We consider this to treat the VNFs inside the chain. Our research firstly shows the difference between these types/categories of VNFs and then use the resulting information in the chain composition problem modelling. This is different from what we have found in the literature review which avoids getting into the details of each VNF and hence misses opportunities of improving the performance of service chains.

# Chapter 4

# Research Methodology

In this chapter, we present how we can approach the existing challenges to achieve the aim and objectives initially set. In particular, we justify our choice of conducting testbed experiments rather than using network simulation framework. We also highlight the significance of VNF characterisation as a prior step in defining a service composition scheme where the network latency can be reduced. Moreover, we discuss the limitation of the adopted method, and we describe other methods such as network simulation and emulation.

## 4.1 Finding system characteristics from testbed experiments

One of our research aims is to limit the impact of virtualisation on the network performance of the service chains. So first we need to concretely measure that impact and evaluate to what extent it can influence the computational and network metrics of NFs. Initially, we have made some attempts to simulate a virtualised environment. However, it has turned out to be impractical due to the high complexity of the virtualisation layer and the time constraints. Also, to the best of our knowledge, there has been no study simulating the virtualisation layer itself. In addition to that, NFs have many particularities so their simulation or emulation would be very sophisticated and so, inaccurate and questionable. Therefore, we have proceeded with setting up our testbed on which we can run experiments examining virtualisation and NF characteristics.

### 4.1.1 Studying the virtualisation impact

We study the virtualisation impact on the computational and network performance of a pfSense Firewall. We implement firewall rules using Linux iptables and

compare it with its counterpart in pfSense. We aim to measure the following metrics.

- Throughput (Transactions/second)
- Round Trip Latency ($\mu$s/transaction).
- 90$^{\text{th}}$ Percentile Latency ($\mu$s).
- 99$^{\text{th}}$ Percentile Latency ($\mu$s).

We will use the following software tools for network performance assessment:

- Ping *"is a network administration utility used to test the reachability of a host on an Internet Protocol network. It is also used to measure the Round-Trip Time (RTT)"* [72].
- hping3 *"is a command-line oriented TCP/IP packet assembler/analyzer. The interface is inspired to the ping(8) Unix command, but hping is not only able to send ICMP echo requests. It supports TCP, UDP, ICMP and RAW-IP protocols, has a traceroute mode, the ability to send files between a covered channel, and many other features"* [73].

### 4.1.2 Characterising the NF performance

We aim to determine the factors that can influence the performance of VNFs. For example, understand the VNFs' behaviour towards different types of traffic and the role of their underpinning software implementation on their performance, and evaluate how the order of VNFs can impact the overall network performance of the service chains.

For this purpose, we intend to use one commodity server which is enough to run multiple NFs in separate VMs running Ubuntu 14.04 and FreeBSD. We study the following NFs:

- pfSense (Firewall and NAT) *"is a free, open source customised distribution of FreeBSD specifically tailored for use as a firewall and router that is entirely managed via web interface. In addition to being a powerful, flexible firewalling and routing platform, it includes a long list of related features and a package system allowing further expandability without adding bloat and potential security vulnerabilities to the base distribution"* [74].
- OVS switch, as a monitoring NF, *"is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to*

*enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g. NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag). It is also used for traffic monitoring"* [75].

- Snort IDS/IPS *"is an open source network intrusion prevention system, capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis, content searching/matching, and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts, and much more"* [76].

- Suricata IDS *"is a free and open source, mature, fast and robust network threat detection engine. The Suricata engine is capable of real-time intrusion detection (IDS), inline intrusion prevention (IPS), network security monitoring (NSM) and offline pcap processing. Suricata inspects the network traffic using powerful and extensive rules and signature language and has powerful Lua scripting support for detection of complex threats"* [77].

### 4.1.3 Running big data applications on a cluster of IoT devices

In order to simulate a data centre environment, we created a cluster of Raspberry Pis running application in virtualised setup using Docker. It was an attempt to understand and confirm the effect of virtualisation layers on network performance of data centre applications. This has thoroughly been discussed in Appendix A.

## 4.2 Mathematical modelling

Our problem needs to be expressed mathematically before proposing a heuristic approach. The study will be described in chapter [**?**]. We follow the following steps for the mathematical modelling and formulation.

- **Define the problem variables and constraints:** We split the initial chain composition problem into two sub-problems, namely, VNF instantiation and traffic distribution. We formulate and model each of these sub-problems. We leverage queuing and graph theories in our modelling. We also define constraints and assumptions in our equations.

- **Prove the problem is NP-hard to introduce the heuristic approach:** We need to prove that the VNF instantiation and traffic distribution problem

is NP-hard by reducing it to the Multiple Knapsack Problem, whose decision is demonstrated NP-hard, into a simplified version of our studied problem. This means it is impossible to find an optimal solution to the problem and then a heuristic approach should be followed.

- **Propose the algorithms to resolve the research problem:** We intend to propose a heuristic solution resolving the formulated problems. Then, we need to develop the proposed algorithms in a testbed environment. We intend to use OpenStack to manage VNF instantiation and traffic steering.

## 4.3 Tesbted evaluation

We aim to use the knowledge from the VNF characterisation experiments to design a chain composition to reduce network latency in the service chains. Only one server would not be sufficient since we need to run multiple instances of NF and set up at least two subnets. For this purpose, we can use the available two commodity servers in the lab on which we can deploy OpenStack (Newton Release).

OpenStack has a modular architecture with various code names for its components [78, 79]:

- Keystone (Identity Service) is a shared service that provides authentication and authorisation services throughout the entire cloud infrastructure. The Identity service has pluggable support for multiple forms of authentication.

- Nova (Compute) provides services to support the management of virtual machine instances.

- Swift (Object Storage) provides support for storing and retrieving arbitrary data in the cloud. The Object Storage service provides both a native API and an Amazon Web Services S3-compatible API. The service provides a high degree of resiliency through data replication and can handle petabytes of data.

- Cinder (Block Storage) provides persistent block storage for compute instances. The Block Storage service is responsible for managing the life-cycle of block devices, from the creation and attachment of volumes to instances, to their release.

- Glance (Image Service) is the Image Registry, it stores and manages guest (VM) images, Disk Images, and snapshots. It also contains prebuilt VM template. Instances are booted from glance image registry.

- Neutron (Networking) provides various networking services to cloud users (tenants) such as IP address management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies). This service provides a framework for software-defined networking (SDN) that allows for pluggable integration with various networking solutions.

- Horizon (Dashboard) provides a web-based interface for both cloud administrators and cloud tenants. Using this interface, administrators and tenants can provision, manage, and monitor cloud resources. The dashboard is commonly deployed in a public-facing manner with all the usual security concerns of public web portals.

- Ceilometer (Telemetry) is responsible for metering Information. It can be used generate bills and based on the statistics of usage. Its API can be used with external billing systems. Administrators can create certain alarms that are triggered based on performance statistics.

- Heat (Orchestration) creates a human and machine-accessible service for managing the entire lifecycle of infrastructure and applications within OpenStack clouds. It contains human-readable templates with simple instruction that is read by the Heat Engine. Heat along with Ceilometer can create an auto-scaling the cloud.

## 4.4 Limitations of the method

Running experiments on testbed is usually reproducible, provides accurate results, and its outcomes can be applied in production systems. We have deployed the same applications like the ones used in data centres or workplaces. However, the method fails in the scalability test since its environment cannot support large workloads and has a few computational instances. To overcome this problem, we stress the existing infrastructure to approach the real production environment conditions. Nevertheless, the question arises whether we still have the same outcomes if we apply our approach in a large-scale environment. The answer is affirmative because we have merely reproduced all the setup and configuration of a production environment but on less number of computational nodes (two servers instead of hundreds).

## 4.5 Other methods

At the beginning of our research, we studied *Sync*, a synergistic scheme to jointly consolidate network policies and virtual machines. *Sync* has been proven effective in reducing the end-to-end delay by nearly 40% and network-wide communication cost by 50% while ensuring full compliance with network policies [54]. We aim to implement *Sync* in both simulated and emulated networks at apprehending the best practices in developing such research techniques and acquiring technical skills usable in testing and evaluating our research proposals. In the following sections, we briefly illustrate these implementations.

### 4.5.1 Simulation

ns-3[1] is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use, it is free software, licensed under the GNU GPLv2 license.

In *Sync* implementation, we develop and helper classes in C++ programming language. The model defines the objects that *Sync* should deal with. For instance, we define what an application, a flow, a policy, and a middle-box (MB) are, and how they can interact between each other, e.g., application should emit a flow, policy should be associated to a sequence of MBs. The helper classes aim to ease the setup of other model classes, e.g., change the flow rate or update the policy routing in case of an update on the chain. Our source code is published on the GitHub repository `https://github.com/wajdihajji/sync-ns-3.git`

### 4.5.2 Emulation

Mininet[2] allows the setup of a realistic virtual network where it is possible to issue commands on hosts or configure the virtual switches. The classes are pre-defined in mininet so that they can be used directly. However, it is sometimes required to customise the classes standard behaviour to fit our needs, for example, change the OVS switch to behave as an MB. We develop *Sync*'s network controller running on an emulated network in Python programming language. The *Sync* implementation is described in the Appendix B.

[1] https://www.nsnam.org/
[2] http://mininet.org/

## 4.6 Summary

Our research methodology is based on three cornerstones. 1) Testbed experiments (see 5.1) to assess the virtualisation overhead on VNF performance and for VNF characterisation. 2) We leverage queuing, and graph theories in a mathematical formulation (see 6.1) and modelling of the VNF instantiation and traffic distribution problem and we prove its NP-harness. 3) We explore other methods such as network simulation and emulation (see B), and we develop SDN controller to evaluate joint consolidation of network policy and VMs.

# Chapter 5

# Virtualisation and NF Characterisation

In this chapter, we have conducted experiments to characterise a set of open source and production NF to understand the impact of the virtualised deployment, their resource utilisation as well as their order in the chain on their performance. We also present an experimental approach exploiting the NF characterisation to improve the end-to-end delay of the service chains. In summary, the contribution of this chapter is as follows:

- We show how moving the hardware-based NFs to the virtual form can have a severe impact on their network performance.
- Through NF characterisation, we have found that NFs can be classified to I/O and CPU bound functions, the former category is sensitive to the traffic rate in packets per second while the performance of the latter is mostly affected by the traffic rate in bits per second.
- We show how understanding the software implementation of VNFs can help to optimise resource utilisation.
- The order of the NF in the chain has an impact on the end-to-end of the traffic traversing it, but taking into account that such reordering may engender policy violation.

## 5.1 Virtual Network Function (VNF)

### 5.1.1 Experiment Setup

We aim in this experiment to measure the experienced performance degradation by a VNF. We utilise three commodity servers in this test-bed, two Pentium and

one Dell servers. The Dell server hosts the VNF (pfSense firewall) while the two other servers host the Client and Server VMs.

We use ping and Netperf utilities to measure the network latency and the TCP request/response (RR) performance. pfSense NAT and firewall are interposed between the internal and exterior networks. We run two sets of tests. The first is where the pfSense firewall is enabled while in the second set, it is disabled and replaced by custom forwarding rules implemented directly on a server OS, a representation of hardware-based NF. Fig. 5.1 summarises the experimental setup.

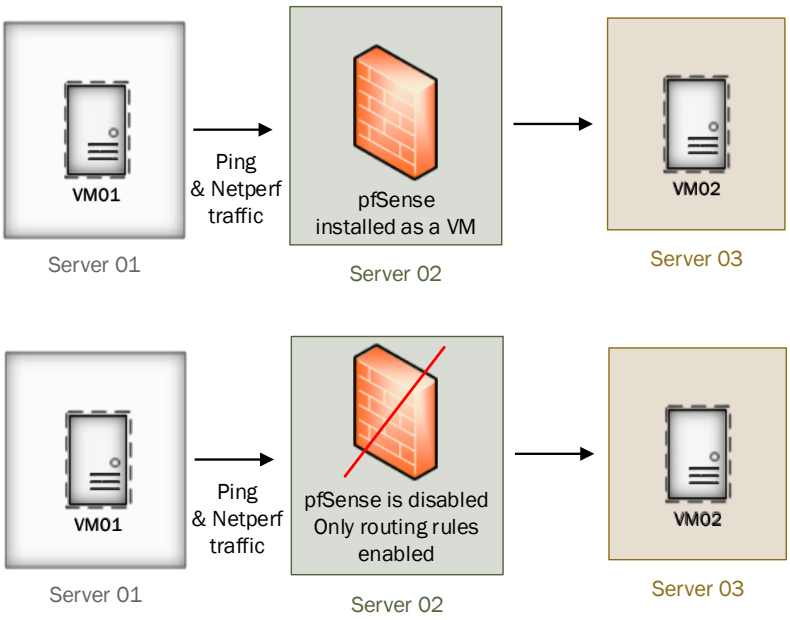



Figure 5.1: Network Topology in pfSense Experiment

- Test set 1: pfSense is enabled. Measure latency and TCP RR for:
  - ICMP traffic between HW-based server and HW-based server.
  - ICMP traffic between HW-based server and VM.
  - ICMP traffic between VM and HW-based server.
  - ICMP traffic between VM and VM.
- Test set 2: we measure the same metrics as in test set 1 but when pfSense is disabled and yet replaced by custom routing rules installed directly on the Ubuntu OS.

For each test stated above, we send 10,000 ICMP packets in 10 seconds. Also, we run Netperf to measure the round-trip latency, TCP throughput (transactions

per second) and variants of latency measurements (e.g. max, min, mean, Stddev, $90^{th}$ and $99^{th}$ percentiles). In our analysis, we notably shed the light on the $90^{th}$ and $99^{th}$ percentiles metrics as they are the most relevant to user experience.

### 5.1.2 Experiment Results

The results are represented In Fig. 5.2. Where pfSense is disabled, the RTT is minimum for the different combinations of sender/receiver whether they are VM or commodity server. Packets take less time to travel from client to server. Whereas in the virtualised firewall, the RTT increases of more than 100%. For instance, in Fig. 5.2a the 99th percentile of RTT in the case of pfSense is disabled is nearly 0.5ms while it is more than 1.1ms where traffic needs to traverse the firewall. The same observation is still true in Fig. 5.2b, 5.2c, and 5.2d where the 99th percentiles are 0.5ms, 0.5ms, and 0.6ms respectively on condition pfSense is running and RTT are more than 1.2ms, 1.2ms, and 1.2ms respectively in the other situation (forwarding by routing rules).

Even the four graphs correspond to different setup and nature of sender/receiver; the network performance is similar when pfSense is disabled, and it is significantly affected in the opposite case (pfSense is enabled). This proves how traversing a virtualised network function can delay packets forwarding and cause considerable latency.

Using Netperf benchmarking tool has helped to measure the TCP request/response performance as well as the latency in different percentiles. Precisely, we have used Omni tests to be able to perform and display several measurements in a single run and on a single output. We have reported four metrics in this test:

- Metric 1: Throughput (Transactions/second)
- Metric 2: Round Trip Latency ($\mu$s/transaction).
- Metric 3: $90^{th}$ Percentile Latency ($\mu$s).
- Metric 4: $99^{th}$ Percentile Latency ($\mu$s).

By analysing the results in Fig. 5.3, there are two dimensions of performance degradation that can be noticed. Firstly, it depends on whether the sender-/receiver is virtualised, for instance, throughput decreases slightly from 1061.06 Trans/s to 1008.78 Trans/s in case of traffic between Server-VM and VM-VM respectively. Latency as well increases from Server-Server to VM-VM, in which case, 90th percentile latency rises from 1162 $\mu$s to 1173 $\mu$s. This means that the network traffic experiences more latency in case both ends are virtualised than the

case they are not. The same remark has been recorded for 99th percentile latency and Round Trip Latency.

Also, performance degradation becomes more perceptible when traffic has been set to traverse a virtualised firewall (pfSense). It is clear in the same Fig. 5.3 how all the measured network metrics are impacted. Throughput is almost the double in case both sender and receiver are not virtualised than in the opposite case. It is 2592.98 Trans/s and 1056.80 Trans/s when the test is done between Server-Server where pfSense is off and on respectively. In the same configuration (Server-Server) Round Trip Latency is 385.657 $\mu$s/trans when pfSense is disabled whereas it is significantly higher in case pfSense is enabled (nearly 946.250 $\mu$s/trans). This difference is attenuated when all the environment is virtualised (both sender and receiver are), it is 991.298 $\mu$s/trans and 498.260 $\mu$s/trans where pfSense is on and off respectively. The degradation is still witnessed for the 90th and 99th latency metrics. For instance, in Server-Server setup, 90th percentile latency is 1162 $\mu$s and 426 $\mu$s where pfSense is on and off respectively, a decrease of more than 50%. In VM-VM configuration, 99th percentile latency has seen the same trend; it drops from 1286 $\mu$s to 634 $\mu$s where switching on and then off pfSense, i.e. a decrease of more also than 50%.

Therefore, two factors are causing the network performance degradation. The nature of sender and receiver as well as of the network function intercepting the traffic. The cause behind this is the virtualisation technology that adds a layer for the packet forwarding either in hosts or network functions. By this, it deepens the problem of congestion and latency.

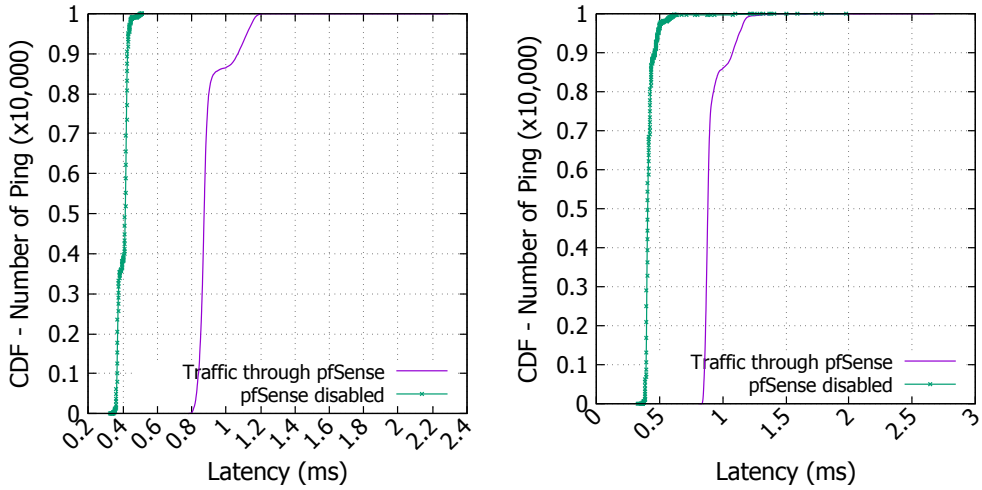
## 5.2 Network Function performance bottlenecks

We have deployed two NFs, namely Snort[1] IDS, configured with community rules[2], pfSense[3] NAT. They are deployed on top of KVM on a server that has 8 cores, 1.2 GHz CPU, 8 GB memory, and Ubuntu 14.04 as OS. We have used hping3[4] packet generator tool to generate testing traffic.

We first investigate the diversity of the CPU performance among different types of NFs. We have used Snort and pfSense, each configured to use one core. For each one of them, we have applied flows comprising large and small packet payload, both at high packet rate (20kpps).

Fig. 5.4a demonstrates the CPU usage of the Snort IDS. For large-payload

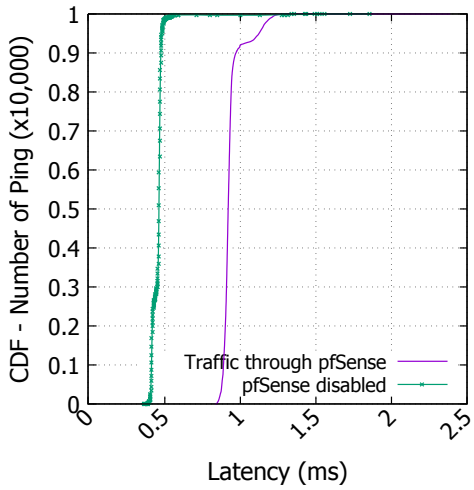
[1]version 2.9.11; https://www.snort.org/
[2]https://www.snort.org/faq/what-are-community-rules
[3]version 2.3; https://www.pfsense.org/
[4]https://linux.die.net/man/8/hping3



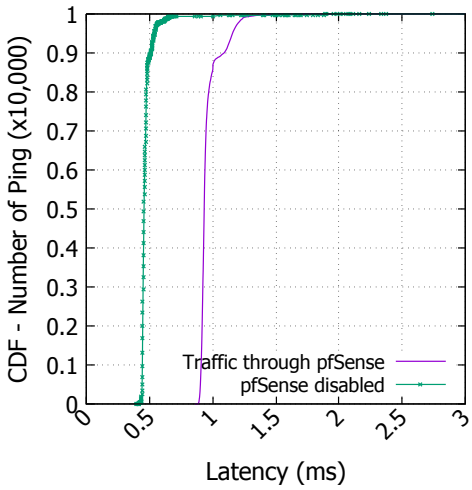

(a) Server to Server




(b) Server to VM




(c) VM to Server




(d) VM to VM

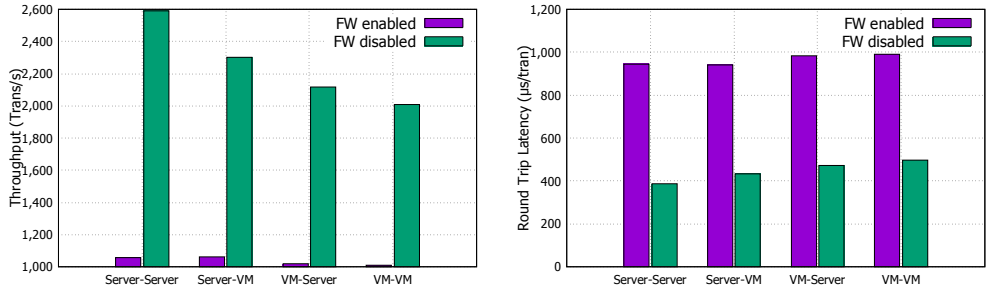
Figure 5.2: Latency measurements

packets, the CPU usage has several fluctuations and is more intense than the case of small-payload packets. This is because IDS aggregates and inspects packets, including payloads, for anomalies. Hence, the larger the payload is, the more time it needs to inspect the packet.

Fig. 5.4b shows that for both types of traffic the CPU usage almost remains the same for the pfSense NAT. This is because NAT only acts on the packet header by replacing the destination address by a predefined one (e.g., an IP in the internal network) regardless of the content of packet payload.

Then, this kind of mapping uses less CPU time comparing to the parsing process seen in Snort.

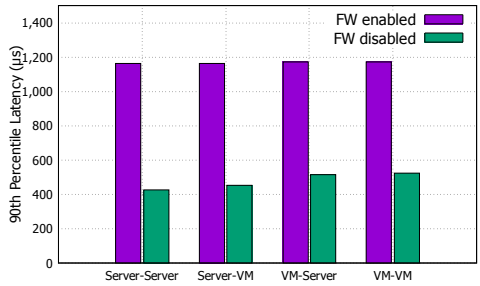
Further, both figures show distinctive levels of CPU usage at the IDS and the NAT (at around 20% and 90% respectively) for flow comprising small payload at high packet rate. The high CPU usage at the NAT is mainly caused by the CPU



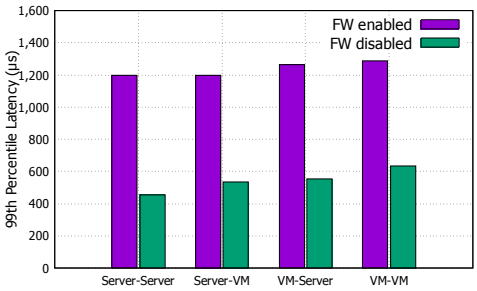

(a) Metric 1




(b) Metric 2




(c) Metric 3

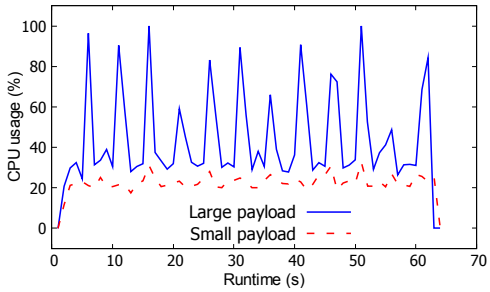



(d) Metric 4

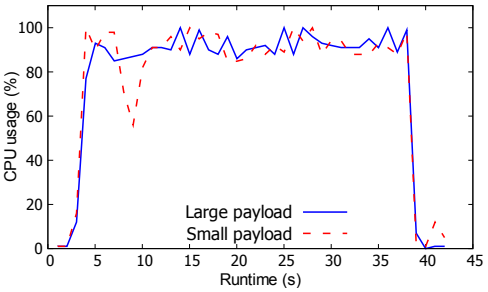
Figure 5.3: Netperf test with virtual VNF (pfSense FW)

interrupts not by the processing effort since the packet header modification is a fractional operation that consumes a few CPU cycles. To verify this assumption, we have analysed the CPU usage details, and we have found that nearly 90% of CPU usage is dedicated to the interrupts while only 1.7% was used by the NAT to perform its computation.

This experiment demonstrates the crucial impact of network traffic type on the CPU usage in different VNF categories.

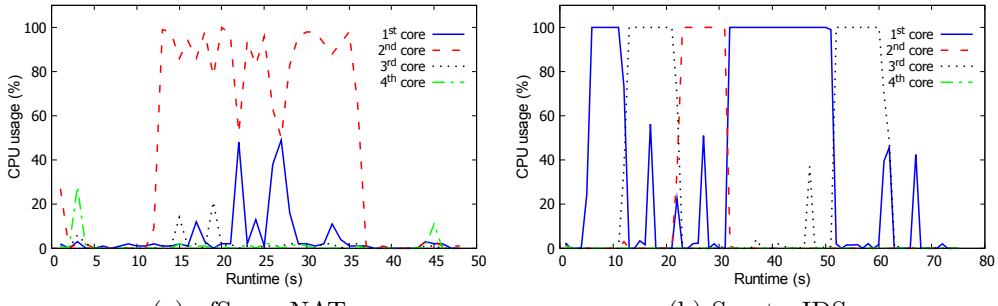



(a) Snort IDS



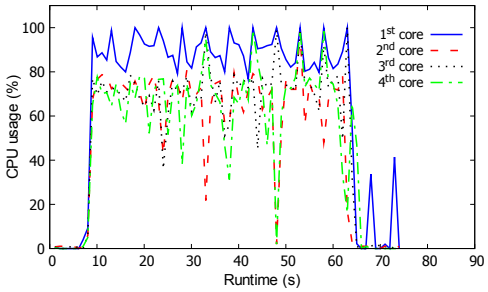

(b) pfSense NAT

Figure 5.4: Different impacts of high packet rate and throughput on the resources utilisation at network functions




(a) pfSense NAT

(b) Snort - IDS




(c) Suricata - IDS

Figure 5.5: NF implementation impact on the CPU usage

## 5.3 Network Function Software implementation

Next, we increased the number of allocated CPU cores from one to four for Snort and pfSense VMs to improve their performance and we applied high-throughput traffic. Surprisingly we have observed that Snort consistently uses only one core at a time. As we can see from Fig. 5.5b, it was using first core 3s to 12s, the third core from 11s to 22s, and second core from 21s to 32s. Fig. 5.5a illustrates the same behaviour for the pfSense NAT but with sporadic and low fluctuations of the other cores, which can be caused by background OS tasks. Our further investigation by examining the source code of both Snort and pfSense has concluded that this observation is attributed to their single-threaded implementation (see Fig. 5.6).

To confirm our conclusion, we identified and tested a multi-threaded version of IDS, Suricata[5], using default rules and the same experimental setup as Snort. Suricata is multithreaded at the Detect stage as shown in Fig. 5.6.

As expected, the results illustrated in Fig. 5.5c demonstrates that the four allocated cores are simultaneously at high usage level during the runtime.

This experiment shows that even VNFs of the same type can have remarkably diverse performance due to the underpinning implementation techniques.

To sum up, we illustrate in Table 5.1 the studied characteristics in some NFs. The gain/drop factor of a VNF means the ratio of incoming to outgoing traffic

[5]version 3.2.4; https://suricata-ids.org/

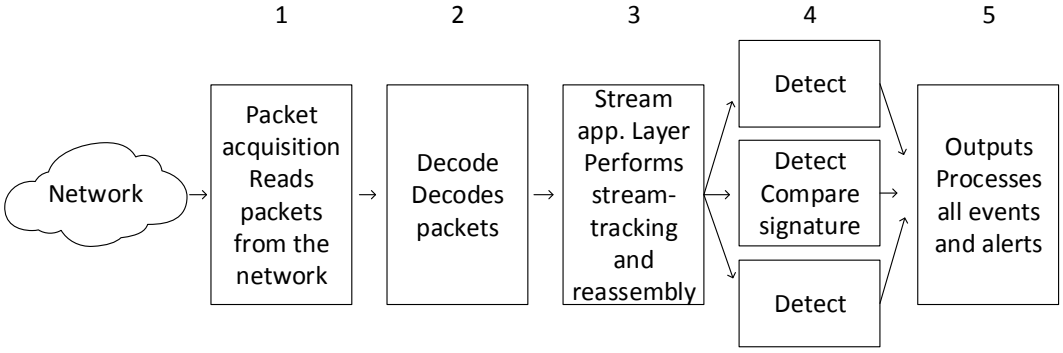



Figure 5.6: Parallel "Detect" in Suricata IDS

Table 5.1: Classification of studied VNFs

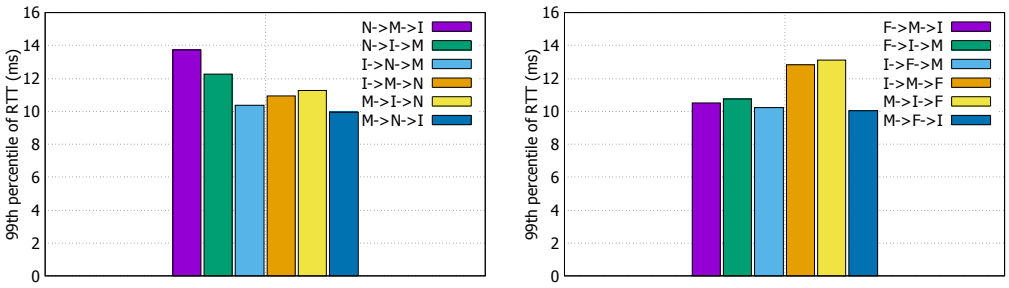
| NF | Action | Compute attribute | Gain/drop factor |
|---|---|---|---|
| **FlowMon** | R header | I/O bound | 1 |
| **IDS** | R header/payload | CPU bound | 1 |
| **Firewall** | R header | I/O bound | $0 \leq x \leq 1$ |
| **NAT** | R/W header | I/O bound | 1 |
| **Load Balancer** | R/W header | I/O bound | 1 |
| **Redundancy Eliminator** | R/W paylaod | CPU bound | $0 < x \leq 1$ |

volume at that NF; it mainly depends on the logic implemented by the NF.

## 5.4 Network Function reordering in the network chain

In this experiment, we change the order of some VNFs in the chains. For example, the first chain has a NAT, IDS, and FlowMon, the second has an FW, IDS, and FlowMon. With three distinct NFs, there are six different possible permutations of VNF sequence in each chain. For all the permutations, we apply the same network traffic.

The end-to-end delay of the chain is different from one permutation to another and is also dependent on the VNFs involved in the experiment. Here are the main




(a) Network Chain **N**AT–**I**DS–Flow**M**ON (b) Network Chain **F**W–**I**DS–Flow**M**ON

Figure 5.7: Impact of VNF order on the end-to-end delay of network chains. However, we ignore the implementation of the VNF.

remarks for Fig. 5.7a.

- When the NAT is in the first order, we get the highest RTT.
- When the NAT is in the third order, it is less high RTT.
- When the NAT is in the second order - in the middle - we get the lowest RTT.

When the NAT is in the middle of the chain, ingress and egress packets take more time to reach it (they go first through either FlowMon or IDS), and this reduces the packet rate and the queuing time at its level. The NAT is then able to influence the performance of the whole chain as it is the weak link (I/O-bound NF). When the NAT is in the first order, it receives a high packet rate (directly from the source), and that creates congestion, so delay. When it is in the third order, ingress packets have a less high rate (compared to the case of the NAT in the first position), but egress packets (going back to the source) comes directly from the destination (so no processing).

The same reasoning can still explain Fig. 5.7b but in this case, there are two weak VNFs which are the FlowMon and the IDS compared to the FW. So these former VNFs will influence more the end-to-end delay in the chain.

However, we should consider that in some cases the VNF re-ordering could violate the network policy implemented by the chain.

## 5.5 Proof-of-concept experiments

In this section, we describe the aim and setup of the proof-of-concept experiment.

### 5.5.1 Idea

We sum up the finding on the VNF characterisation as follows.

- Single-threaded application cannot use all available cores on commodity servers, in particular, single-threaded NFs suffer from this limitation.
- Single-threaded NF that performs one function, e.g. NAT, IDS, Traffic shaping, etc. need only one core to run properly on a multi-core processor (in analogy to a bus carrying one passenger, who obviously needs only one seat.).
- NF has distinct sensitivity towards packet rate and payload size. For instance, the performance of NFs handling packet payload affects the network performance of throughput-intensive traffic, the packet rate impacts other NFs writing/reading packet header, and their performance influences both latency-sensitive and throughput-intensive traffics.

### 5.5.2 Method

Practical steps for the chain composition setup.

- Allocate only one core to the single-threaded NFs so that we save more cores that can be utilised by other applications or particularly to create multiple instances of the same NF for parallel packet processing.
- For traffic entering NFs dealing with packet headers, we distribute its flows based on their packet rates. We make sure that each flow goes to the right instance of the NF depending on its packet rate. This will attenuate or even avoid the performance degradation experienced by the NF when it is receiving packets at a high rate and where packet loss likely occurs.
- For traffic entering NFs dealing with packet payload, we make a distribution of its flows based on both packet rate and payload size (i.e. the throughput), so each flow goes to the right instance of such NFs.
- Packet rate and payload size firmly depend on the application that creates the flow. For example, online gaming is a real-time application which is latency-sensitive, this means, e.g. when players take simple actions that go over the network, the packets carrying them are not large but they need to be transmitted as soon as possible. Otherwise, troubles can be seen on the application (see also the example of video control - Pause/Play/Volume on YouTube). Besides, for example, Hadoop exchanges data between the workers, packets here are large (fat) and at a high rate.

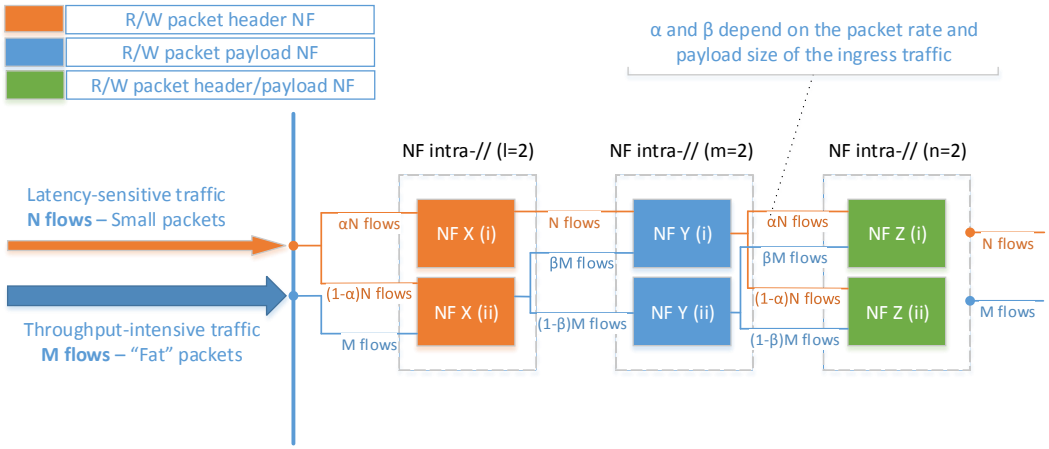



Figure 5.8: A particular use case where N flows belong to SH traffic category and M flows to FL category – see "Traffic characteristics" for SH and FL meaning

**Traffic characteristics**

We consider different types of traffic as shown below.

- Traffic where the packets have a large payload (fat packets, F for fat) and high packet rate (H for high): we call this traffic category FH
- Packets with large payload and low packet rate (L for low): FL
- Packets with small payload (S for small) and high packet rate: SH
- Packets with a small payload and low packet rate: SL

### 5.5.3 Experiment set-up

We thus consider FH, FL, SH, and SL traffic. We use 4 VMs as senders; each couple sends the same kind of traffic, first two VMs always have large packet payload (F), the two remaining VMs always have small packet payload (S) so that in all cases we introduce a candidate of each traffic (throughput versus latency). We use two VMs for the same traffic to show and prove the traffic distribution benefit/effect.

Without loss of generality, we send ICMP packets using hping3 network tool so that we can customise the packet payload and rate. All packets belonging to FH traffic carry a payload of the size of Ethernet MTU (1500 bytes) and at a rate of 20kpps, FL: 1500 bytes payload at 2kpps, SH: 16 bytes payload at 20kpps, and lastly, SL: 16 bytes payload at 2kpps.

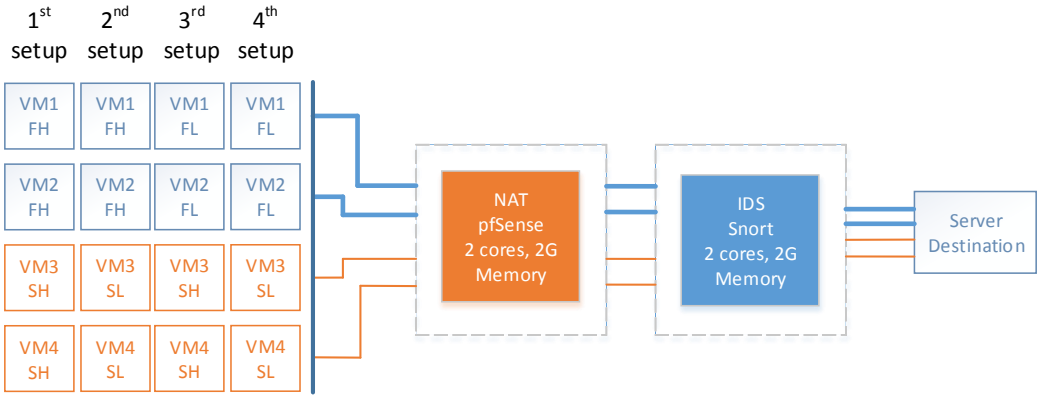



Figure 5.9: Traffic traversing a service chain composed of a NAT and an IDS

We consider a service chain composed of two virtualised NFs (using KVM virtualisation technology on Ubuntu server 14.04), pfSense 2.3.3 as a NAT (R/W packet header), and Snort 2.9.9 as an IDS (R packet payload).

Sender VMs are on server-01 (8 cores/1200.00 MHz, 8G memory), virtualised NFs are located on server-02 (8 cores/1256.718 MHz, and 8G memory), the destination is a server with four cores/1998.000 MHz and 4G memory.

Here is the possible set-ups for the senders:

- VM1 (FH), VM2 (FH), VM3 (SH) and VM4 (SH): we call this set-up FHSH
- VM1 (FH), VM2 (FH), VM3 (SL) and VM4 (SL): FHSL
- VM1 (FL), VM2 (FL), VM3 (SH) and VM4 (SH): FLSH
- VM1 (FL), VM2 (FL), VM3 (SL) and VM4 (SL): FLSL

### Baseline experiment

We measure the packet loss and network latency and throughput through the service chain when we apply each set-up of senders described in the previous section. We allocate two cores and 2G memory for each NF.

### Proof-of-concept experiment

We change the resources dedicated to the NFs to be one core and 1G memory for each one of them. However, we create two instances of each NF (each has one core and 1G memory), and we apply our approach under each traffic set-up (FHSH, FHSL, FLSH, FLSL), like what we exactly do in the baseline experiment. We measure again the packet loss, latency and throughput and lastly, we compare the results of the two experiments.

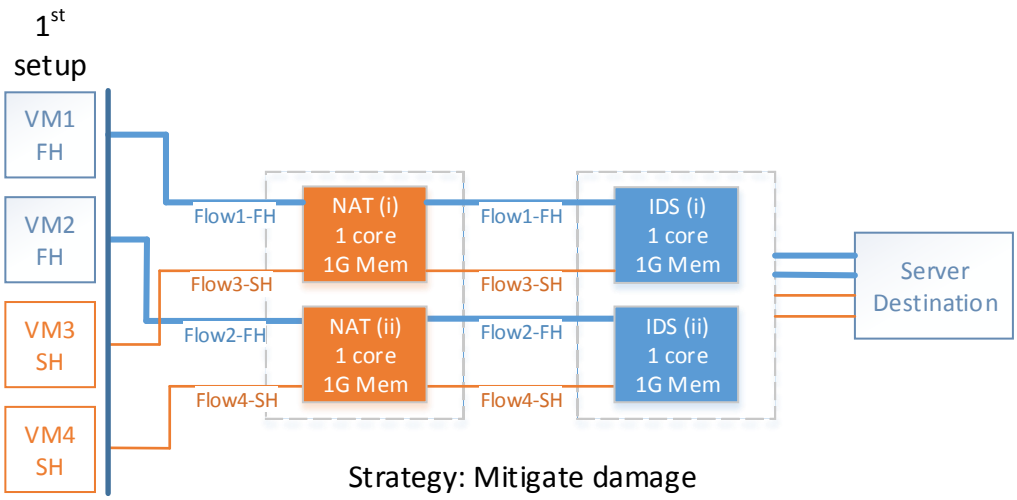



Figure 5.10: Applied traffic: FHSH – Packets at high rate are sent to the NFs, extreme network conditions. Traffic distribution takes place to reduce or avoid the packet loss and then to improve the network latency and throughput as much as possible

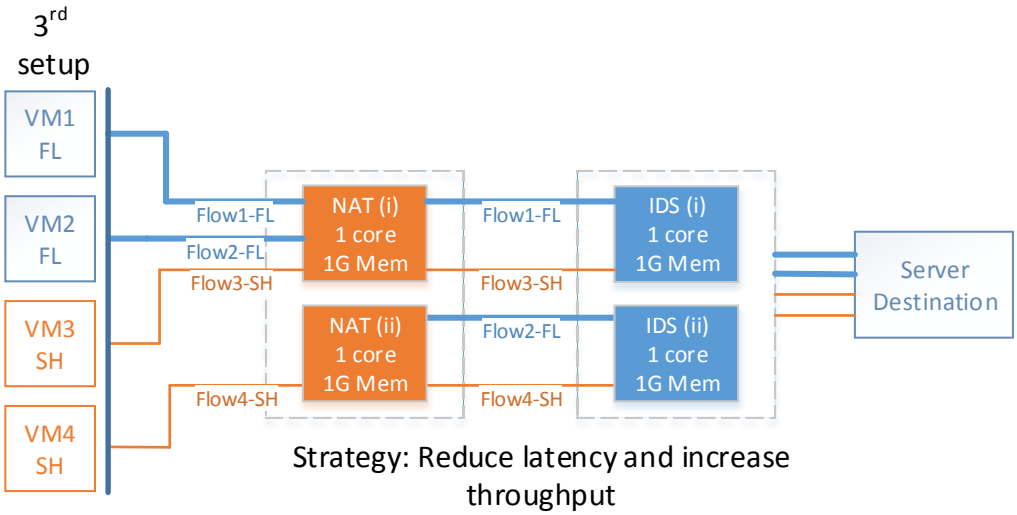



Figure 5.12: Applied traffic: FLSH – Same objective as for the traffic category FHSL

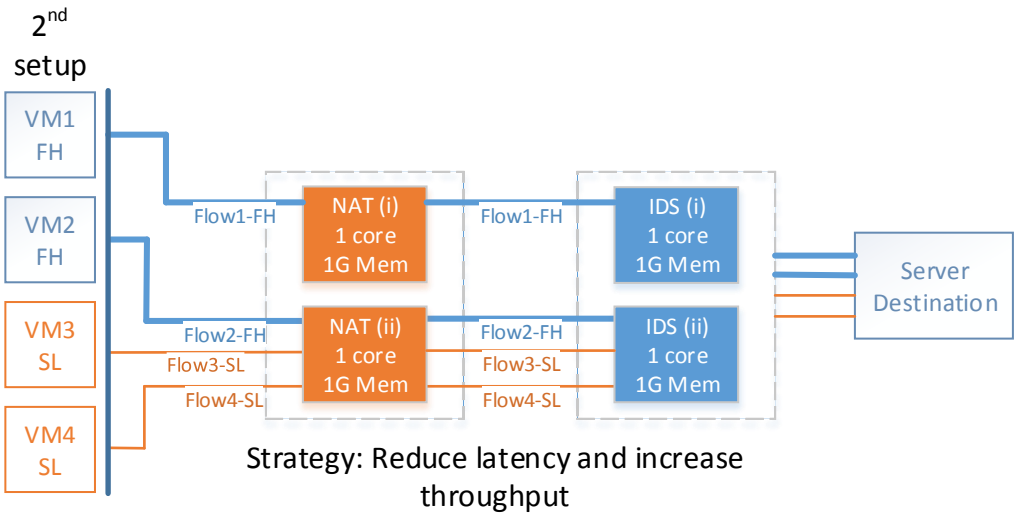



Figure 5.11: Applied traffic: FHSL – Traffic at high and low packet rates, the traffic distribution plays crucial role to reduce latency and increase throughput, and also to reduce the packet loss

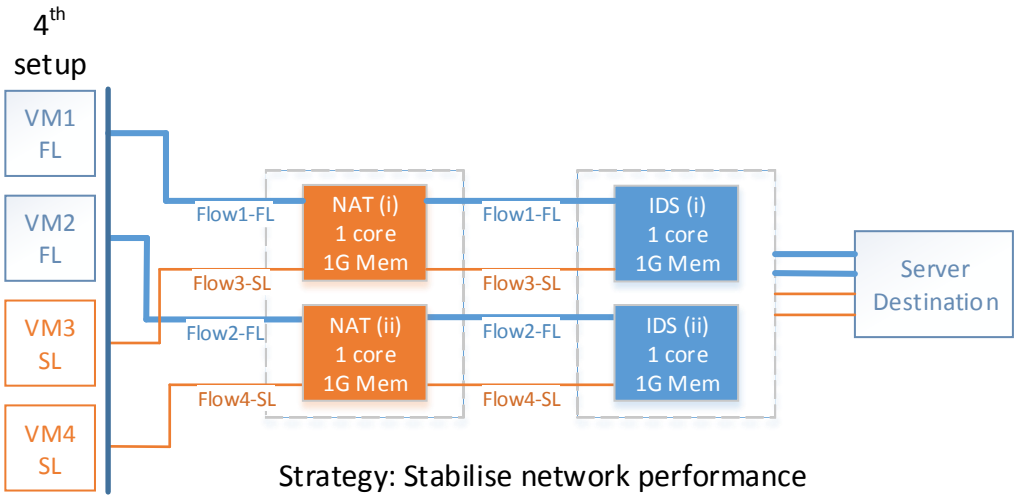



Figure 5.13: Applied traffic: FLSL – The network conditions are ideal (traffic at low rate), the objective of traffic distribtion is to stabilise the network performance

### 5.5.4 Results and conclusion

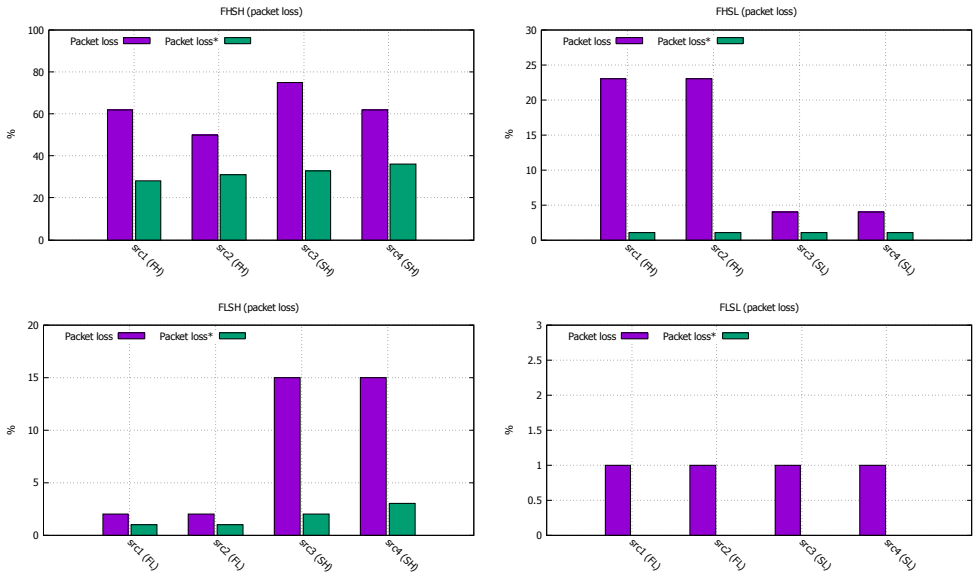



Figure 5.14: Packet loss has been significantly reduced for FH and SH traffics. We must note that these results have been conducted as a proof of concept for proving the usability of our approach. We stressed our experiment environment to highlight the benefit of our idea. So in real data centre, resources will be sufficiently supplied to the infrastructure and we will not that high packet loss.

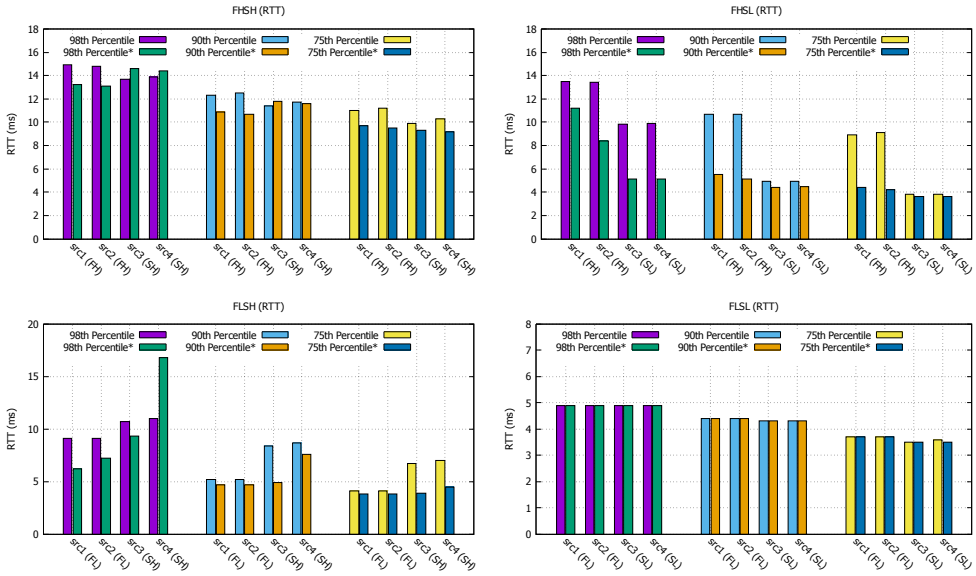



Figure 5.15: Network latency decreases for all kinds of traffic - except for FLSL where it remains the same

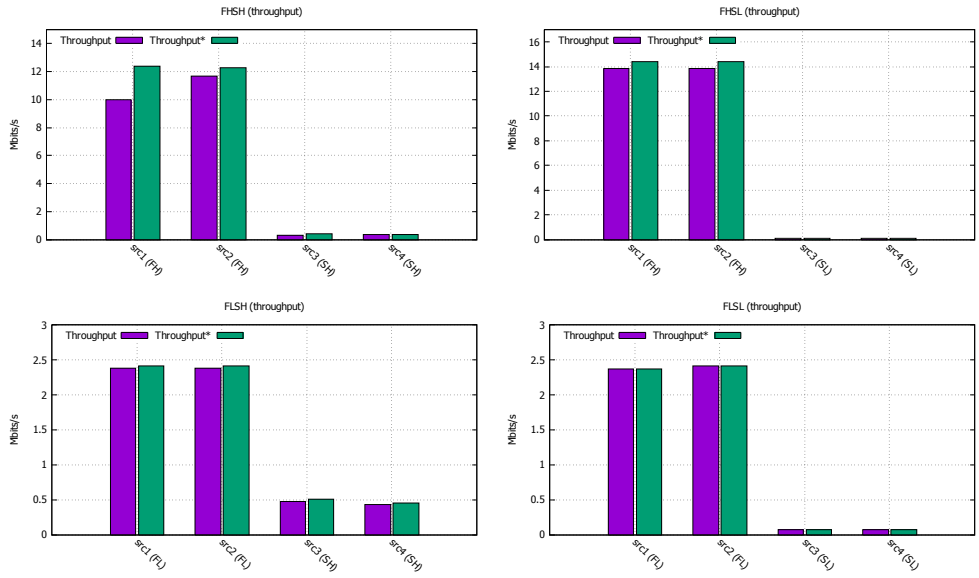



Figure 5.16: Network throughput increases more for FH traffic

- FHSH (Fat-High and Small-High)
  - Affected traffic: both throughput-intensive and latency-sensitive.
  - Packet loss dropped from 62% to 32% i.e., an improvement of 48.59%.
  - RTT decreased by 6.87%.
  - Throughput increased by 13.98%.
- FHSL (Fat-High and Small-Low)
  - Affected traffic: throughput-intensive.
  - Packet loss for throughput-intensive traffic reduced from 23% to 1%, i.e., by 95.65%.
  - Latency dropped by 43.27%.
  - Throughput increased by 4.10%.
- FLSH (Fat-Low and Small-Low)
  - Affected traffic: latency-sensitive.
  - Packet loss reduced from 15% to 3%, i.e., by 83.34%.
  - Latency decreased by 15.10%.
  - Throughput increased by 6.75%.
- FLSL (Fat-Low and Small-Low)
  - Ideal conditions, none of the traffic is affected.

- Packet loss went from 1% to 0%.
- Throughput and latency are still the same.

In a nutshell, our results show how packet loss is reduced by 75.86%, latency dropped by 21.74%, and network throughput increased by 8.28%.

This experiment illustrates how the network chain performance cannot simply be improved by scaling out/in (horizontal scaling) or up/down (vertical scaling) the VNFs comprising the chain. It depends on the implementation and the sensitivity of VNFs towards the network traffic. For instance, the implementation of some VNFs limits the benefit of vertical scaling especially when the VNFs is single-threaded application.

## 5.6 Summary

In this chapter, we have evaluated the virtualisation impact on the network performance of a firewall. Then, we have conducted a set of experiments targeting a firewall, a NAT, a flow monitor, and two IDSs. We have experimentally demonstrated that the VNF can be I/O or CPU-bound depending on how it is handling the traffic. The VNF underlying software implementation can significantly determine its resource utilisation so understanding this detail can be helpful in optimising the resource allocation. VNF reordering is not usually possible since it may violate the network policies and could bring more harm than benefit, that's why reordering was not considered in our solution.

The idea behind this study is to leverage the resulting knowledge in the mathematical formulation of the VNF instantiation and traffic distribution problem, which will be the focus of the next chapter.

# Chapter 6

# Dynamic Network Function Composition

In this chapter, we propose *Natif*[1], a VNF-Aware VNF insTantIation and traFfic distribution scheme. *Natif* proposes a VNF instantiation, and traffic distribution that relates the VNF characteristics to the network flows attributes. Based on the study presented in the previous chapter, we propose to classify the VNFs as I/O bound and CPU bound functions. For example, when dealing with a VNF classified as I/O bound, *Natif* calculates the number of needed instances based on the packet rate of the ingress flows, and it *prioritises* the packet rate criterion over the network throughput in the traffic distribution. For a CPU bound NF, it *mostly* relies on the throughput either in the VNF instantiation or the traffic distribution. We have conducted proof-of-concept experiments as an initial step to prove *Natif*'s efficiency compared to a typical chain composition set up.

## 6.1 Problem formulation and modelling

### 6.1.1 Problem notations

#### Network Function

We consider the virtual form of NFs. Let $M = \{m_1, m_2, \ldots\}$ be the list of NFs. Each $m_i \in M$ is defined by a set of parameters. For instance, $m_i.s$, $m_i.cpu$, and $m_i.mem$ represent, respectively, the server ID where $m_i$ is deployed, the required CPU and memory by $m_i$. Let also the parameter $m_i.proc$ (*proc* for *processing*) denote whether the VNF is I/O bound or CPU bound. If $m_i.proc = 1$, the VNF is CPU bound, else, i.e. $m_i.proc = 0$, it is I/O bound. In case the VNF is both

[1] https://github.com/wajdihajji/natif.git. *Natif* is a French word that means innate, original, or natural

CPU and IO bound, for example, a DPI, we set $m_i.proc = 2$.

Depending on $m_i.proc$, we evaluate the processing capacity distinctively. For an I/O bound NF, we quantify it by the number of processed packets per second, referred to by $m_i.c_{pps}$, for a CPU bound category, we rely on the amount of data parsed per second, i.e. the throughput, which is noted $m_i.c_{dps}$.

We also consider the software implementation of the NFs. We use the parameter $m_i.thd$ for this. If $m_i.thd = 0$, $m_i$ is single-threaded VNF and if $m_i.thd = 1$, it is multi-threaded. In particular, if $m_i.thd = 0$, we impose that the number of allocated cores to $m_i$ should be equal to the number of its created instances, so, each instance only uses one core.

We define the gain/drop factor for an VNF $m_i$ as $m_i.gd$, where $m_i.gd \in \mathbb{R}_{\geq 0}$. For example, $m_i.gd = 1$ for NAT or IDS since they perform header field mapping or traffic inspection, and $0 < m_i.gd \leq 1$ for Redundancy Eliminator (RE) as it reduces the volume of egress traffic by removing redundant data.

The way we calculate the gain/drop factor varies in accordance with the VNF category. Thus, we use two versions of $m_i.gd$: $m_i.gd_p$ to note the gain/drop factor regarding the packet rate, when $m_i.proc = 0$, and $m_i.gd_d$ for throughput gain/drop factor, in case $m_i.proc = 1$.

The end-to-end delay of flow when traversing a VNF is composed of a transmission delay and a processing delay. The former depends on the link capacity (the bandwidth) which is considered relatively stable in data centres [34]. The latter is due to the processing time, i.e. service time, of the incoming traffic plus the latency incurred by the packet queuing at the NFs, i.e., the waiting time. Let $D$ be the transmission delay matrix, where $D(m_i, m_j) = D(m_j, m_i)$ is the delay between $m_i$ and $m_j$, and $D(m_i, m_j) = -1$ if the delay is unknown or $m_i$ and $m_j$ are not reachable. We define the service time as the time that the VNF takes to process a packet or a bit of data based on its category.

The service time of a VNF $m_i$ is given as follows.

$$t_s^i = 1/m_i.cap \tag{6.1}$$

where,

$$m_i.cap = \begin{cases} m_i.c_{pps}, & m_i.proc = 0 \\ m_i.c_{dps}/\delta_i, & \text{otherwise} \end{cases} \tag{6.2}$$

where $\delta_i = avg(pkt_{size})$ is the average packet size in the flows traversing $m_i$.

For simplicity and without loss of generality, we consider M/D/1 queue at VNFs and VNFs process packets in a First-Come-First-Service (FCFS) discipline. We refer by $\lambda^i$ to the arrival rate of all flows traversing $m_i$, i.e., how many packets per second traversing the VNF $m_i$. $\lambda^i$ is determined by a Poisson process [80] and

is defined as follows. The use of Poisson process is backed as it is widely used to model random points in time and space such as times of radioactive emissions and arrival times of customer at the service point.

$$\lambda^i = \sum_{f_i \in E(*, m_i)} f_i.pr \tag{6.3}$$

Given the utilisation $\rho_i = \lambda^i \times t_s^i$, the average waiting time $t_w^i$ of $m_i$ is

$$t_w^i = \frac{t_s^i \times \rho_i}{2(1-\rho_i)} = \frac{\lambda_i \times {t_s^i}^2}{2(1-\lambda_i \times t_s^i)} \tag{6.4}$$

And the processing delay of $m_i$ is:

$$t_p(m_i) = \begin{cases} t_s^i, & \lambda_i \leq m_i.cap \\ t_w^i + t_s^i, & \text{otherwise} \end{cases} \tag{6.5}$$

**Network chain**

We represent a chain as a Directed Acyclic Graph (DAG) *G = (V, E)*, where $V$ is the set of vertices representing a subset of VNFs in $M$, while $E$ is the set of edges representing the links joining them. The first node of the DAG is called entry VNF, and the last node is called exit NF. We assume that all DAGs have only one entry and one exit NF. If a DAG has more than one entry or more than one exit, one entry VNF and one exit NF, both with zero cost (functionless NF), are added to the graph, along with costless (no delay) edges connecting them to the original entry/exit nodes.

Each path from the entry to the exit nodes is a traversable path called a *branch*. When a procedure of vertical scaling is performed in an NF, e.g., increasing or decreasing the number of its instances, new branches are created in the graph. In this case, we call these new branches as *sub-branches*. Let $CH = \{ch_1, ch_2, \ldots\}$ be the list of deployed network chains in the data centre, $B_i$ and $SB_i$ the lists of *branches* and *sub-branches*, respectively, in the chain $ch_i$.

Initially, there were three branches in Fig. 6.1; $b_1$, $b_2$ and $b_3$. Then, after scaling out $nf2$ and $nf3$, the chain has got six *sub-branches*.

The number of *sub-branches* equals to the sum of the product of the number of instances of each distinct VNF on all *branches*.

$$|SB_i| = \sum_{b \in B_i} \prod_{m \in b} |M| \tag{6.6}$$

where $SB_i$ and $B_i$ are the lists of sub-branches and branches, respectively, in $ch_i$, $M$ is the list of instances of the VNF $m$, and $m \in b$ means $m$ is deployed on the

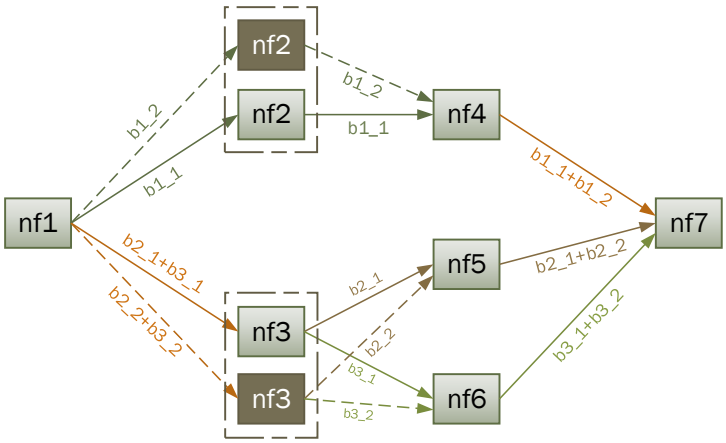



Figure 6.1: Chain sub-branches

branch $b$. For example, in Fig. 6.1, $|SB| = (1 \times 2 \times 1 \times 1) + (1 \times 2 \times 1 \times 1) + (1 \times 2 \times 1 \times 1) = 6$.

### Flows

We examine flow-based traffic. Each set of flows traverses the appropriate network chain according to the policy defined for it. We note $F = \{f_1, f_2, \ldots\}$ as the list of existing and active flows. Each flow has a non-static 5-tuple since it can be altered by one of the VNF while traversing the chain (e.g. a NAT). Nevertheless, it is still possible to determine in advance the flow 5-tuple at each hop of the chain as we know what type of operations the VNFs perform. The number of flows can also be known in the data centre, since we know the ensemble of communicating hosts. For example, the application server interacts with the interface and stores data to the database instance.

We define the following variables to describe a network flow. Flow throughput by $f_i.thp$, packet rate by $f_i.pr$, source IP by $f_i.sip$, destination IP by $f_i.dip$, source port by $f_i.sport$, destination port by $f_i.dport$, and protocol by $f_i.proto$. To determine the attributes of the egress flow at the VNF $m_i$, we refer to the following equations:

$$f_{out}.pr = \begin{cases} m_i.gd_p \times \sum_{f_i \in E(*,m_i)} f_i.pr, & \text{if } \lambda_i \leq m_i.cap \\ m_i.gd_p \times m_i.c_{pps}, & \text{otherwise} \end{cases} \tag{6.7}$$

$$f_{out}.thp = \begin{cases} m_i.gd_d \times \sum_{f_i \in E(*,m_i)} f_i.thp, & \text{if } \lambda_i \leq m_i.cap \\ m_i.gd_d \times m_i.c_{dps}, & \text{otherwise} \end{cases} \tag{6.8}$$

where $f_i \in E(*, m_i)$ refers to all the flows entering $m_i$.

We primarily consider the flow packet rate and throughput in our model. We use these two metrics when they are available. Otherwise, we use the statistics collected at the switches to feed the prediction model to infer them (section 6.2.3).

**Policy**

Let $P = \{p_1, p_2, \ldots\}$ be a list of policies. We note $p_i.list$ the list of VNFs used by $p_i$ and $p_i.len$ its length. Each policy can rule one or many flows, and it is in many-to-one correspondence with the network chain. We assume the number of policies equals to the number of branches of all chains, i.e., $\forall p \in P, \exists! ch_i, \exists! b \in B_i$ *s.t. b implements p.*

### 6.1.2 Problem definition

The expected delay of a flow $f_i$ traversing a branch $b_i$ governed by a policy $p_i$ is construed as follows.

$$\begin{aligned} T(p_i) &= D(f_i.src, p_i.list[1]) \\ &+ \sum_{j=1}^{p_i.len-1} (D(p_i.list[j], p_i.list[j+1]) \ + \ t_p(p_i.list[j])) \\ &+ D(p_i.list[p_i.len], f_i.dst) \end{aligned} \tag{6.9}$$

We aim to reduce the end-to-end delay of the traffic traversing the network chains. On the one hand, we adequately scale in/out the VNFs in a way that the incoming flow packet rates and throughout do not exceed the sum of the maximum capacities of the existing instances. On the other hand, we shape the traffic distribution scheme by logically linking the flows and the VNF through their characteristics.

**Problem definition.** *Given the set of flows $F$, policies $P$, NFs $M$, chains $CH$, their branches $B$, their sub-branches $SB$, and delay matrix $D$. We aim to determine the needed number of instances of each VNF to ensure that all ingress flows are accommodated. Afterwards, we map the flows on the resultant chain sub-branches $SF$ based on both flow and VNF properties to reduce the total end-to-end delay at the chains.*

$$\text{Minimise} \quad \sum_{p_k \in P} T(p_k) \quad \text{Subject to:}$$

$$\forall p_k \in P, \ p_k \text{ is satisfied} \tag{C1}$$

$$\forall p_k \in P, \ \forall m_i \in p_k.list, \begin{cases} \sum_{f_i \in E(*,m_i)} f_i.pr \leq \sum_{m_j \in M_i} m_j.c_{pps}, & m_i.proc = 0 \\ \sum_{f_i \in E(*,m_i)} f_i.thp \leq \sum_{m_j \in M_i} m_j.c_{dps}, & \text{otherwise} \end{cases} \tag{C2}$$

$$\forall \ f_i \in E(*, m_i), \exists m_j \in \ M_i, \begin{cases} f_i.pr \leq m_j.c_{pps}, & m_i.proc = 0 \\ f_i.thp \leq m_j.c_{dps}, & \text{otherwise} \end{cases} \tag{C3}$$

The constraints (C1) impose that all policies should be satisfied. (C2) ensure that there should be $|M_i|$ VNFs capable of handling all the incoming flows, either for I/O bound or CPU bound NFs. (C3) highlight that in a flow-based distribution scheme, we might have individual flows that cannot be accommodated by any of the existing VNF instances. Hence, we ensure that for any flow traversing a group of VNF instances, there should be a $m_i$ that has the sufficient capacity either regarding the packet rate or throughput.

The above problem can be proven to be NP-Hard.

**Proof.** Consider a special case of the *NF Scaling and Traffic Distribution Optimisation* problem that includes a chain composed of three NFs: $nf_1, nf_2$, and $nf_3$. They are sequentially connected: $nf_1$ is directly connected to $nf_2$ and $nf_2$ is directly connected to $nf_3$. They are organised in a way that there is only $1 = 1 \times 1 \times 1$ branch in this chain to be traversed by the $n$ existing flows. In this case, flows have to enter in $nf_1$, pass through $nf_2$, and leave the chain by $nf_3$. Suppose the capacity of $nf_1$ is enough to accept all flows, but the capacities of $nf_2$ and $nf_3$ are not, which indicates that they are overloaded. In this case, a reasonable solution is to scale $nf_2$ and $nf_3$ out in at least one unit each so that they can be able to accept all the $n$ flows. In this new setup which is composed of 1 $nf_1$, 2 $nf_2$, and 2 $nf_3$, three new *sub-branches* are created, totalising $4 = 1 \times 2 \times 2$ *sub-branches* in the chain. Then, the original problem becomes to find an appropriate *sub-branch* for each of the $n$ flows that results in the overall low latency in this new setup.

Consider each flow to be an item, where its requirement (throughput or packet rate) is the item size. Thus, each VNF can be seen as a knapsack $k_j$ with limited capacity $k_j.cap$. The profit of assigning flows to each VNF is the negative of the flow delays. Then, the *NF Scaling and Traffic Distribution Optimisation* problem becomes finding a path for each flow through the VNFs that maximises the total profit. In other words, this becomes a Multiple Knapsack Problem (MKP) [81], whose decision version has already been proven to be NP-hard. Therefore, the MKP problem is reducible to our problem in polynomial time, and hence the *NF Scaling and Traffic Distribution Optimisation* problem is NP-hard.

## 6.2 *Natif*'s mechanisms

The network administrators set up the traffic steering decisions in the chains in advance, they consider the traffic attributes (e.g., the 5-tuple) and also the VNF functionality. For instance, in the FW configuration, we define what is the required action when "drop" or "allow" conditions are met. However, in particular

**Algorithm 1** *NF Instantiation()*

**Input:** $CH$, $F$, $B$, $M$
**Output:** updated $CH$

1: **for** each $ch \in CH$ **do**
2: **for** each $b \in B$ **do**
3: $F' = \{$flows traversing $b\}$
4: $f_{pr} = \sum_{f \in F'} f.pr$
5: $f_{thp} = \sum_{f \in F'} f.thp$
6: **for** each $m \in M$ on $b$ **do** ▷ in order
7: **if** $m.proc = 0$ **then**
8: $m.reqCapH = f_{pr}$
9: $m.instNb = ceil(m.c_{pps}/m.reqCapH)$
10: **else**
11: $m.reqCapP = f_{thp}$
12: $m.instNb = ceil(m.c_{dps}/m.reqCapP)$
13: **end if**
14: $f_{pr} = f_{pr} * m.gd_p$
15: $f_{thp} = f_{thp} * m.gd_d$
16: **end for**
17: CreateInstances()
18: **end for**
19: **end for**
20: Output the new structure of the chains in $CH$

cases, like for an LB, the steering is offloaded to the VNF itself which internally determines how to forward the traffic. For consistent traffic distribution, we propose to overwrite the internal steering decisions made directly by the VNF (like the LB traffic distribution since it may degrade the performance of VNFs in one of the introduced categories). Further, we impose a flow-based traffic distribution in chains so that it does not impair the logic in the stateful NFs.

### 6.2.1 Network Function instantiation

We calculate, using a network tool called *dstat* [82], the total packet rates (number of packets per second) and throughput (size of network traffic per second) of all flows traversing each branch of the chain. Then, depending on the VNF category (whether it is I/O or CPU bound), we determine how many instances needed for that NF. As we monitor the flows and we know the implemented logic of each NF, we can identify the path they need to traverse.

In Algorithm 1, lines 1 and 2 show the scope of the instantiation process, which is all the existing chains. On each branch, we sum the flows (line 3), then, we determine the required capacity of each VNF (lines 8 and 11) and the number of needed instances (lines 9 and 12) using the ceiling mathematical function that

**Algorithm 2** *Flow Distribution()*

**Input:** $F$, $M$
**Output:** New mapping of the flows on the existing paths

1: $\hat{F} = \emptyset$ ▷ structure for mapped flows
2: **for** each $m \in M$ **do**
3: $F' = \{$flows traversing $m$ in order$\}$
4: l = length($F'$)
5: $I = \{$list of instances of $m\}$
6: **while** $F' \neq \emptyset$ **do**
7: **if** $m.proc = 0$ **then**
8: **if** $m.c_{pps}/(f_i.pr \times l) > neg\_thsd\_h$ **then**
9: **if** $m.cap_p/(f_i.thp \times l) > neg\_thsd\_p$ **then**
10: $f_i \leftarrow \arg\max_{f \in F'} f.pr$
11: $m_j \leftarrow \arg\max_{m \in I} m.c_{pps}$
12: **else**
13: $f_i \leftarrow \arg\max_{f \in F'} f.thp$
14: $m_j \leftarrow \arg\max_{m \in I} m.c_{dps}$
15: **end if**
16: **else**
17: $f_i \leftarrow \arg\max_{f \in F'} f.pr$
18: $m_j \leftarrow \arg\max_{m \in I} m.c\_cps$
19: **end if**
20: UpdateNFCapacity($m_j$)
21: **else**
22: **if** $m.c_{dps}/(f_i.thp \times l) > neg\_thsd\_p$ **then**
23: **if** $m.c_pps/(f_i.pr \times l) > neg\_thsd\_h$ **then**
24: $f_i \leftarrow \arg\max_{f \in F'} f.thp$
25: $m_j \leftarrow \arg\max_{m \in I} m.c_{dps}$
26: **else**
27: $f_i \leftarrow \arg\max_{f \in F'} f.pr$
28: $m_j \leftarrow \arg\max_{m \in I} m.c_{pps}$
29: **end if**
30: **else**
31: $f_i \leftarrow \arg\max_{f \in F'} f.thp$
32: $m_j \leftarrow \arg\max_{m \in I} m.c_{dps}$
33: **end if**
34: UpdateNFCapacity($m_j$)
35: **end if**
36: $\hat{F}(f_i) = \hat{F}(f_i) + m_j$ ▷ $\hat{F}(f_i)$ is a list
37: $F' = F' \setminus \{f_i\}$
38: **end while**
39: **end for**
40: Output the mapped flows $\hat{F}$

rounds a number up to the nearest integer. Following that, to move to the next NF, we amend the flow characteristics according to the gain/drop factor (lines 14 and 15). Finally, we call *CreateInstances()*, line 17, to create the needed instances.

### 6.2.2 Traffic distribution

After the vertical scaling decision and implementation, the chain will have new sub-branches, and therefore the flows need to be mapped again onto the new paths. The problem consists of having a set of flows traversing different VNF sequences that are including similar instances at each level. We aim to accommodate all the flows in a flow-based mode with consideration of the VNF characteristics. So, for $n$ flows traversing $m$ VNF instances, the distribution module should provide which flow should traverse which instance and ensure that no flow is left or a VNF is overloaded.

In the mapping strategy, to reduce the likelihood of having no accommodated flows, we start by assigning the most massive flows (either in packet rate or throughput) to the VNF instances having the *considerable* maximum remaining capacity. Following that, the flow status (mapped or not) and the VNF remaining capacity will be updated, and the process will resume until the completion of mapping all the flows on all the instances of all the branches. In Algorithm 2, we adopt a different approach to the instantiation module, we mainly look at the VNFs and their associated flows, and we match between them as described in lines 6 to 37. For a given NF, the mapping finishes when there is no flow left untreated. The output of the algorithm is a data structure in which the distribution module describes the path of each flow. However, in some cases, only relying on the VNF category and the flow attributes can lead to poor mapping decisions. For instance, if at a CPU bound NF, we are receiving flows with negligible throughput (relatively to the VNF capacity) but with high packet rate, considering only the throughput in the traffic distribution will create congestion at one of the VNF instances (the *victim* NF). Thus, we define the variable *negligibility threshold* for both VNF categories. It means if the flow size (either in packet rate or throughput) cannot influence the VNF performance, then we consider another criterion of distribution. For the same VNF above, the flow throughput does not affect the VNF performance, so we rely on the packet rate instead.

As an illustration, in Fig. 6.2, there are six flows, $a$, $b$, $c$, $d$, $e$, $f$, traversing the chain composed of two VNFs $X$ and $Y$. The first VNF $X$ is I/O bound, the second VNF $Y$ is CPU bound. So, the performance of the VNF $X$ is bounded by its capacity regarding the packet rate, for $Y$'s performance, its throughput capacity limits it. In the first step, we sum the total packet rates and throughput of all flows; then, we calculate how many instances needed for each NF. Following that, we create the required instances, and we perform the flows mapping, step by step, as explained below. In this case, none of the flows has a negligible size comparing to the VNFs capacity.

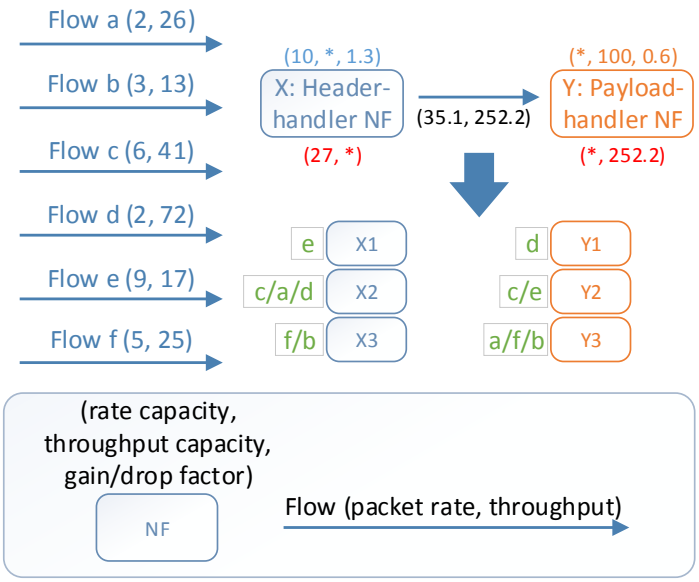



Figure 6.2: Algorithms application

1. $X1 \leftarrow e$, $X2 \leftarrow c$, $X3 \leftarrow f$. Each time we map a single flow, we update the remaining capacity of the VNF instances, and we still follow the rule *'bigger flow goes to the VNF with higher capacity'*, where the definition of *bigger* and *higher* depend on the VNF category.

2. $X1 \leftarrow \emptyset$, $X2 \leftarrow a$, $X3 \leftarrow b$.

3. $X1 \leftarrow \emptyset$, $X2 \leftarrow d$, $X3 \leftarrow \emptyset$. When moving to the second VNF in the network chain, we need to apply the gain/drop factor of the last VNF ($X$ in this case) to get the new flows attributes.

4. $Y1 \leftarrow d$, $Y2 \leftarrow c$, $Y3 \leftarrow a$.

5. $Y1 \leftarrow \emptyset$, $Y2 \leftarrow e$, $Y3 \leftarrow f$.

6. $Y1 \leftarrow \emptyset$, $Y2 \leftarrow \emptyset$, $Y3 \leftarrow b$.

### 6.2.3 Traffic prediction

Flow rates and throughput (see explanation in 6.2.1) are dynamic and mainly depend on the communicating applications. By leveraging the SDN capabilities, we can use the traffic statistics recorded at the forwarding devices to find out the flows attributes or for other purposes, e.g., infer the communicating VM groups like we achieved in [83]. We aim to use the past traffic data to train a prediction algorithm to help in characterising the future flows. For this purpose, we adopt the prediction model *ARIMA* [84] to determine the flows attributes for our algorithms. Without the output of the prediction, any new flow entering the network chain will be mapped to the set of the VNF instances in respect to the associated policy and based on its current attributes.

Our controller makes decisions after each cycle of the traffic prediction run. The quality and reliability of working data, as well as their rate of change, determine the period of the cycle.

## 6.3 Conclusions

In the chapter, we illustrated the mathematical modelling of the problem as well as the proposed solution. We have shown the cornerstones that *Natif* relies on, namely, network function instantiation, traffic distribution scheme, and traffic prediction. The next chapter will be dedicated to describe the experimental evaluation. We will measure the impact of the combined application of the three techniques above to demonstrate how the proposed solution can reduce the end to end delay of the traffic traversing network chains without sacrificing the network bandwidth.

# Chapter 7

# Experimental Evaluation

After formulating the VNF instantiation and traffic distribution problem, we propose our heuristic called *Natif* for a services chain composition aiming at reducing the end-to-end delay. We describe in this chapter the implementation of *Natif* in OpenStack based environment running on four nodes. We also illustrate the performance evaluation of our solution compared to greedy and network-aware service composition schemes.

## 7.1 System design and implementation

### 7.1.1 System architecture

We have implemented *Natif* in an OpenStack[1] cloud environment. We have deployed our modules alongside the OpenStack controller. Our source code, around 1,500 lines, is written in Python using OpenStack API. The implementation has three main blocks: *Interactor*, *Engine*, and *Orchestrator*. The first one provides tools to instantiate NFs, create network chains and perform traffic steering. The second block implements the logic of the three algorithms presented section 6.2. The last block plays the role of Orchestrator as it synchronises between the different modules as shown in Fig. 7.2.

### 7.1.2 Controller modules

#### Chain reading and creation

It allows reading a configuration file describing the network chain details and then calls the OpenStack subroutines to instantiate the VNF VMs.

[1]https://www.openstack.org/software/newton/

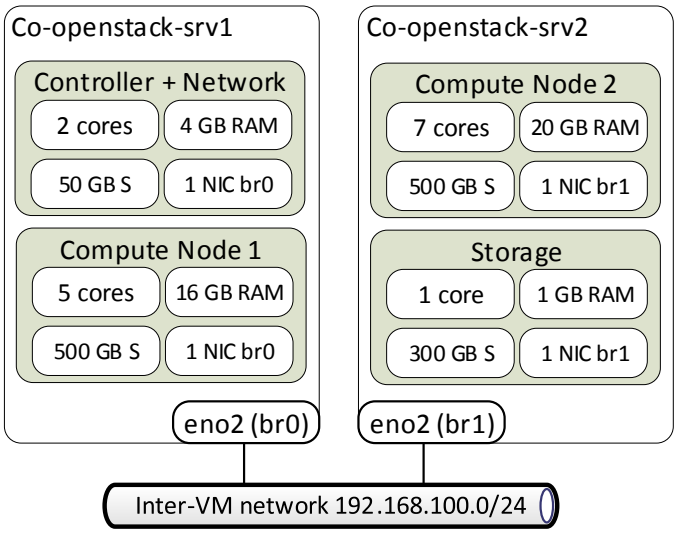



Figure 7.1: OpenStack setup

**NF instantiation**

Each VNF has deployment steps which are defined in configuration files, and the module uses this latter to instantiate the corresponding VMs.

**Flow mapping**

It determines the path of each flow and calls the SFC subroutines to implement the forwarding rules.

**NF instantiation update**

After the *NF instantiation*, the module updates the number of instances of each NF.

**Chains update**

With introducing or removing the VNF instances after *NF instantiation*, the module updates the flow mapping by implementing/deleting the corresponding routes.

**Flow attributes prediction**

For each cycle, the module invokes *Ceilometer* to retrieve data regarding the communicating VMs. It trains the *ARIMA* prediction model to the flow attributes to the *NF instantiation* and *flow mapping* algorithms.

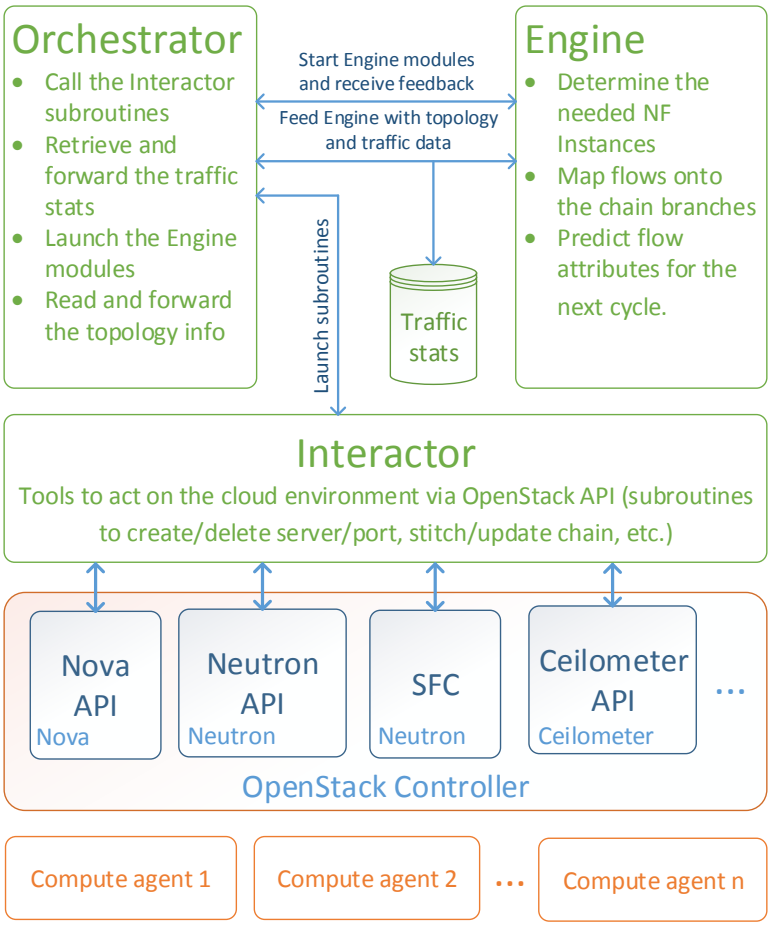



Figure 7.2: System design

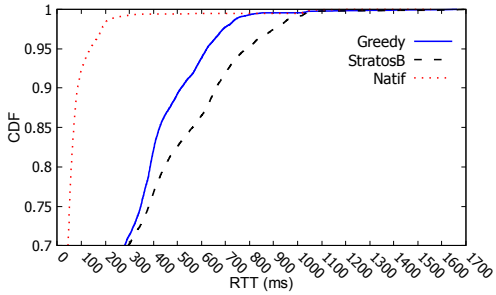
## 7.2 Experimental evaluation

### 7.2.1 Testbed experiment

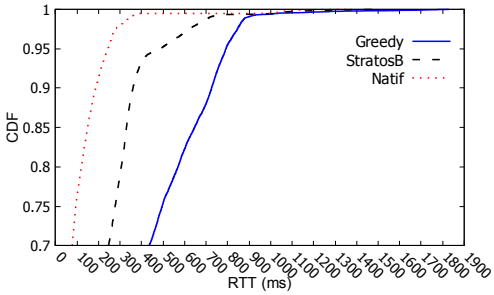
Fig. 7.1 shows the testbed setup. We use two servers with 8 cores and 32 GB RAM each. On the first one, we deploy one instance of OpenStack as a controller, and the second instance as a compute node 1. On the second server, we deploy two OpenStack instances, the first as another compute node and the second for the storage. The *SFC* extension is installed on the controller node as it controls and manages the network chains while the *flow monitor* is on the other nodes (compute and storage) to collect network stats of the NFs.

We set up two communicating networks (internal and external) with 2:1 oversubscription rate. Six source VMs are on the external network and three destinations VMs on the internal one. All of them are located on the same physical server and are separated from the VNF server where the network chains are deployed. We define three network traffic profiles. Profile 1 (*P1*): all flows have a high packet rate but low throughput. Profile 2 (*P2*): all flows have high throughput but random packet rate. Profile 3 (P3): mixed flows from *P1* and *P2*.

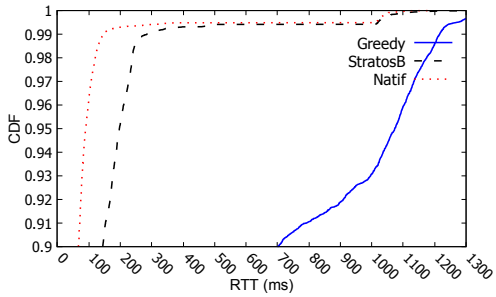
We also used two reference scenarios for the performance assessment. Greedy:




(a) Profile 1: high packet rate flows




(b) Profile 2: high throughput flows




(c) Profile 3: Mix of profiles 1 and 2

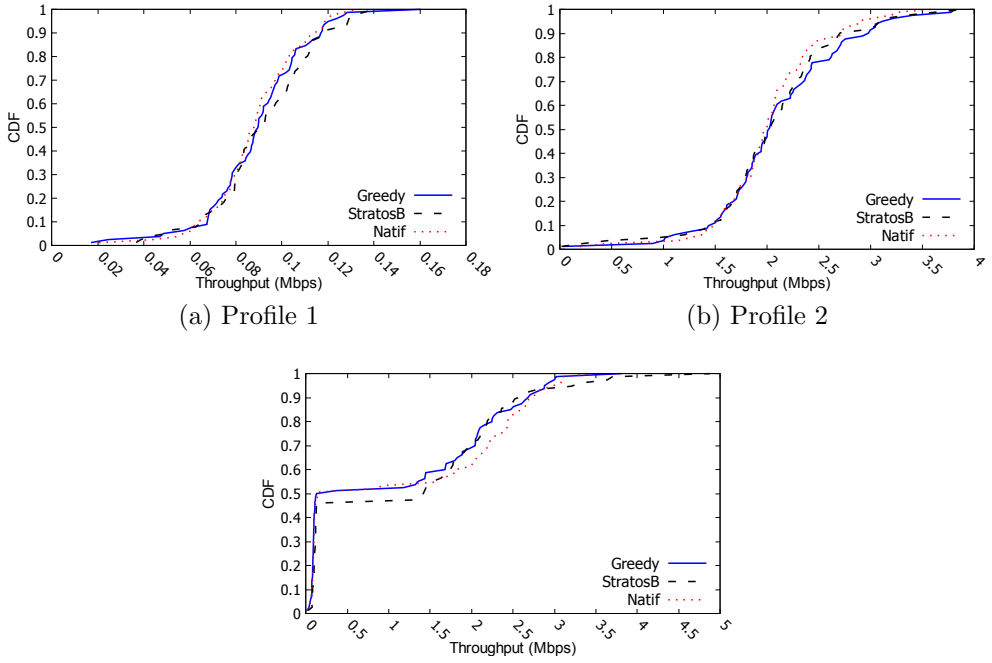
Figure 7.3: RTT (ms)

it relies on the resources allocation overprovision. We scale up all the VNFs in the chain, and we equally distribute the available cores and memory on them. Stratos-based (*StratosB*)[17]: it splits the flows between the existing instances based on their throughput. It also considers the total throughput of egress traffic to determine the number of instances of each NF. Thus, *StratosB* performs a network-aware traffic distribution to reduce the likelihood of the network congestion and link saturation.

For each scenario, we apply the three profiles, and then we measure the chain throughput and end-to-end delay of all flows.

### 7.2.2 Network performance evaluation

Fig. 7.3 shows how *Natif* has considerably reduced the RTT in the three profiles. For instance, Fig. 7.3a illustrates how, at the 90$^{\text{th}}$ percentile, *Natif* achieves around 70ms RTT comparing to 590ms and 750ms RTT, i.e., 850% and 1070% improvement compared with Greedy and *StratosB*, respectively. The next Fig. 7.3b shows the RTT for *P2*, *StratosB* outperforms Greedy, but it is still less efficient than *Natif*. For example, at the 90$^{\text{th}}$ percentile, the RTT is at 200ms, 400ms, and 700ms for *Natif*, *StratosB* and Greedy, respectively. When we simultaneously apply mixed profiles (*P3*), *Natif* remains better than *StratosB* as it reduces the RTT from 140ms to 70ms at the 90$^{\text{th}}$ percentile and from 290ms to 145ms at 99$^{\text{th}}$ percentile (i.e., an improvement of 100%). It also achieves better results compar-




(a) Profile 1

(b) Profile 2

(c) Profile 3

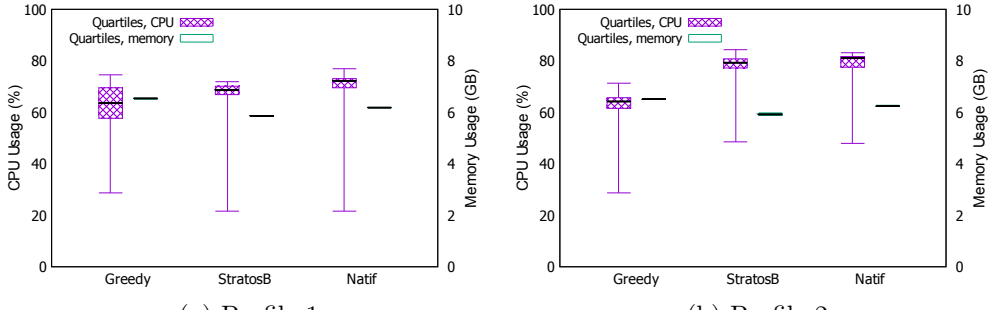
Figure 7.4: Throughput (Mbps)

ing to Greedy as it reduces the latency from 700ms to 70ms at 90$^{th}$, which means ten times better.

Fig. 7.4a and 7.4b pick out an identical behaviour of the throughput in the three approaches. However, Fig. 7.4c shows how *Natif* achieves throughput 8% better compared to *Greedy* and *StratosB*.

In Fig. 7.3a, *StratosB* and Greedy do not capture the fact that the high packet rate (even if it is incurring low throughput) can degrade the I/O bound VNFs performance, like the FW and the NAT in the studied network chain. With neglecting this characteristic in the VNF and having more expanded network chain comprising more I/O bound NFs, the performance degradation would be more drastic than the current scenario. In Fig. 7.3b, we apply high throughput traffic, so *StratosB* correctly recognises the needed VNF instances but partially succeeds in making a correct traffic distribution since it does not consider the packet rate when dealing with flows traversing an I/O bound NF. For *P3*, *StratosB* still performs well with high throughput flows but not better than *Natif* since there is still high packet flows that need to be suitably distributed, thus, the results in Fig. 7.3b. Greedy focuses on the horizontal scaling of the NFs, which does not help when the VNF has a single-threaded implementation so it will not be able to use all the allocated cores.

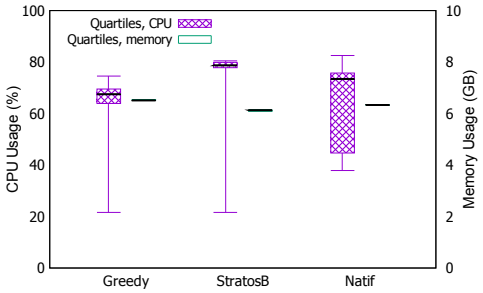
To conclude, *Natif* has reduced the end-to-end delay without scarifying the network throughput, and in some cases, it improves both metrics simultaneously, e.g., in *P3* (Profile 3 as described above).




(a) Profile 1




(b) Profile 2




(c) Profile 3

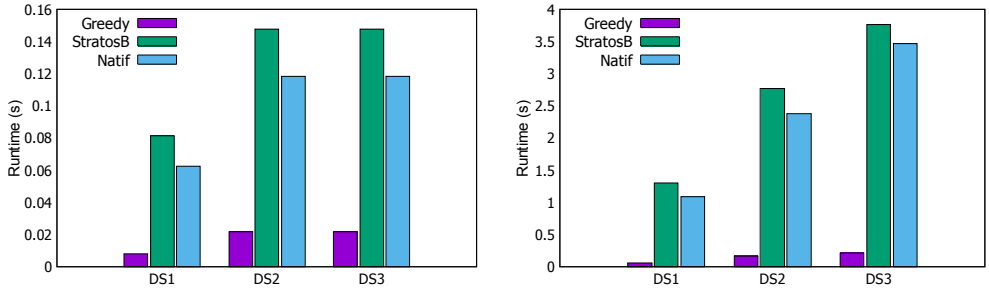
Figure 7.5: CPU and memory usage captured at the compute node where all the VNFs are running

### 7.2.3 Computational utilisation

We aim to understand how the different approaches can affect the NFs' utilisation of the computation resources. In Fig. 7.5, the compute node, where the VNFs are running, has almost the same CPU usage for *StratosB* and *Natif*. However, the usage is different when looking at Greedy. The latter leads to underuse of the available resources since it does not consider the VNF implementation. In *P3*, *Natif* consumes slightly less CPU resources than *StratosB*. In Fig. 7.5, we show also the memory consumption of the compute node. We observe the same pattern through the three profiles. Greedy has the most critical memory usage, then comes *Natif* and lastly *StratosB*. For Greedy, it always allocates more resources to the VNFs than the other methods. *Natif* maximises the usage of the underlying resources which explains the relative increase in the memory usage comparing to *StratosB*. For example, while *StratosB* fails to distribute the high packet rate flows having different throughput, *Natif* distributes traffic based on both packet rate and throughput.

### 7.2.4 Algorithm evaluation

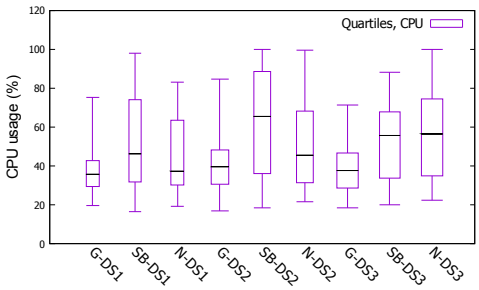
Each approach applies the *NF instantiation* and the *flow mapping* algorithms. In this section, we measure the runtime of each algorithm using different datasets and the corresponding computational usage. We define three datasets. *DS1*:




(a) NF instantiation module




(b) Flow mapping module




(c) CPU usage

Figure 7.6: Runtime and CPU usage of the three algorithms while performing VNF instantation and flow mapping. G, SB, and N refer to Greedy, StratosB, and Natif, respectively

consisting of 100 NFs, 30 chains, and 1k flows. *DS2*: 200 NFs, 60 chains, and 2k flows. *DS3*: 300 NFs, 90 chains, and 3k flows.

Fig. 7.6a shows the runtime of the *NF instantiation* algorithm. We remark that Greedy noticeably takes less time than the case for its counterpart algorithms. Also, its processing time slightly grows with the increasing data-set sizes (from 0.04s for *DS1* to nearly 0.025s for *DS2* and *DS3*). Furthermore, *Natif* outperforms *StratosB*.

For example, for *DS1*, *Natif* takes 0.06s whereas *StratosB* takes more than 0.08s. The same trends can be observed with larger data-sets.

In Fig. 7.6b, for the *flow mapping*, *Natif* still outperforms *StratosB*, and Greedy has the shortest runtime. Also, the runtime evolves linearly with larger data-sets, e.g., in *Natif*, it increases from 1s to 2.3s to 3.3s and in *StratosB* from 1.3s to 2.8s to 3.8s, for *DS1*, *DS2*, and *DS3*, respectively.

In the first algorithm, the three approaches aim to determine the number of VNF instances to accommodate the ingress traffic. In Greedy, the chain has a fixed number of instances, and the resources are statically allocated. So the time is mostly spent on stitching the chain, i.e., implementing the links between the NFs, which explains the short runtime. *StratosB* determines the number of instances solely based on the ingress traffic throughput. However, *Natif* considers two criteria, the packet rate and the throughput, which results in a different

number of VNF instances. When dealing with CPU-bound NF, *Natif* and *StratosB* have a *similar* behaviour with minor differences (Section 6.2.2). Whereas, when it comes to dealing with traffic traversing an I/O bound NF, *StratosB* still considers the throughput criterion while *Natif primarily* looks at the packet rates. As a result, for the case of I/O bound NF, when considering the throughput rather than the packet rate, we can have more VNF instances since an I/O bound VNF has less sensitivity to the throughput than a CPU-bound NF. Hence, more instances mean more time for the *NF instantiation*.

The same reasoning can still explain why *Natif flow mapping* algorithm outperforms its counterpart in *StratosB*. In case of dealing with I/O bound NF, *Natif* will output fewer instances than *StratosB*, and therefore, the flow mapping will be taking less time.

Fig. 7.6c shows the effect of the different algorithms on the CPU usage of the controller node. In *DS1* and *DS2*, *StratosB* have the most significant CPU usage, then comes *Natif*, and lastly Greedy. As highlighted above, this reflects the logic behind each approach. However, in *DS3*, *Natif* slightly needs more computational resources than *StratosB*. *Natif* runs more iterations than the other approaches, and this may have a clearer repercussion especially with larger data-sets

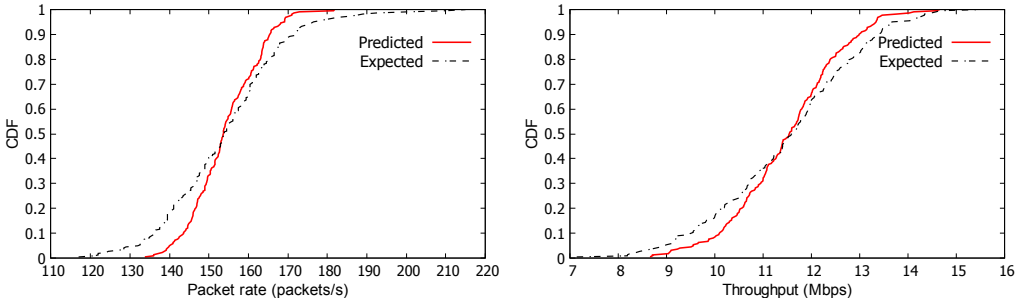
### 7.2.5 Prediction model evaluation

To evaluate *Arima* prediction model accuracy, we have used it to predict the future network traffic characteristics (packet rate and bandwidth in Mbps) traversing a firewall in the chain. This is needed to calculate the traffic rate and throughput of the incoming flows in the next cycle of *Natif* processing, i.e., VNF instantiation and traffic steering.

The parameters of the *Arima* model are defined as follows:

- p: The number of lag observations included in the model, also called the lag order.
- d: The number of times that the raw observations are differenced also called the degree of differencing.
- q: The size of the moving average window, also called the order of moving average.

In our experiment, we set p,d, and q to 5, 1, and 0, respectively. First, we have trained the *Arima* model with 900 seconds of real traffic packet rate and bandwidth. Then, we have compared between the expected and predicted traffic as illustrated in Fig. 7.7.




(a) Predicting packet rate (b) Predicting throughput (Mbps)

Figure 7.7: Arima prediction mode accuracy

Fig. 7.7a show very close behaviour of expected and predicted packet rate with a maximum error of 35 packets per second. Otherwise, the accuracy is quite high. We also use *Arima* with the same parameters to predict the traffic bandwidth. As shown in Fig. 7.7b, the maximum error is around 1.9 Mbps, except that, *Arima* has succeeded in predicting the bandwidth accurately.

## 7.3 Conclusion

NFV has facilitated the deployment and management of VNFs, but it has deepened the virtualisation overhead which particularly hurts the performance of latency-sensitive applications. To compensate this overhead, we have proposed a simple but efficient solution that leverages the knowledge on VNFs to propose a VNF qualitative categorisation and chain composition proven useful in reducing the end-to-end delay. An experimental evaluation conducted in OpenStack testbed has shown how *Natif* has reduced the latency by 188% on average in realistic and production network chains.

We aim in future work to answer to following questions. How to find out the gain/drop factor of certain VNFs such as Redundancy Eliminator since, in this case, the factor primarily depends on the packet payload (unlike NAT or proxy where the factor is known and static). Also, how we can tweak our approach to be applicable for bursty traffic where the Arima prediction algorithm would have insufficient time to be trained.

# Chapter 8

# Conclusions and Future Work

This chapter summarises the research findings of this thesis and illustrates how the achieved work has contributed to achieving the research objectives set from the outset. It also outlines the applicability and possible extension of our approach in other areas such as edge computing.

## 8.1 Summary

The thesis has illustrated the main background key concepts we have been relying on to address critical challenges in data centre networks. We have shown how SDN has significantly improved the network programmability and flexibility by decoupling the data plane from the control plane. On the other hand, NFV and SFC have presented opportunities to ease the management of the service chains and investigate further research directions, which significantly enables the application of our proposed approach. *Natif*, the main contribution of our research, leverages the knowledge of network functions regarding performance sensitivity and resource utilisation to propose a network function instantiation and traffic distribution scheme within the service chains. Both proof-of-concept and testbed experiments have proven the efficiency of *Natif* in reducing the network latency without disturbing the network throughput compared to a network-aware service chain composition solution.

## 8.2 Conclusions

The following describes the main findings of this thesis:

- Virtualisation has caused a significant performance degradation to virtual network functions compared to their hardware-based counterparts.

- In the context of NFV, NFs are bounded either by CPU or I/O resources, as the case of software applications. CPU-bound NFs are sensitive to the amount of traffic handling it, while I/O-bound NFs are mainly affected by the traffic rate in terms of packets per second. A selective approach considering both categories is essential to reflect the performance bottlenecks of diverse types of NFs making the service chains.
- Identifying the NFs underpinning implementation can largely enhance their resource utilisation.
- We have modelled the NF instantiation and traffic distribution scheme, and we have proven its NF-harness.
- We have designed and set up an OpenStack based environment to assess *Natif*'s performance. *Natif* has been able to reduce on average 188% of network latency compared to other approaches.

## 8.3 Future work

Our method presented in this thesis focuses on understanding and identifying the research challenges (virtualisation and NF performance sensitivity) and afterwards exploit the eventual knowledge to advance the state-of-the-art and design more efficient approaches in chain composition. We aim to move our expertise and acquired experience to the edge of the network, the edge computing, where the environment is still being explored. For this purpose, we are investigating two main ideas:

### 8.3.1 Application performance benchmarking on Raspberry Pi

We evaluate the following deployment approaches to determine which one is the most appropriate (in terms of reliability and flexibility) in edge environments. We intend to benchmark applications performance handling different workloads (applications can be for network functions, big data, machine learning).

- **Docker** facilitates the application deployment and running using containers. Containers allow developers to wrap their applications with all the libraries and dependencies they need.
- **Xen** is a hypervisor using a μ-kernel design, which is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS).

- **Unikernel** is a single address space machine image constructed by using library operating systems and can run directly on a hypervisor or hardware with OS interposition.

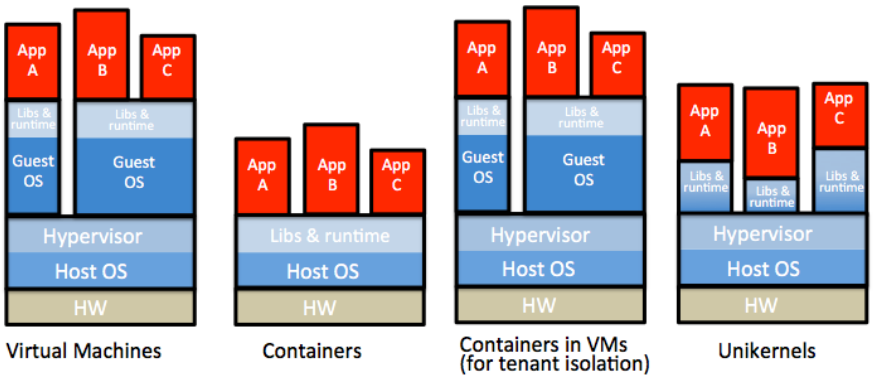



Figure 8.1: Differences between Container, Unikernel, and Virtual Machine [6]

### 8.3.2 Service Chains Cloning and Placement in the Context of Edge Computing

The study aims to overcome the inadequacy of the high resource requirements of network chains and the limited capacity of Raspberry Pi as representative of edge devices. We introduce a new definition of network chains horizontal scaling. Practically it means cloning the same network chain to allow optimised workload distribution on the set of edge devices. This is particularly interesting because of two properties characterising the edge environment.

- Abundant number of edge devices with limited capacity. So, "fragmenting" VNFs into smaller entities (with less capacities) could fit with the available resources on these devices.

- Polling is scheduled at the Things (e.g., sensors) so that the traffic processing at the edge. Therefore, it can be considered in the chain cloning and placement decisions.

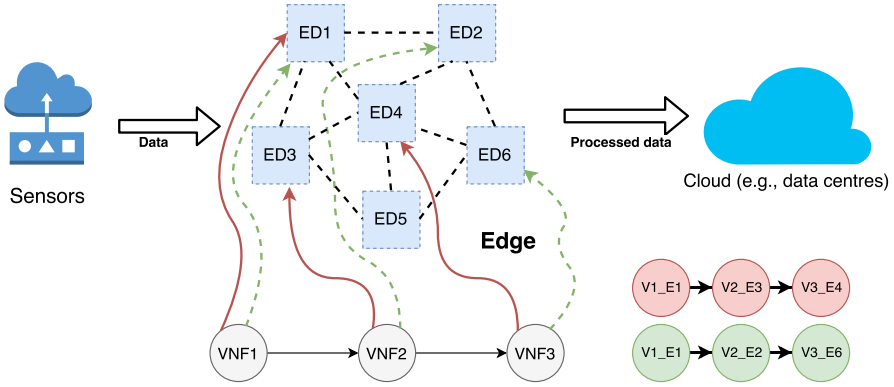



Figure 8.2: Simple scenario of chain cloning and placement

# References


[1] Service function chaining use cases in data centers 04. URL: https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-04.

[2] Wenrui Ma, Jonathan Beltran, Zhenglin Pan, Deng Pan, and Niki Pissinou. Sdn-based traffic aware placement of nfv middleboxes. *IEEE Transactions on Network and Service Management*, 14(3):528–542, 2017.

[3] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.

[4] What is network service chaining? definition. URL: https://www.sdxcentral.com/sdn/network-virtualization/definitions/what-is-network-service-chaining/.

[5] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad. A survey on service function chaining. *Journal of Network and Computer Applications*, 75:138–155, 2016.

[6] Unikernels meet nfv. URL: https://www.ericsson.com/research-blog/unikernels-meet-nfv/. Accessed: 2018-01-29.

[7] Middleboxes: Taxonomy and issues status. URL: https://www.ietf.org/rfc/rfc3234.txt.

[8] Ramana Rao Kompella, Kirill Levchenko, Alex C Snoeren, and George Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 255–266. ACM, 2009.

[9] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[10] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, et al. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, 2011.

[11] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, 2012.

[12] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: predictable message latency in the cloud. *ACM SIGCOMM Computer Communication Review*, 45(4):435–448, 2015.

[13] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, 2015.

[14] Jiao Zhang, Fengyuan Ren, and Chuang Lin. Survey on transport control in data center networks. *IEEE Network*, 27(4):22–26, 2013.

[15] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, and Timothy Wood. Performance management challenges for virtual network functions. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 20–23. IEEE, 2016.

[16] Justine Sherry, Sylvia Ratnasamy, and Justine Sherry At. A survey of enterprise middlebox deployments. 2012.

[17] Aaron Gember, Robert Grandl, Ashok Anand, Theophilus Benson, and Aditya Akella. Stratos: Virtual middleboxes as first-class entities. *UW-Madison TR1771*, page 15, 2012.

[18] S Kumar, M Tufail, S Majee, C Captari, and S Homma. Service function chaining use cases in data centers. *IETF SFC WG*, 2015.

[19] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46. ACM, 2010.

[20] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI*, volume 13, pages 329–342, 2013.

[21] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 7. ACM, 2013.

[22] Anish Hirwe and Kotaro Kataoka. Lightchain: A lightweight optimisation of vnf placement for service chaining in nfv. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 33–37. IEEE, 2016.

[23] Sahel Sahhaf, Wouter Tavernier, Didier Colle, and Mario Pickavet. Network service chaining with efficient network function mapping based on service decompositions. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–5. IEEE, 2015.

[24] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao. Plan: Joint policy- and network-aware vm management for cloud data centers. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1163–1175, April 2017.

[25] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research*, pages 143–149. ACM, 2017.

[26] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *RPE Report*, pages 1–42, 2005.

[27] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[28] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[29] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[30] Paul Quinn and Thomas Nadeau. Problem Statement for Service Function Chaining. RFC 7498, April 2015.

[31] Service function chaining use cases in data centers 06. URL: https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06.

[32] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 479–491. ACM, 2015.

[33] Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. Echo: Recreating network traffic maps for datacenters with tens of thousands of servers. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 14–24. IEEE, 2012.

[34] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review*, 45(4):139–152, 2015.

[35] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[36] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[37] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[38] David Erickson. *Using network knowledge to improve workload performance in virtualized data centers*. PhD thesis, Citeseer, 2013.

[39] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[40] Dennis M Volpano, Xin Sun, and Geoffrey G Xie. Towards systematic detection and resolution of network control conflicts. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 67–72. ACM, 2014.

[41] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. *ACM SIGCOMM Computer Communication Review*, 45(4):43–56, 2015.

[42] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 539–550. ACM, 2014.

[43] Liting Hu, Karsten Schwan, Ajay Gulati, Junjie Zhang, and Chengwei Wang. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *Proceedings of the 9th international conference on Autonomic computing*, pages 3–12. ACM, 2012.

[44] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 467–478. ACM, 2014.

[45] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 66–70. IEEE, 2011.

[46] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and TV Lakshman. Application-aware data plane processing in sdn. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2014.

[47] Zafar Ayyub Qazi, Jeongkeun Lee, Tao Jin, Gowtham Bellala, Manfred Arndt, and Guevara Noubir. Application-awareness in sdn. *ACM SIGCOMM computer communication review*, 43(4):487–488, 2013.

[48] Michael Jarschel, Florian Wamser, Thomas Hohn, Thomas Zinner, and Phuoc Tran-Gia. Sdn-based application-aware networking on the example of youtube video streaming. In *2013 Second European Workshop on Software Defined Networks*, pages 87–92. IEEE, 2013.

[49] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 307–318. ACM, 2014.

[50] Keqiang He, Weite Qin, Qiwei Zhang, Wenfei Wu, Junjie Yang, Tian Pan, Chengchen Hu, Jiao Zhang, Brent Stephens, Aditya Akella, et al. Low latency software rate limiters for cloud networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 78–84. ACM, 2017.

[51] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014.

[52] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review*, 45(4):465–478, 2015.

[53] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical report, Technical Report, 2013.

[54] Lin Cui, Richard Cziva, Fung Po Tso, and Dimitrios P Pezaros. Synergistic policy and virtual machine consolidation in cloud data centers. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[55] Hendrik Moens and Filip De Turck. Vnf-p: A model for efficient placement of virtualized network functions. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 418–423. IEEE, 2014.

[56] Sevil Mehraghdam, Matthias Keller, and Holger Karl. Specifying and placing chains of virtual network functions. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 7–13. IEEE, 2014.

[57] Stuart Clayman, Elisa Maini, Alex Galis, Antonio Manzalini, and Nicola Mazzocca. The dynamic placement of virtual network functions. In *2014*

*IEEE network operations and management symposium (NOMS)*, pages 1–9. IEEE, 2014.

[58] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 49–54. ACM, 2015.

[59] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 7–12. ACM, 2012.

[60] Ali Mohammadkhan, Sheida Ghapani, Guyue Liu, Wei Zhang, KK Ramakrishnan, and Timothy Wood. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*, pages 1–6. IEEE, 2015.

[61] Pengfei Duan, Qing Li, Yong Jiang, and Shu-Tao Xia. Toward latency-aware dynamic middlebox scheduling. In *2015 24th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2015.

[62] Fangxin Wang, Ruilin Ling, Jing Zhu, and Dan Li. Bandwidth guaranteed virtual network function placement and scaling in datacenter networks. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2015.

[63] Jiaqiang Liu, Yong Li, Ying Zhang, Li Su, and Depeng Jin. Improve service chaining performance with optimized middlebox placement.

[64] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 98–106. IEEE, 2015.

[65] Xin Li and Chen Qian. The virtual network function placement problem. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 69–70. IEEE, 2015.

[66] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM computer communication review*, volume 43, pages 27–38. ACM, 2013.

[67] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 511–524. ACM, 2016.

[68] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[69] Michael Till Beck and Juan Felipe Botero. Coordinated allocation of service function chains. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

[70] Lianjie Cao, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. Nfv-vital: A framework for characterizing the performance of virtual network functions. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 93–99. IEEE, 2015.

[71] Rashid Mijumbi, Sidhant Hasija, Steven Davy, Alan Davy, Brendan Jennings, and Raouf Boutaba. Topology-aware prediction of virtual network function resource requirements. *IEEE Transactions on Network and Service Management*, 14(1):106–120, 2017.

[72] Ping. URL: https://linux.die.net/man/8/ping. Accessed: 2018-09-29.

[73] Hping3. URL: https://tools.kali.org/information-gathering/hping3. Accessed: 2018-09-29.

[74] pfsense. URL: https://www.pfsense.org/about-pfsense/. Accessed: 2018-09-29.

[75] ovs. URL: https://www.openvswitch.org/. Accessed: 2018-09-29.

[76] Snort. URL: https://www.snort.org/faq/what-is-snort. Accessed: 2018-09-29.

[77] Suricata. URL: https://suricata-ids.org/. Accessed: 2018-09-29.

[78] Major components of openstack. URL: https://waqarafridi.wordpress.com/2014/10/13/major-components-of-openstack/.

[79] Introduction to openstack. URL: https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html.

[80] Poisson process. URL: https://www.randomservices.org/random/poisson/index.html.

[81] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Other knapsack problems. In *Knapsack Problems*, pages 389–424. Springer, 2004.

[82] dstat. URL: https://linux.die.net/man/1/dstat.

[83] Wajdi Hajji, Fung Po Tso, Lin Cui, and Dimitrios P Pezaros. Experimental evaluation of sdn-controlled, joint consolidation of policies and virtual machines. In *Computers and Communications (ISCC), 2017 IEEE Symposium on*, pages 1338–1343. IEEE, 2017.

[84] H Zare Moayedi and MA Masnadi-Shirazi. Arima model for network traffic prediction and anomaly detection. In *Information Technology, 2008. ITSim 2008. International Symposium on*, volume 4, pages 1–6. IEEE, 2008.

[85] Fung Po Tso, David R White, Simon Jouet, Jeremy Singer, and Dimitrios P Pezaros. The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In *Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on*, pages 108–112. IEEE, 2013.

[86] David Mosberger and Tai Jin. httperfa tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[87] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

# Appendix A

# Understanding the Performance of Low Power Raspberry Pi Cloud for Big Data

We have extended our original project in [85] and constructed a cloud of 200 networked Raspberry Pi 2 boards for US$ 9,000. Such systems are highly portable, running from a single AC mains socket, and capable of being carried in a luggage.

We have carried out an extensive set of experiments with representative real-life workloads in order to understand the performance of such system in big data analytics. In summary, the contribution of this work is as follows:

- We designed and conducted a set of experiments to test the performance of a single node and a cluster of 12 Raspberry Pi 2 boards with realistic network and CPU bound workload in both native and virtualised environments.
- We have found that overhead for CPU-bound workload in virtualised environment is significant, giving up to 67.2% performance impairment.
- We have found that the performance of running big data analytic in virtualised environment comparable to native counterpart, albeit noticeable but trivial overhead for CPU, memory and energy.

## A.1 Experiment Setup

We describe in detail our testbed, methodology and performance metrics used to evaluate different combinations of tests in this section.

In an edge cloud we anticipate two distinctive environments—either a native environment for high performance or a virtualised environment for high elasticity. Therefore, we have tested the performance of single nodes and clusters in both

environments. In all experiments we either use a single node Raspberry Pi 2 Model B, which has a 900 MHz quad-core ARM Cortex-A7 CPU, 1 G RAM, and a 100 Mbps Ethernet connection, or a cluster of 12 nodes. For their virtualised counterparts, we have configured the node(s) with Docker, a lightweight Linux Container virtualisation, on each Raspberry Pi with Spark and HDFS running atop. We have chosen Spark because it has become one of the most popular big data analytics tools. We selected Docker not only because it is low-overhead OS level virtualisation but also the full virtualisation has not been fully supported by Raspberry Pi 2's hardware. The operating system (OS) installed on the Raspberry Pis is Raspbian (https://www.raspbian.org/).

### A.1.1 Single Node Experiments

In this set of experiments, we attempt to find the baseline performance with and without virtualisation for a single Raspberry Pi 2 Model B board. The experiments include using a client, which has an Intel i7-3770 3.4 GHz quad-core CPU, 16 GB RAM and 1 Gbp/s Ethernet, sending various workload to server, a Raspberry Pi node, using *httperf* [86]. The client used is remarkably more powerful than the server for ensuring that performance will only be limited by server's bottleneck. The server runs Apache web server to process web requests from client. The client is instructed to generate a large number of Web (HTTP) requests for pulling web documents of size 1 KB, 4 KB, 10 KB, 50 KB, 70 KB and 100 KB respectively from servers using *httperf*. These workload sizes are chosen because traffic in cloud data centre is comprised of 99% small mice flows and 1% large flows [87]. For each specific workload size, the client starts from sending a very small number of requests per second to the server initially, and gradually increases the number of requests per second by 100 until the server cannot accommodate any additional requests. This means that the server has reached its full capacity.

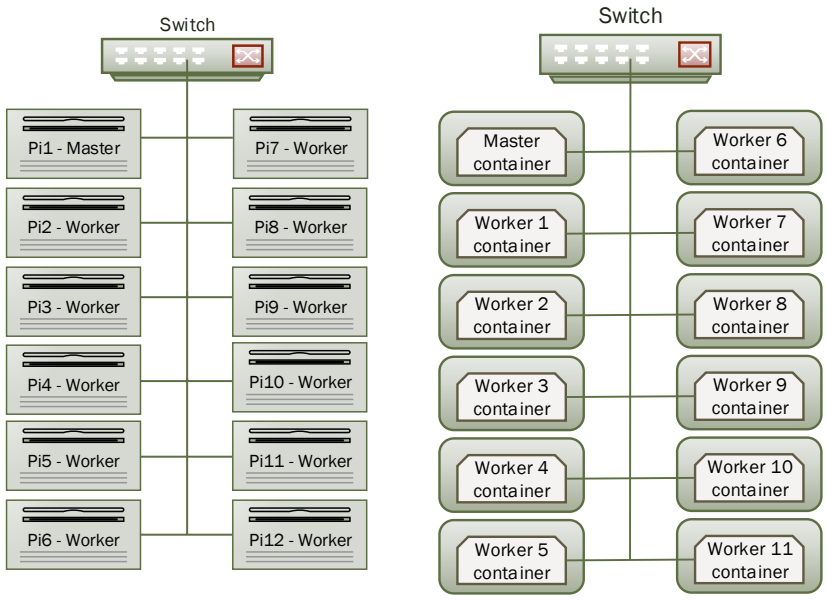
### A.1.2 Cluster Experiments

We have conducted all experiments on a low-power compute cluster consist of 12 Raspberry Pi 2 Model B. All Raspberry Pis are interconnected with a 16-Port Gbp/s switch. Alongside with system performance metrics, we are equally interested in energy consumption of the whole cluster when experiment is underway. We used MAGEEC (http://mageec.org/wiki/Workshop) ARM Cortex M4-based STM32F4DISCOVERY board to measure energy consumption of individual Raspberry Pi throughout experiments. This board was designed by the University of Bristol for high frequency measurement of energy usage.

Also on each node, we installed Spark 1.4.0 and Hadoop 2.6.4 for its HDFS.

We configured node 1, *i.e.*, Pi 1, as a master for Hadoop and Spark, and others, *i.e.*, Pi 2–12, as workers.

For Spark, each worker was allocating 768 MB RAM and all 4 CPU cores. For HDFS, we set the number of replica to 11 so that data are replicated on each worker node. This set-up was not only considered for high availability but also to avoid high network traffic between nodes as we predict that Raspberry Pi has a hardware limitation on the network interface speed. Figure A.1a shows the cluster design.




(a) (b)

Figure A.1: Cluster Layout. (**a**) Native set-up; (**b**) Virtualised set-up.

In the second phase of the experiment, we installed Docker and created a Docker container on each node of the cluster. Docker container hosts both Spark 1.4.0 and Hadoop 2.6.4 with the same setup as in the native environment. So the container is considered as a Virtual Machine running on the Raspberry Pi. We have established a network connection between the 12 containers and have made them able to communicate between each other. Figure A.1b illustrates this set-up.

In both native and virtualised environments, we have run both *Wordcount* and *Sort* jobs on our low-power cluster with job sizes varying from 1 GB to 4 GB and to 6 GB, representing small, medium and large job sizes respectively. The large job size was set to 6 GB because we have found that job size greater than this will cause Docker daemon forcibly killed by the OS because the CPU is significantly overloaded with the process. Also in all experiments we left the system idle for 20 s and the experiments started at the 21-st s.

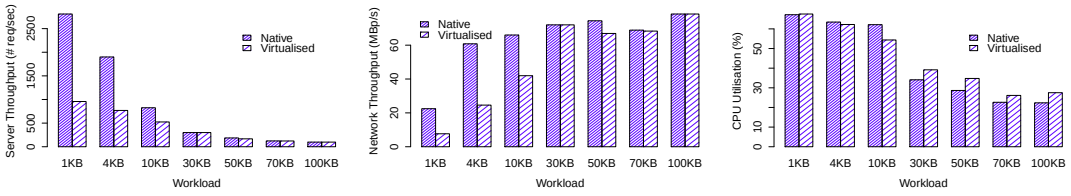
In all experiments, we have measured and collected the following metrics to examine the performance:

- Execution time: the time taken by each job running different workloads.
- Network throughput: the transmission and reception rates in each node of the cluster.
- CPU utilisation: the CPU usage in each cluster node.
- Energy consumption: energy consumed by a Raspberry Pi worker node (chosen randomly).

## A.2 Experiment Results

### A.2.1 Single Node Performance

Our test results for single node performance are shown in Figure A.2. We first examine the results for native environment. Obviously, Figure A.2a shows that the average number of network requests served by the server decreases from 2809 req/s to 98 req/s for 1 KB and 100 KB workloads respectively. In the meantime, their corresponding network throughput, as shown in Figure A.2b and CPU utilisation, as shown in Figure A.2c exhibit monotonically increasing and decreasing patterns respectively, but with flatter tails. The average network throughput for 1 KB and 100 KB workloads are 22.5 Mbp/s and 78.4 Mbp/s respectively, whereas CPU utilisation for 1 KB and 100 KB workloads are 67.2% and 22.3% respectively. These observations demonstrate that small-sized workloads such as 1 KB and large-sized workloads such as 100 KB are CPU and network bounded respectively.




(a) (b) (c)

Figure A.2: Single server performance. (**a**) Server throughput; (**b**) Network throughput; (**c**) CPU utilisation.

Next we examine the results for virtualised environment. At first glance we can clearly observe that all results for virtualised environment exhibit identical patterns as native environment. However, our performance has pinpointed significant virtualisation overhead, particularly for small workloads. Figure A.2a shows

that server throughput for 1 KB workload is profoundly impaired by 65.9%, dropping from 2,809 req/s to 957.5 req/s, leading to significant degradation in network throughput (Figure A.2b) while the CPU utilisation remains equally high as native counterpart. Similarly the impairment for 4 KB and 10 KB workloads are 59.6% and 36.4% respectively. Nevertheless, the performance for large workloads including 30 KB, 50 KB, 70 KB and 100 KB, in terms of server and network throughput, are on par with their native counterparts. In comparison the CPU utilisation for these workloads are only 12%–23%, representing fractional but significant overhead.

The remarkable overhead observed for the small-sized workloads has inspired us to investigate this issue further. When Docker is installed, a software-based bridged network, by which the Docker daemon connects containers to this network by default, is automatically created. Therefore, when workload is small not only the hardware network interface frequently interrupts CPU for packet delivery but also the software bridge triggers similar amount of interrupts for container under test. On the contrary, when workload is large, fewer hardware and software interruptions arise from both physical and virtual network interface.

### A.2.2 Spark and HDFS in the Native Environment

We first present Spark's performance in the native environment. Table A.1 shows the total execution time for 1 GB, 4 GB and 6 GB jobs. We observed that job completion time varies with actual job sizes. For instance, for *WordCount*, it increases slightly from 60.2 s for 1 GB job by 9.3% to 65.8 s for 4 GB job but increases substantially by 82.4% to 109.8 s for 6 GB job. Similar trend is observed in *Sort*, it takes 122.4 s to complete 1 GB job, then 129.7 s and 224.8 s, or 5.96% and 83.7% longer, for 4 GB and 6 GB files respectively. Comparing job completion time between *WordCount* and *Sort*, it is apparent that *Sort* is more CPU demanding because time taken by *Sort* job is almost usually double of what is consumed by *WordCount*. This is because in *Sort*, words need to be counted and then sorted, whereas in *WordCount* words need only to be counted.

Table A.1: Execution times for WordCount and Sort jobs in the Native Environment.

| File Size | "Native" WordCount | "Native" Sort |
|---|---|---|
| 1 GB | 60.2 s | 122.4 s |
| 4 GB | 65.8 s | 129.7 s |
| 6 GB | 109.8 s | 224.8 s |

To explain this non-linear increase in completion time between 4 GB and 6 GB jobs, we have investigated further and found that *Sort* for 4 GB job requires 32 tasks whilst 6 GB file needs 46. Given that there are 44 cores available in the cluster, there is sufficient computation capacity for accommodating 32 task concurrently. However, in the case when 45 or more tasks are spawn, all available cores are used, as demonstrated in Figure A.3c, and the remaining tasks will have to wait for CPU time. Worse still, if they depend on some specific tasks, they will have to wait until their completion although free CPU time will arise when some non-dependent tasks finish early. On the other hand, Spark is memory hungry whilst Raspberry Pi's RAM is sparse. As evidenced by Figure A.3c, memory has been fully utilised at most of the time throughout experiments. This implies that there may be constant memory swapping that could further lengthen the completion time. In *WordCount*, there are 15 tasks for 4 GB file versus 44 for 6 GB file, in the former case there are enough CPU resources to run all tasks whereas in the latter all CPU cores are dedicated to run the job, this can be observed in Figure A.3c where CPU usage is at 100% over data processing time whilst it is at nearly 80% for 4 GB file in Figure A.3b.

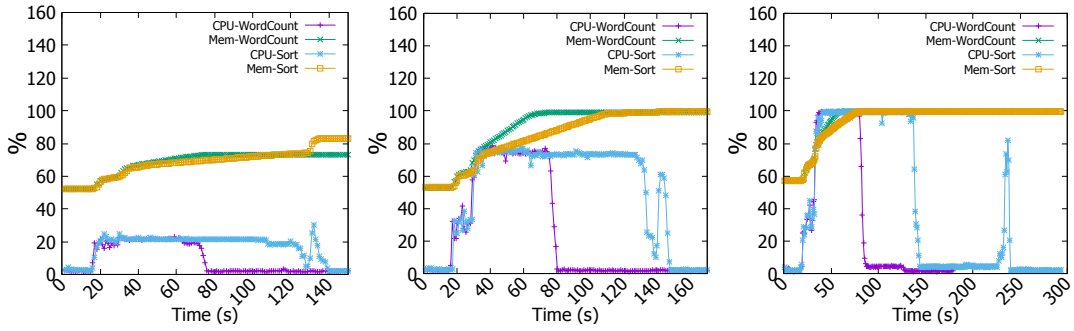



Figure A.3: CPU and memory usage. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

Next, we describe the CPU, memory and network usage performance results. In *WordCount* of 1 GB job, in Figure A.3a memory consumption increases to about 75% and remains steady till the end of the operation. For CPU utilisation, we can see that it rises from nearly 1% (idle) to nearly 20% (busy) and remains unchanged all over the computation process. For network throughput, Figure A.4a shows that there is no significant traffic activity, at the beginning of the job, data are received by workers at the rate of 40 kb/s, and this is the client (namenode) request message for workers to start computing. For files of 4 GB and 6 GB, we noted the same behaviour but the increase in CPU and memory usage is more prominent. For instance, in Figure A.3b for 4 GB file, memory usage increases gradually from 50% to 100% in about 70 s and CPU goes up from nearly 1% to

30% in the *tasks submission* stage and then sharply reaches 80% at the second 40 for the *count* stage as indicated in the log files.

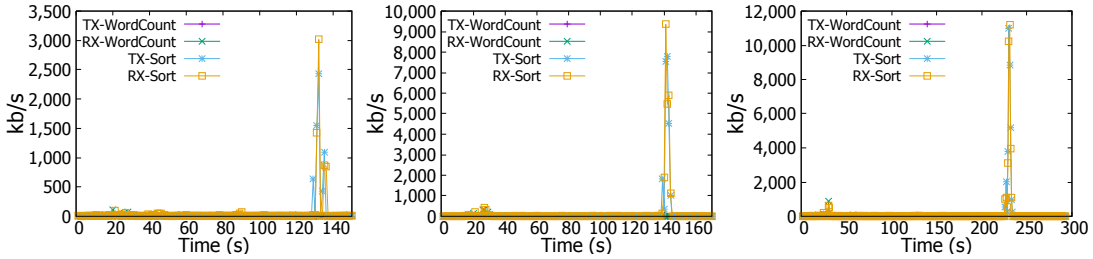



Figure A.4: Network transmission (TX) and reception (RX) rates. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

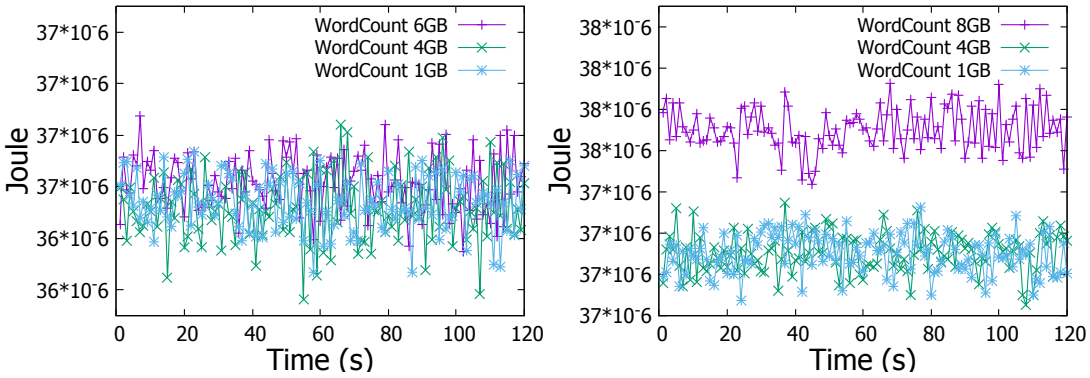
As reflected by Figure A.3c the increase is sharper for the 6 GB file where both memory and CPU reach 100%. In the 6 GB file, as explained above, since there are more tasks (46 tasks) than available CPU cores (44 cores), the CPU and memory are exhaustively used for an extended period of time. Moreover, we observe the same two stages as in the 4 GB file.

In *Sort*, CPU and network usage patterns are different from those observed in *WordCount* job. For example, in Figure A.3a for the 1 GB job, CPU usage increases to the same level as *WordCount* job for the same file size, and it remains steady throughout the experiment, but at the end of the job CPU decreases dramatically to a very low level and then suddenly reaches a peak. When analysing log files, we have found an explanation for these changes. In the beginning, *tasks submission* stage takes a few seconds to complete, this is happening also in *WordCount*, it explains both CPU and memory increase to 30% and 60% respectively. Afterwards, *map* stage starts and consumes most of the time taken by the job, lastly the *shuffling* process causes the peak witnessed by CPU usage.

In addition, *Sort* is accompanied with a peak in the network transmission and reception rates where they reach nearly 3.2 Mbps as shown in Figure A.4a. Same changes have been witnessed for 4 GB and 6 GB files but with quantitative differences. For instance, as illustrated in Figure A.4b,c network transmission and reception rates reach at the end of the *Sort* job 9.6 Mbps and nearly 11.2 Mbps for 4 GB and 6 GB files respectively. CPU and memory usages increase as well to nearly 80% and 100% for 4 GB file and to 100% and 100% for 6 GB file respectively as reflected in Figure A.3b,c. These changes are explained above by the fact that *Sort* job witnesses three phases; *task submission*, *map*, and *shuffling*. In the shuffling stage, a high network activity is noticed at the end of *Sort* job (e.g., Figure A.3a at 130 s, Figure A.3b at 140 s, and Figure A.3c at 235 s). Furthermore, outputs coming from workers need to be consolidated to have the

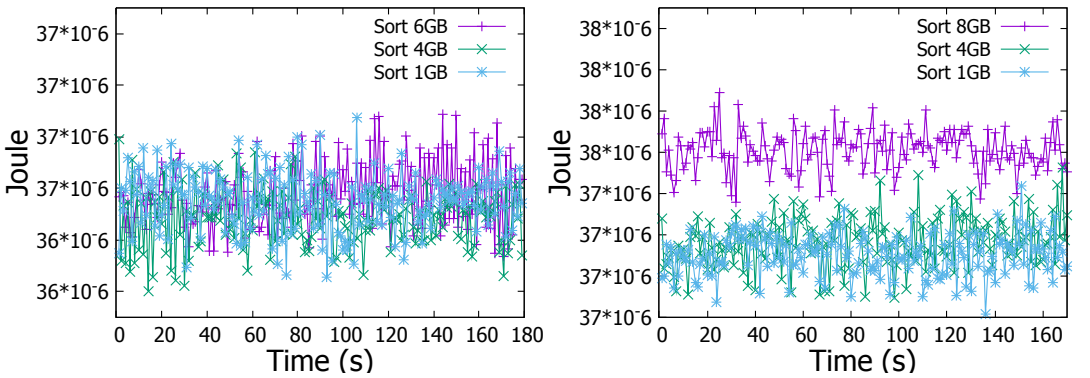
final result, this is achieved in the *reduce* stage (combining results of workers) and it causes the high CPU and memory usage.

Regarding the energy consumption, through Figures A.5a and A.6a we can obviously observe that actual energy consumption depends on the job sizes. It is slightly higher for 6 GB files than for 1 GB and 4 GB files in both *WordCount* and *Sort* jobs. To confirm this observation, we run *WordCount* and *Sort* on file of 8 GB, even with some task failures on some Raspberry Pis, we noticed the behaviour more clearly as shown in Figures A.5b and A.6b. Therefore, workload affects the energy consumption, the more intensive the workload is, the more important is the energy consumption by the Raspberry Pi device.




(a) (b)

Figure A.5: Energy measurement in a Raspberry Pi Worker node in WordCount job. (**a**) WordCount Job (1-4-6 GB files); (**b**) WordCount Job (1-4-8 GB files).




(a) (b)

Figure A.6: Energy measurement in a Raspberry Pi Worker node in Sort job. (**a**) Sort job (1-4-6 GB files); (**b**) Sort job (1-4-8 GB files).

### A.2.3 Spark and HDFS in Docker-Based Virtualised Environment

In the second phase of our experiments, we present results from virtualised environment, followed by comparing and contrasting the results with that of native ones.

We first have a look at the job completion time as shown in Table A.2. At the first glance, we can clearly see that job completion times for 1 GB and 4 GB exhibit fractional difference, smaller than 3%, between native and virtualised platforms for both *WordCount* and *Sort*.

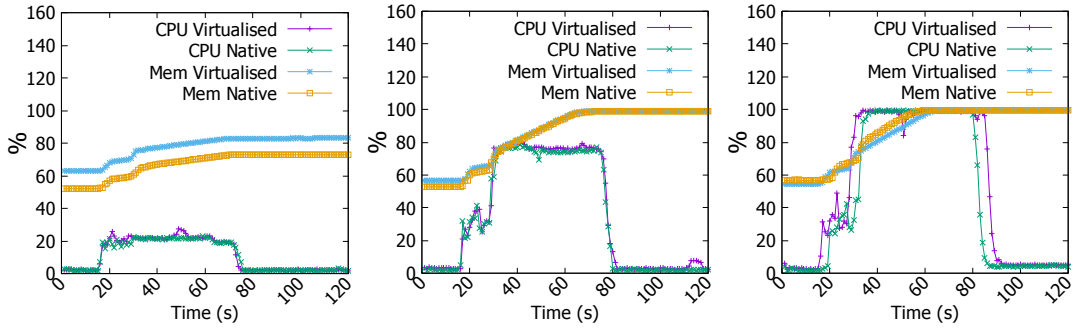
Table A.2: Execution times for WordCount and Sort jobs in Virtualised Environment.

| File Size | WordCount in Docker | Sort in Docker |
|---|---|---|
| 1 GB | 58.2 s | 121.1 s |
| 4 GB | 64.7 s | 132.2 s |
| 6 GB | 116.5 s | 236.5 s |

However, in *WordCount* of 6 GB file, execution with Docker clearly takes more time than the case without it, at 109.8 s and 116.5 s respectively, an increase of nearly 6.1%. Similarly, *Sort* on the 6 GB file takes more time in Docker than in the native environment, an increase from 224.8 s to 236.5 s, representing 5.2% longer completion time.

#### Virtualisation Impact on CPU and Memory Usage

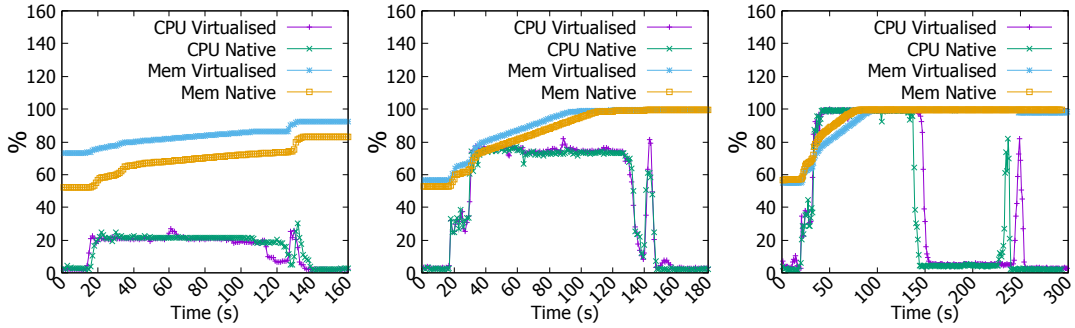
Figure A.7a shows that CPU usage, in 1 GB file *WordCount* job, has same behaviour in both native and virtualised environments but with a few irregularities where Docker is running (at 20-th and 50-th s). Memory consumption is higher in virtualised platform as Docker daemon requires already memory resources to run its processes. In *WordCount* of 4 GB file, CPU and memory usages have the same patterns in both environments (Figure A.7b). Whereas, in *WordCount* of 6 GB file, we have noticed remarkable difference in the CPU usage, Figure A.7c shows that it is more important and extended in the virtualised set-up.




(a) (b) (c)

Figure A.7: CPU and memory usage in WordCount job. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

In *Sort* job of 1 GB file, the difference only resides in the memory usage. With Docker, memory consumption is higher than is the case in the native environment as unveiled in Figure A.8a. We have also noticed a few irregularities in CPU usage in virtualised environment. As for the 4 GB *Sort* job, Figure A.8b demonstrates nearly identical patterns in both environments. Figure A.8c demonstrates a more obvious difference in CPU utilisation between two environments in which virtualised platform exhausts CPU resource earlier and for longer periods of time.




(a) (b) (c)

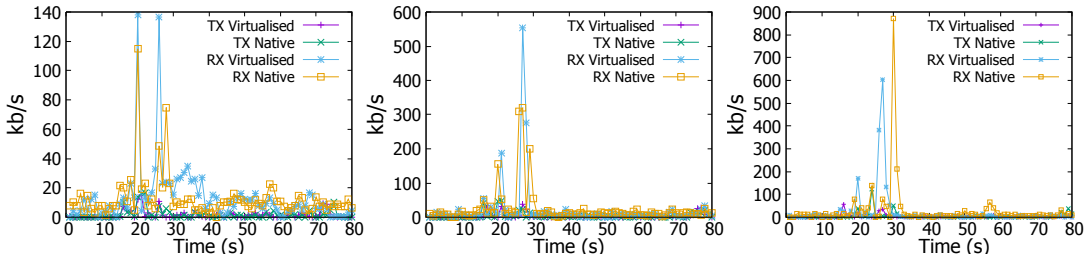
Figure A.8: CPU and memory usage in Sort job. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

These set of experiments have demonstrated that virtualisation incurs a more prominent overhead when the jobs are more demanding.

### Virtualisation Impact on Network Usage

Figure A.9a shows that *WordCount* does not produce significant network traffic with two spikes at the rate of 140 kb/s. Similarly, Figure A.9b shows very small difference in network throughput for 4 GB job in *WordCount*. However, the network behaviour becomes different for 6 GB job. Network reception rate becomes

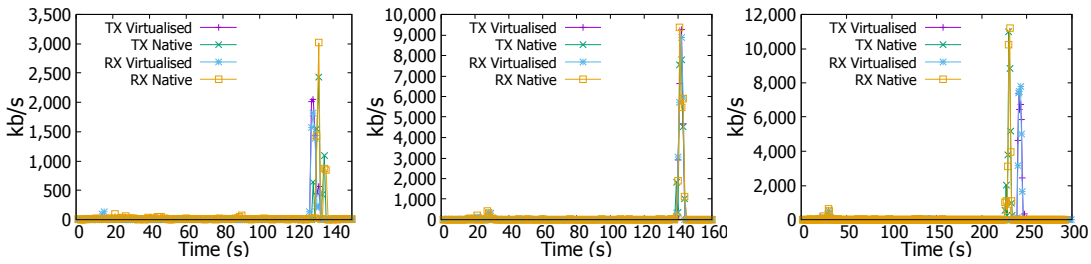
more intensive in the native environment than it is in the virtualised counterpart as shown in Figure A.9b. For example, at 28-th s reception rate in virtualised environment reaches nearly 600 kb/s while in the native environment it is nearly at 900 kb/s.




(a) (b) (c)

Figure A.9: Transmission (TX) and reception (RX) rates in WordCount job. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

In *Sort* job, we have noticed a different network behaviour from the case in *WordCount*. In Figure A.10a there is a high network traffic at the end of the experiment, this is a consequence of the *shuffling* process where workers are sharing results for consolidation. Reception and transmission rates are more intensive in the native environment than where Docker is running. In Figure A.10b we have found identical behaviour in network usage in both environments, however the rate is higher than it is in 1 GB file for the same job; transmission and reception rates reach nearly 9.600 Mbps.




(a) (b) (c)

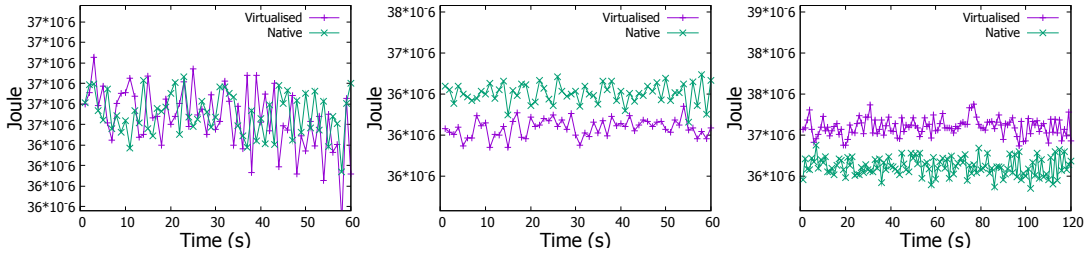
Figure A.10: Transmission (TX) and reception (RX) rates in Sort job. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

Lastly, we can see from Figure A.10c that network usage is remarkably more intensive in the native environment. For instance reception and transmission rates reach 11.2 Mbps in the native environment while they are at nearly only 8 Mbps in virtualised one. The difference is about 3.2 Mbps or 28.6%.

### Virtualisation Impact on Energy Consumption

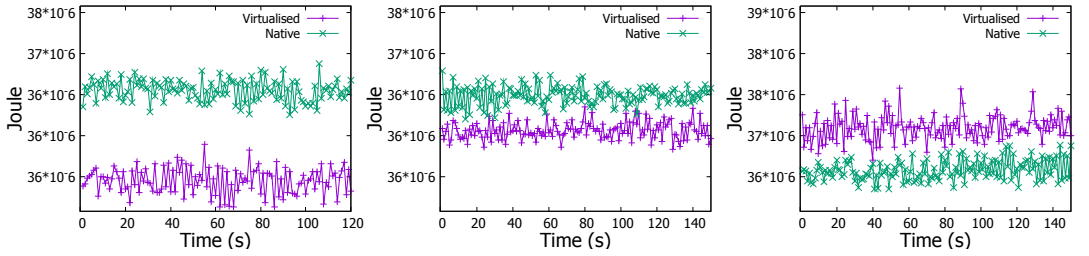
In this section, we will investigate how much overhead, if any, virtualisation has in terms of energy consumption.

Figure A.11a depicts the energy consumed by a Raspberry Pi cluster worker member when it is involved in *WordCount* job on 1 GB file, energy levels are very similar. However for *WordCount* on 4 GB file, energy is more important in the native environment than in virtualised one as shown in Figure A.11b. However, in *WordCount* for 6 GB job, as revealed in Figure A.11c energy level becomes clearly higher when jobs are running inside Docker containers. It arises from $3.66\times10^{-5}$ Joule to $3.71\times10^{-5}$ Joule, so an increase of 1.3%. For *Sort* job, same patterns have been observed for the case of 4 GB and 6 GB jobs as shown in Fig. A.12b,c.




(a) (b) (c)

Figure A.11: Energy measurement in WordCount job. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.




(a) (b) (c)

Figure A.12: Energy measurement in Sort job. (**a**) 1 GB file; (**b**) 4 GB file; (**c**) 6 GB file.

## A.3 Summary

In this work, we have designed and presented a set of extensive experiments on a Raspberry Pi cloud using Apache Spark and HDFS. We have evaluated their performance through CPU and memory usage, Network I/O, and energy consumption. In addition, we have investigated the virtualisation impact introduced by Docker, a container-based solution that relies on resources isolation features

available on Linux kernel. Unfortunately, it has not been possible to use Virtual Machines as a virtualisation layer because this technology is not yet supported in the current releases on Raspberry Pi.

Our results have shown that the virtualisation effect becomes more clear and distinguishable with high workloads, e.g., when operating on a big amount of data. In a virtualised environment, the running tasks require more CPU and memory consumption while the network throughput decreases, and burstiness occurs less often and less intensively. Furthermore, it has been proven that energy level consumed by the Raspberry Pi arises with the high workload and it is additionally affected by the virtualisation layer where it becomes more important.

# Appendix B

# Experimental Evaluation of SDN-Controlled, Joint Consolidation of Policies and Virtual Machines

We aim, through a Mininet-based test-bed implementation[1], to evaluate *Sync* and understand which factors determine its performance in terms of execution time and resource consumption. Unlike ns-3 based *Sync* simulation in [54], Mininet based implementation gives realistic results and is readily deployable on real hardware[2].

This Appendix chapter is organised as follows. In Sec. B.1, we introduce the principal algorithms that comprise its processing mechanism. Then, we present our system design in Sec. B.2. Particularly, we discuss the controller implementation and how SDN capabilities have been extended to reflect VMs, flows and policies characteristics. In Sec. B.3, we describe our experiment set-up and evaluate *Sync* based on several criteria. Finally Sec. B.4 concludes the chapter.

## B.1 Sync Algorithm

*Sync* is a synergistic scheme for dynamic VM and policy consolidation runnable on top of an SDN-based environment. The problem formulation and the proposed model primarily deal with hardware-based MBs due to their popularity, better performance compared with their virtualised counterpart, and their flexibility and support for in-network policy and service deployment. In modelling the problem, we consider a multi-tier DC network, which is structured under a multi-root tree

[1]Source code available on GitHub https://github.com/wajdihajji/sync.git
[2]https://mininet.org/

topology. Our experiments are running atop of $k$-ary fat-tree.

### B.1.1 Get Communicating VM Groups

Handling all VM instances at the same time could incur an intolerable running time for Sync algorithms and it would hinder the scalability characteristics for the whole solution. In real data centres, several tenants share or own a set of VMs or resources, and there are groups of VMs that communicate between each other performing a logically similar operation. The algorithm partitions all VMs into isolated groups in which VMs do not communicate with a VM outside their group. These VM groups will be the input of other algorithms.

A group $G$ is defined as the VMs that communicate between each other, and none has a connection/relationship with other VMs outside the group.

### B.1.2 Policy Migration

This algorithm focuses on migrating the policies, in other words defining again the MBs; replace them with the same type of MBs as the deployed ones. In the meantime, it prepares for the VM migration by updating the *preference matrix* responsible for rating best candidate source and destination servers for VM pairs. Prior to policy migration, the algorithm should have a complete view on the *Cost Network* trees related to each flow and each policy. The *Cost Network* graphs will be the search space of the shortest paths related to policies.

The function responsible for getting the shortest path aims at reducing the Communication Cost through the migration of policies.

We define the *Communication Cost* of all traffic from VM $v_i$ to $v_j$ as

$$\begin{aligned} C(v_i, v_j) &= \sum_{p_k \in P(v_i, v_j)} f_k.rate \sum_{L_s \in R_k(v_i, v_j)} c_s \\ &= \sum_{p_k \in P(v_i, v_j)} \{C_k(v_i, p_k.in) \\ &\qquad + \sum_{j=1}^{p_k.len-1} C_k(p_k.list[j], p_k.list[j+1]) \\ &\qquad + C_k(p_k.out, v_j)\} \end{aligned} \tag{B.1}$$

where $C_k(v_i, p_k.in) = f_k.rate \sum_{L_s \in R(v_i, p_k.in)} c_s$ is the communication cost between $v_i$ and $p_k.in$ for flows which matched $p_k$. Similarly, $C_k(p_k.out, v_j)$ is the communication cost between $p_k.out$ and $v_j$ for $p_k$, and $C_k(p_k.list[j], p_k.list[j+1])$ is the communication cost between $p_k.list[j]$ and its successor MB in $p_k.list$. Note also that $R(n_i, n_j)$ is the routing path between nodes (i.e., servers, MBs or switches).

### B.1.3 VM Migration

Each server has a preferred VM list (which is constructed in *Policy migration* algorithm) to host according to the corresponding *preference matrix and list*. In addition, VM migration incurs a *utility cost* depending on the server destination location. Besides, each server has a limited capacity, which determines whether it can host more VMs. These parameters are considered in the VM migration decision. Since VM and server preferences might be in some cases contradicting, a modified version of a Gale-Shapley algorithm has been adopted to address this challenge and guarantee a stable matching all the time.

The *utility* of migration $A(v_i) \rightarrow \hat{s}$ is defined to be the expected benefit through migration:

$$U(A(v_i) \rightarrow \hat{s}) = \mathcal{C}_i(A(v_i)) - \mathcal{C}_i(\hat{s}) - C_m(v_i) \tag{B.2}$$

where $C_m(v_i)$ is an estimated migration cost related to the VM, and $C_i(s_j)$ is defined, in turn, as:

$$\mathcal{C}_i(s_j) = \sum_{p_k \in P(v_i,*)} C_k(v_i, p_k.in) + \sum_{p_k \in P(*,v_i)} C_k(v_i, p_k.out) \tag{B.3}$$

The VM migration algorithm, for a given VM group, initialises and obtains the preference list (where no policy violation or overused server capacity) of all servers. It sets all VMs as unmatched (no server yet chosen to migrate to). First, it starts with getting the most preferred server through calculating the migration utility and it subsequently checks that the selected server has enough capacity to host the VM. If that's the case then it moves to the next VM in the group, otherwise, it rejects less preferable VMs that were located to the server in question. Following that, it updates the *best_rejected* variable with the most preferred that has been rejected by the server. Lastly, it adds the server to the blacklists of all lower ranked VMs than *best_rejected*.

## B.2 System Design and Implementation

### B.2.1 System Architecture

In Fig. B.1, topology and the controller are running on separated environments, they communicate through OpenFlow to add rules to switches and via out-of-band control channel (network sockets) to exchange or update information related to flows, MB and VM placement, in case a migration decision is made. The controller is composed of mainly 8 modules that work collaboratively to identify VM groups, migrate VMs and policies. In Mininet, OpenFlow switches ensure the

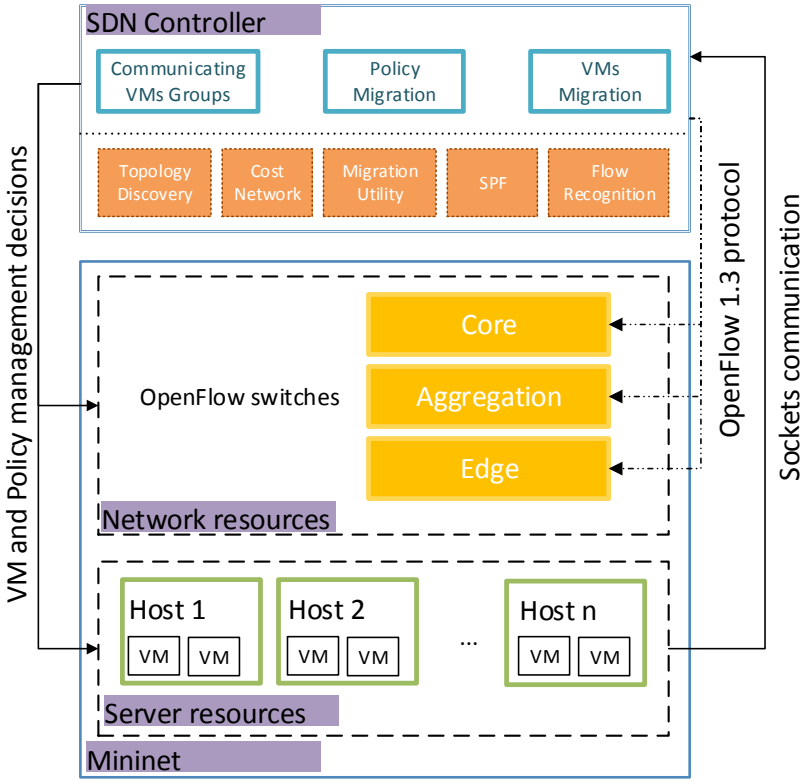



Figure B.1: Architecture design

communication between VMs and servers in a Fat-tree topology.

We consider an MB as Mininet host attached to an aggregation switch. In the experiment, any type of MB only receives and forwards packets with no modification. So when a packet travels from source A to destination B through three MBs (e.g. mb1, mb2, and mb3), it only goes through and the forwarding rules are set, in advance, in the OpenFlow switches. We assume MBs are hardware-based and hence their positions are fixed. In addition, we have modelled the VM as a user process running on a Mininet host (a server in the topology), each process has an ID which is also considered as the ID of the VM. Upon creation or migration, the user process will be created or killed and instantiated accordingly. The policy is defined as a set of 3 MBs, each one governing one or many flows, which are in turn modelled as *Netperf*[3] traffic between VM pairs.

### B.2.2 Controller Modules

In this section, we describe the main components of the controller, their roles, and the interactions between them.

[3]http://www.netperf.org/netperf/

### Topology Discovery

It is a built-in feature in Ryu[4] controller. It keeps track of switches registration/de-registration and added/removed links.

We construct the topology as a graph using Ryu *topology_api_app* module where vertices emulate switches and edges/links are the connections. A major limitation of this function is that it does not have a view of the instantiated VMs, flows, or policies. For this reason we have designed and created, in parallel, a communication channel to make the controller aware of the above information useful for Sync's usage.

### Cost Network Construction

In order to make a decision of policy migration for a given flow, Sync needs to construct a *Cost Network* tree in which hosts and MBs are represented with links and corresponding weights.

For sake of improved performance, we build in advance all the *Cost Network* trees. This preliminary task is justified as the positions of MBs are meant to be fixed (hardware-based) and over a limited period of time, the flows characteristics are still unchanged. In addition, the weight of each edge can updated when used (in case flow rates are changed), and we can also prune the cost network (removing some nodes and edges) if some MBs are not available.

### Shortest Path First (SPF)

This module deals with the *Cost Network* tree of flows related to a given VM group. It gets the shortest path for a flow traversing a chain of MBs according to a specific policy. It returns the optimum positions of server source and server destination and the set of MBs in between.

### Flow Recognition

In the controller, a flow database is built following the reception of information from the network regarding communicating VMs, therefore their IPs and the used protocol and ports are stored to match against entries in the policy database. Flow information can be obtained by querying the network in which presumably we know in advance what traffic are initiated in a period of time, as the set of flows have been generated randomly in the experiment. In addition, Ryu can get real-time statistics by using the function "ofp_event.EventOFPFlowStatsReply".

[4]https://osrg.github.io/ryu/

#### Utility Of Migration

This module is essential to help with *VM Migration* decision, it aims to evaluate what is the impact on the *Communication Cost* when migrating a VM from source server to destination server. This module is used to get the *maximum utility of migration*, which helps in identifying the candidate servers for the VM to migrate to.

#### Get Communicating VM Groups

This module operates on a Python list of VMs and flows to output $n$ VM groups, which are the input of *Policy Migration* and *VM Migration* modules.

#### Policy Migration

For a given VM group, this module works on a Python list of flows. Using output from SPF module, it migrates policies by updating the corresponding Python dictionary. Finally, it updates the *Preference Matrix* by incrementing the value that corresponds to the key (server, VM) in a Python dictionary as well.

#### VM Migration

It takes as input a VM group and outputs the new allocations for the VMs. It calls other sub-functions such as *"Get Maximum Utility"*, *"Initialise Black List"*, *"Check Server Capacity"*, *"Get Unprocessed VMs"*, and *"Obtain Preference List"*. For each VM in the group, it looks for an optimum location based on the *Utility Cost* and server capacity metrics. In the end, it constructs a Python dictionary that contains the new allocations of VMs and sends it to the topology environment via a Network Socket.

### B.2.3 Communication

The communication between the topology and the controller is ensured by two channels, one via OpenFlow used by Ryu to get acknowledged of the switches and links introduced, updated, or removed, and the second one through Network Sockets used by Sync to get information on instantiated VMs, flows, MBs, and service chains (corresponding to policies).

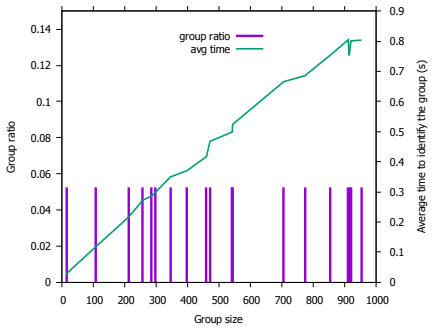
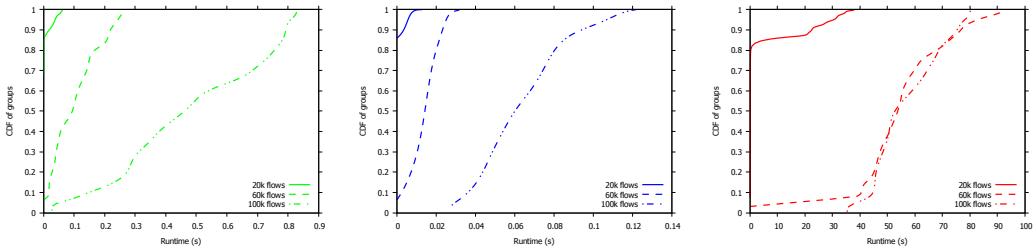



Figure B.2: Group distribution




(a) Get Communicating VMs algorithm

(b) Policy Migration

(c) VM Migration

Figure B.3: *Sync* performance evaluated with growing number of flows, group sizes in the three levels are 36, 31, 19, respectively.

## B.3 Experimental Evaluation

### B.3.1 Experiment Set-up

We ran our experiments on two identical servers (8 Cores/1.2Ghz and 8GB Memory). Ubuntu 14.04 is running atop of them and they belong to the same network and have directly a physical connection through a 1Gbps switch.

In server A, there are Mininet version 2.3.0d1, OpenFlow 1.3[5] and Python 2.7.6. Initially, we create and set OpenFlow switches and hosts, we also construct topology tree, VMs and flows database, which will be shared with the controller modules later on. In server B, we have installed and configured Ryu controller 4.10.

We run *Sync* with different combinations of VMs, flows, policies and MBs. We have fixed our topology size in every run with fat-tree's $k$=14. That means the number of edge switches equals to 98, the number of aggregation switches is 98, the number of core switches is 49, so the number of switches in total is 245, and the number of hosts is 686.

We have run all experiments 10 times to get average results so that we mitigate measurement irregularity and noisy statistical data. Variations in results can be caused by OS tasks running in background or logging processes executed to collect the results.

[5]https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

### B.3.2 Group Formation

*Sync* is designed to operate on VM groups. In order to better understand the performance of *Sync*, it is important for us to show how groups are distributed in the topology. We will then show the efficiency of *Sync*'s *Getting Communicating VM Groups* algorithm for forming these groups. We particularly show results of 100k flows, 10k VMs, and 80 MBs, which is the most representative set-up in our experiment as it involves many VMs and consequently many groups.

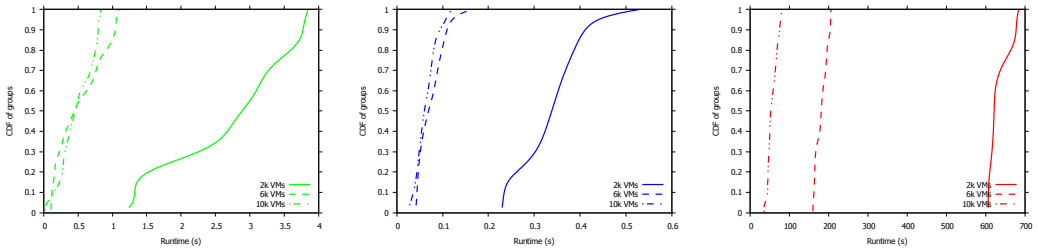
In Fig. B.2, almost every group represents 5% of the set of groups. The curve of average time to form the group evolves linearly with growing number of group sizes, that is expected since, as explained in Section B.1.1, the run time is affected by the group size. However, a slight dip appears for group size 915 which takes about 0.7534s to get identified. This change is due to the difference in order of appearance of groups. To explain this, we look at the group in question and its two neighbours in Fig. B.2, whose sizes are 912, 915, 921 VMs, they take 0.805s, 0.7534s, 0.8017s, and their order of appearance are first, twelfth, and ninth, respectively. So group of 915 VMs appears lastly in the three groups, that means *Get Communicating VMs* operates on less number of VMs and flows at the order twelfth than at the first and ninth iterations. The aforementioned information are read from logs related to the experiment. Same explanation are still applicable on the two groups of sizes 912 and 921 VMs.

### B.3.3 Overall Performance Results

In this section, we study the impact of topology characteristics on *Sync* performance. However, we do not present the consumed network resources as *Sync* is mainly a workload intensive task, and the only network activity induced by it can be seen when sending VM and policy migration decisions to the Mininet topology.

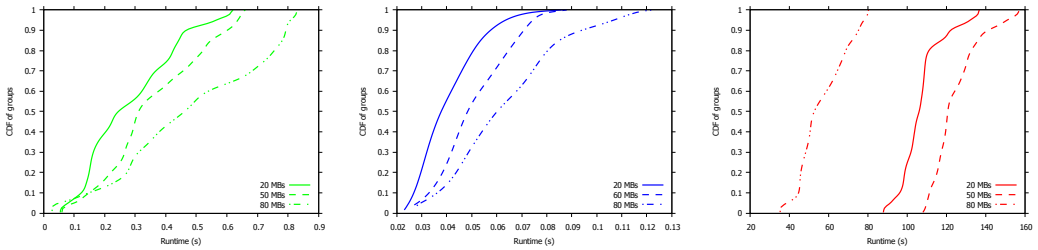
Firstly, we fix the number of VMs and MBs (we set them at the maximum values of the experiment; 10k VMs and 80 MBs in a Fat-tree topology with $k$=14), at the same time, we change the number of flows starting from 20k to 100k flows. In each case, we measure the time taken for each group to run *Sync* algorithms, *Get communicating VM Groups*, *Policy migration*, and *VM migration*.

In Fig. B.3, we observe how the growing number of flows causes longer runtime for *Sync* algorithms. For instance, where the number of flows is set at 20k, all groups finish in 0.08s, 0.01s, and 38s in the three algorithms respectively, at 60k flows, all of them finish in 0.28s, 0.035s, and 90s, and with 100k flows, the runtimes of all groups reach nearly 0.75s, 0.12s, and 80s respectively. In *Sync* design, flows have always been involved in all algorithms. For example, in getting the communicating VMs, *Sync* looks for associated flows to each VM to conclude the




(a) Get Communicating VMs algorithm (b) Policy Migration (c) VM Migration

Figure B.4: *Sync* performance evaluated with growing number of VMs, group sizes in the three levels are 4, 14, 19, respectively.




(a) Get Communicating VMs algorithm (b) Policy Migration (c) VM Migration

Figure B.5: *Sync* performance evaluated with growing number of MBs, group sizes in the three levels are 25, 21, 19, respectively.

relations between VMs and therefore recognise and define groups. This means that when the number of flows grows, the search space becomes larger and more importantly, the VM could have more associated flows. This also leads to an increase in the runtime of other algorithms. The discrepancy seen for VM migration when runtime is 80s for 100k flows, and 90s for 60k flows is due to the fact that the algorithm in question considers, besides the number of flows, the policy violation constraints. The latter depends on the MB positions which are initially set in a random way.

Secondly, we set the number of flows to 100k and vary the number of VMs. Fig. B.4 demonstrates the results for this set of experiments. Surprisingly, we can observe that all three algorithms finish in less time for 10k VMs than for 2k VMs and 6k VMs. With 10k VMs, groups finish in 0.6s, 0.11s, and 135s for the three algorithms, respectively. In comparison, they take 3.8s, 0.51s, and 690s in 2k VMs settings. This is because when there is a large number of VMs, each VM will be source or destination for less flows than where there is a big number of flows and a small number of VMs. In the beginning of the experiment, we randomly allocate flows to VMs. This means, for example, for *Get communicating VM Groups* in the case of 2k VMs and 100k flows, *Sync* checks the flows related to a single VM, and then constructing one group will subsequently be more time-consuming.

This set of experiments has shown that the number of VMs has a measurable

effect on *Get communicating VM Groups* and *Policy migration* on one hand, and *VM migration* on the other hand. In *Get Communication VM Groups* and *Policy migration*, for 6k and 10k VMs, the difference is not apparent, however, it becomes considerable in *VM Migration* algorithm.

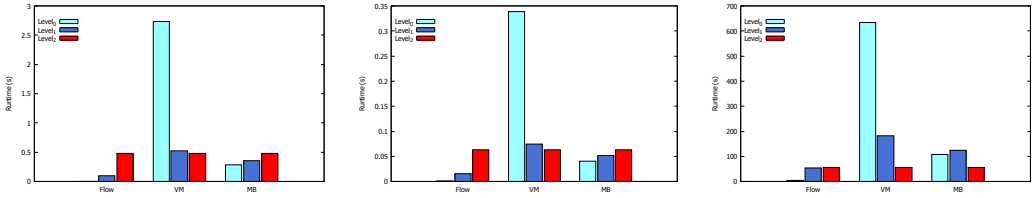
Thirdly, we fix the number of flows and VMs. In Fig. B.5a and B.5b, we observe how run time for *Get communicating VM Groups* and *Policy migration* evolves linearly with the number of MBs, albeit not too significantly. For example, *Get communicating VM Groups* finishes in 0.61s, 0.65s, and 0.81s for 20, 50, and 80 MBs, respectively. The same behaviour is recorded in *Policy Migration* algorithm. However, we do not see the same linear evolution of execution time in *VM migration*. In Fig. B.5c, with 80MBs, it takes less time than for other number of MBs, and the difference is quite noticeable; 80s, 135s and 158s for 80, 20 and 50 MBs, respectively. Thus, the number of MBs have a considerable impact on all the three algorithms unlike the VMs and Flows factors. In *VM Migration* and *Policy Migration*, the number of MBs is involved directly in the processing, as in the former, it is needed to check the feasibility of the migration process, and in the latter, *Sync* will migrate MBs according to the output of *SPF* module described in section B.2.2.

In addition, we have recorded the group average runtime, i.e., how much time on average groups take in each algorithm to finish processing under various settings. In Fig. B.6, there are three histograms, each describes the evolution on run time based on one factor. As an example, in Fig. B.6a, there are 9 boxes, the first three ones present the average runtime of a group in *Get Communicating VMs* when the number of flows evolves from 20k to 60k, to 80k flows (that correspond to the three levels level 0, level 1, and level 2). The second three boxes are for VM levels (2k, 6k, and 10k VMs), and the last three ones for MB levels (20, 50, and 80 MBs).

In *Get communicating VM Groups*, as shown in Fig. B.6a, *Sync* is more sensitive to the number of flows that other factors, but in case the number of VMs is relatively small, the run time increases dramatically to reach 2.6s when the number of VMs, flows, and MBs are set to 2k, 100k, and 80, respectively. Otherwise, the execution time is at most at 0.5s in all other cases and it is, remarkably, at 0.00434s when the number of flows is at 20k.

In *VM Migration*, the number of VMs has a major effect on the runtime of a group, for instance, when the number of VMs as 20k, the algorithm takes nearly 600s, whereas, in case there are 10k VMs, the execution time falls dramatically to reach about 50s.

To conclude, the three factors have a different impact on the *Sync* algorithms, flows impacts more *Get Communicating VMs* and *Policy Migration* algorithms,




(a) Get Communicating VMs algorithm (b) Policy Migration (c) VM Migration

Figure B.6: Group average runtime measured with growing number of VMs, Flows, and MBs

while the number of VMs can alter significantly the time needed by *VM migration* algorithm. Lastly, the number of MBs has a known effect on *Get communicating VM Groups* and *Policy migration*, whereas, in *VM migration*, its impact becomes unpredictable because VM migration decision depends more on policy violation prevention strategy.

### B.3.4 Resources Utilisation

In all experiments, CPU usage has been nearly at 13%, however, the memory consumption depends on the three input of *Sync* algorithms (number of flows, VMs, and MBs). The active memory increases linearly with the growing number of the aforementioned factors, for example, it reaches 3700 Mbytes in "extreme" set-ups (all values set at the maximum of the experiment). We also remark that memory usage grows significantly with increasing number of VMs, and this is explained by the fact that each VM possesses much information that comes with (e.g. associated flows). For other factors, the increase in memory is relatively limited (nearly 100 MBs). Active memory consumption raises with larger topology, but the CPU usage stands at the same level i.e. nearly 13%.

This means that *Sync* is very resource efficient and has room to scale to much bigger topologies. We also note that our implementation is a reference implementation that does not consider optimisation techniques such as parallelism with multi-controller paradigm, in which multiple controllers can process individual groups concurrently.

## B.4 Summary

*Sync* has provided a novel approach to improving DC network performance by considering both VM and MB placement synergistic-ally. We have designed, implemented and extensively evaluated Sync through a Mininet framework. We have found that *Sync*, which is composed of three key algorithms – Get Communicating

VM Groups, Policy Migration, and VM Migration – is not only efficient but also has fractional system resource footprint. In the future work, we plan to improve *Sync*'s efficiency and performance by adopting multiple controllers in which master node will be responsible for forming VM groups and slave nodes will get fair share to continue on policy and VM migration concurrently.