Container-based Service Chaining: a Performance Perspective

Sergio Livi, Quentin Jacquemart, Dino Lopez Pacheco, and Guillaume Urvoy-Keller
Université Côte d'Azur, CNRS, I3S, France
{livi, jacquema, lopezpac, urvoy}@i3s.unice.fr

***Abstract*—Middleboxes, which implement specific network service functions – e.g. firewalls, load balancers, NATs – have traditionally been deployed as hardware appliances, thereby imposing significant constraints on network operators, who must ensure that the traffic is effectively routed to the appropriate set of middleboxes, following the right order. Being hardware-based, these boxes offer limited upgrade capabilities, i.e. minor tweaks for performance tuning. This problem becomes particularly significant in multi-tenant data centers where each tenant requires their own set of network service functions. A recent trend is to virtualize these middleboxes and turn them into so-called *Virtualized Networks Functions* (VNFs) that can be chained to offer the appropriate services to each tenant.**

When the VNFs are implemented as full-fledged virtual machines (VMs), a non-negligible overhead is added due to the kernel of each VM, and it is even more considerable when scaled up for a whole data center. Considering the industry's direction towards containerization technologies, e.g. Docker, we study the case of VNFs implemented as lightweight Linux containers. We seek to understand how performance evolves as a function of the length of chains of various services; and also of different configuration set-up, for example containers directly connected together or through Open vSwitch switches.

I. Introduction

The network infrastructure is composed of different kinds of equipment, some of which, known as *middleboxes*, are responsible for *network services*. These services consist of one or more functions, each of which is responsible for processing packets [1]. Ubiquitous examples of such network services are firewalls, deep packet inspection devices, application accelerators, NATs, etc.

These functions are often implemented as specialized hardware. Physical cables force the packets through rigid chains of services that manipulate data flows; these chains are statically defined at service deployment. The main advantage of this approach is that it guarantees a good quality of service. The price to pay is limited flexibility, along with tedious network design and long deployment times.

Instead of using specifically designed hardware, a relatively new technique called *Network Function Virtualization* (NFV) suggests to set up the services as virtual appliances on commodity hardware. The services, named *Virtualized Network Functions* (VNFs), are then implemented completely as software, enabling fast deployment cycles. A number of use cases for NFV are described in [2].

NFV brings flexibility, elasticity and easy configuration to middleboxes. For example, services can be easily moved around, and chains of services can be modified on the fly. In multi-tenant data centers, it is even possible to use *dynamic* service chaining in order to provide a personalized set of services for each customer [3]. This is particularly important in cloud environments, where numerous *Virtual Machines* (VMs) need to communicate through private networks. These VMs may live on one or more (shared) physical nodes, and NFV can be used to let customers define their own infrastructure.

Multiple virtualization techniques are available to support VNFs. In particular, with hypervisor-based solutions, e.g. VMware, the host *Operating System* (OS) (or a part of it) manages the access of the VMs to physical resources by rewriting a part of the machine code of the guest OS, while the rest of the code runs unmodified. As a general rule, the guests share the host's CPU architecture. With container-based solutions, e.g. Docker [4], everything happens directly in the host operating system: its kernel manages the isolation between the environments by only granting the guest processes access to a limited number of resources. As a result, all containers share the same host kernel, thereby forcefully running a similar operating system.

Because containers share the host OS's kernel, running a VNF from within a container induces a smaller overhead than running the same VNF from within a full-fledged virtual machine, as the VM needs to run its own kernel in the virtual environment. When considering the large number of VNFs required in cloud environments, running them from VMs instead of containers implies that a significant amount of energy and processing power is wasted on running guest kernels.

In this paper we draw a picture of the performance of NFV, when implemented as Linux containers. We present our testbed, in which we use `iperf3` in order to stress chains of VNFs and measure their maximal performance. In particular, we show how chains of VNFs perform as a function of the length of the chain. We also consider different configuration set-ups, including the use of Open vSwitch [5] (OvS) switches. We show that container-based chains of VNFs perform very well under stress tests, leading to only minor performance losses when making the best possible configuration choices. Considering the rising industry adoption of Docker, a project that enables fast packaging and deployment of applications as Linux containers, a Docker-based registry containing standard VNFs would enable fast deployment and reconfiguration of cloud infrastructures.

II. Related Work

Martins et al. [6] propose ClickOS, a NFV platform based on Xen [7], which provides a wide range of performance results. ClickOS is composed of a number of improvements on the Xen networking code, a customized lightweight guest OS, and a tool to manage the VMs. ClickOS is suitable for Xen users, but is unsuitable for containers due to dependency and deployment toolchain incompatibilities.

Blendin et al. [8] propose an infrastructure that facilitates the deployment of dynamic service chains, thereby enabling the creation of one individual chain per end user. They rely on OpenFlow [9] to route packets to the correct service nodes. There is only one service per node, and each node runs multiple instances of the same service, one for each user. In contrast, we focus on the case of service chains hosted in the same node.

Qazi et al. [10] propose SIMPLE, a solution that leverages Software-Defined Networking (SDN) to enforce logical chaining rules between existing services, without further modifications. The benchmarks provided for SIMPLE are focused on both the management part (measuring deploying time), and the communication overhead necessary to install the forwarding rules on the switches. These metrics are then inspected for scalability. The focus of the authors is to evaluate the efficiency of the management structure, and not of the end-to-end chain performance, which is our focus.

Emmerich et al. [11] give detailed benchmarks on OvS and other virtual switching systems. The authors use minimum-sized packets, trying to reach the line rate of 10GbE cards, all the tests including at least one pass in the physical network interface. The VMs used during the tests are full-fledged virtual machines, and not containers.

The author of [12] ran some tests of OvS chains, using two different ways to connect them: *veth* and *patch ports*, as they are used respectively in OpenStack Neutron and Juno. The outcome is similar to ours, presented in Section IV-B.

III. Methodology

In this Section, we present a testbed in which we conducted several experiments in order to understand the performance of chaining VNF within containers.

Our test machine is a Dell PowerEdge R410, equipped with two quad-core Intel Xeon E5606 processors, running at 2.13GHz without Hyper-Threading. The machine contains six modules of 2GiB of DDR3 RAM, running at 1333MHz, i.e. a total of 12GiB. We installed Debian Sid in order to get the latest software updates. Currently, the kernel version is 4.4.0.

Each experiment consists in a 30-seconds session of `iperf3`, repeated 10 times. Opting for `iperf3` is a natural choice for two reasons. First, the active development of version 2 has stopped (only bugfixes are provided). Second, version 3 ships with new parameters, namely `affinity`, that enables to pin the process to a specific processor, and `zerocopy`, that reduces the number of system calls, moving part of the responsibility directly to the kernel (i.e. using the `sendfile` syscall instead of `write`). Furthermore, it is possible to ask `iperf3` to aggressively take all the available bandwidth, even for UDP tests, a chance previously applicable only with TCP congestion control. Depending on the specific conditions, these modifications can significantly improve the performance.

Some initial observations show that a single flow does not saturate all the CPU cores. To reach the peak load, we need multiple simultaneous flows. This scenario is obtained with parallel and independent instances of `iperf3`. For each flow, a pair of one server and one client is launched on a different TCP/UDP port. The recorded throughput is the sum of the values reported by all the instances.

Our goal is to test chains of container-like entities. The topology building blocks are *network namespaces*, the underlying networking part for Linux containers, e.g. Docker [4] and OpenVZ [13]. A network namespace is an independent copy of the system networking stack, equipped with its own interfaces, routing tables and so on. In particular, each interface can be reassigned to another namespace; and any process started inside a namespace can only see the interfaces associated to its namespace. We expect user-friendly container solutions like Docker to add a small but constant overhead, thus our results should be qualitatively valid across all Linux-based containerization platforms.

To connect our namespaces together, thus creating a chain of namespaces, we employed two techniques. The simplest one is the Linux *virtual ethernet* (veth), that is part of the kernel. As an alternative, we used OvS to create virtual switches. Two approaches are possible to connect switches and namespaces: either veth couples, attaching one end to a switch and the other to a namespace, or OvS *internal ports*, assigning them directly to the namespaces.

Other means to interconnect namespaces also exist. For example, instead of OvS, it is possible to use the virtual bridge shipped with the Linux kernel, whose performances are underwhelming when compared to OvS [11]. The VALE switch [14] relies on the `netmap` API to achieve high(er)-speed throughput. However, its use is not yet widespread, and the integration of VALE within the system is not straightforward.

The complete source code for our testbed is publicly available on GitHub at https://github.com/serl/topoblocktest.

IV. Results

As detailed in Section III, our testbed runs on a single machine, and, in order to be able to chain VNFs, we need to emulate the network that provides the connectivity between the namespaces. This can be achieved either with pairs of virtual ethernet (veth), provided by the Linux kernel itself, or through exogenous softwares, such as Open vSwitch (OvS). In this Section, we first consider the maximum achievable performance with TCP and UDP flows on this infrastructure, which gives us the baseline for assessing the performance of service chaining. Then, we consider the performance loss resulting from chains of OvS switches, as they are commonly used in real-world applications, e.g. with OpenStack. After that, we consider the performance of namespace chaining as a

function of the length of the chain, first without any application running inside the namespaces, and subsequently with two standard VNFs: firewalls and traffic shaping.

A. Calibration

In order to assess the performance of service chaining, we first need to know the maximum achievable performances with our testbed. For this reason, we benchmark the testbed with `iperf3`. We run a number of `iperf3` servers in one namespace, and the same number of `iperf3` clients in another namespace; these two namespaces are connected directly with a veth link. Each experiment is repeated 10 times.

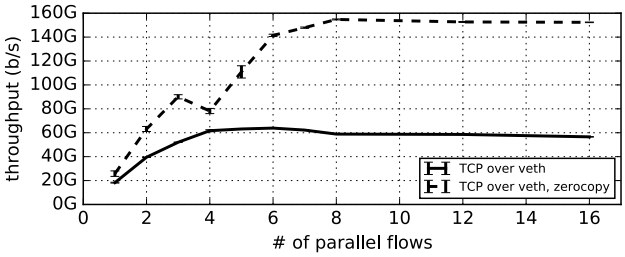


Fig. 1. Throughput with TCP over `veth`, with and without `zerocopy`

Figure 1 plots the throughput obtained on our machine for a number of parallel TCP flows. The full line shows that the maximum throughput for TCP, without any enhancement, is 63.9Gbps±258Mbps. The dashed line shows that the maximum throughput obtained when using the `zerocopy` option of `iperf3` is 154.8Gbps±406Mbps, meaning that `zerocopy` increases performances between 25%-170%. Moreover, we saw that the packet size that we specified to `iperf3` did not impact the obtained throughput because the transmitted packets are aggregated at the network stack as jumbo packets (64 KB).

In contrast, with UDP, Figure 2 shows how packet size significantly impacts the throughput. The maximum throughput is reached for 6 flows composed of 32 KB packets, reaching 105Gbps±2Gbps. Compared to plain TCP, this represents an increase of around 65%; but is a third less than the bandwidth achieved with `zerocopy` activated.

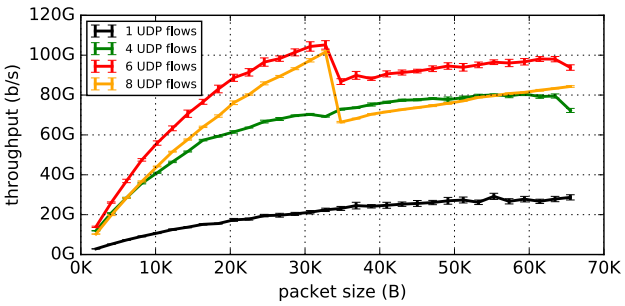


Fig. 2. Throughput with UDP over `veth` in bits/s

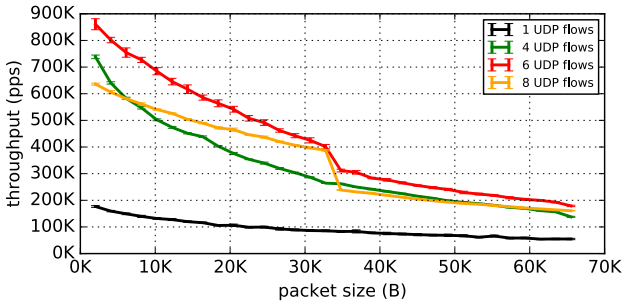


Fig. 3. Throughput with UDP over `veth` in packets/s

In addition, Figure 3 shows that the number of UDP packets per seconds created by the kernel is not constant. The overall load does not depend only on the packets header treatment, but also on their size, even if everything happens on the same physical hardware. When there are more than 4 flows, namely when there are more running instances of `iperf3` than available CPU cores, we see a drop in throughput once the packet size is larger than 32 KB[1]. We believe this behavior is caused by the need for additional CPU cycles, for example for getting additional data into processor caches. We confirmed this intuition by looking at the CPU utilization data, which shows that the processor is fully stressed from 5 flows on.

From these initial benchmarks, we extract our baseline, which equals the best achievable throughput with each protocol: either 8 TCP flows with `zerocopy` enabled, or 6 UDP flows with 32 KB packets.

B. Chains of Open vSwitch switches

In this Section, we study the performance loss induced by chaining OvS switches. Chains of OvS switches are ubiquitous nowadays, as they are the default configuration used by OpenStack. In particular, we are interested in the best way to connect two OvS switches. Figure 4 shows how two OvS switches can be connected either through a pair of veth, or through an OvS patch port.

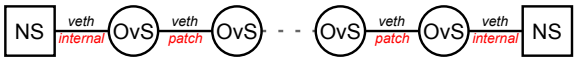


Fig. 4. Methods to connect a pair of OvS switches

Figure 5 plots the performance achieved for 8 TCP flows for a chain composed of up to 20 OvS switches connected with either method. It shows that the performance of veth is lower than the performance of patch ports. As explained by [15], [16], opting for OvS pach ports implies that the data path will be unique inside the kernel, effectively meaning that multiple OvS switches behave like a single larger one. This is confirmed by Figure 5, which shows that OvS switches chained with patch port perform independently of the length of the chain. This part of our work strongly confirms the conclusion of [12].

[1]Even though Figure 3 only plots a handful of cases for the sake of readability, our complete set of experiments confirms this behaviour.

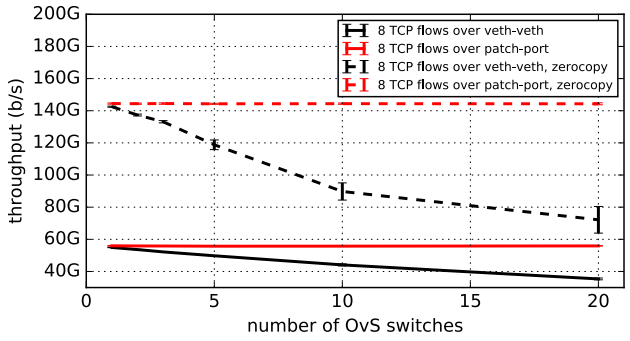


Fig. 5. TCP over a chain of OvS switches

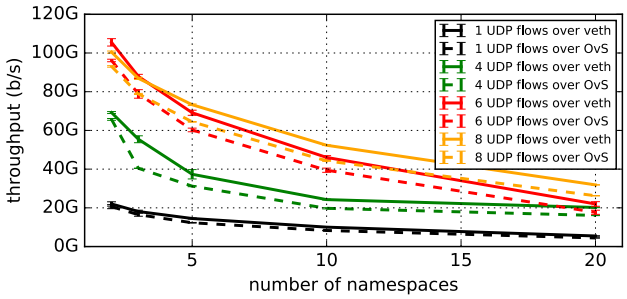


Fig. 8. Throughput with UDP over a chain of namespaces

C. Chains of Namespaces

Now, we focus on chains of namespaces. As Figure 6 shows, there are also multiple ways to connect two namespaces in order to create a chain. The first method is to use a pair of veth interfaces, and to assign each interface to a different namespace, thereby solely relying on the Linux kernel. The second method is to rely on external software, in this case Open vSwitch, in order to connect the namespaces. As detailed in Figure 6, we use internal ports to connect a namespace to a switch, each switch being only connected to two namespaces. Each namespace in the chain is configured to forward packets from one interface to the other one through the use of static routes, and with the forwarding flag enabled.

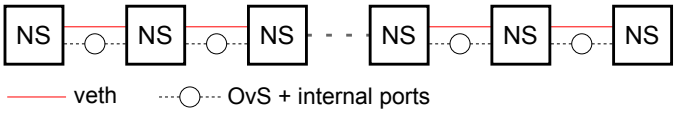


Fig. 6. Methods to chain namespaces

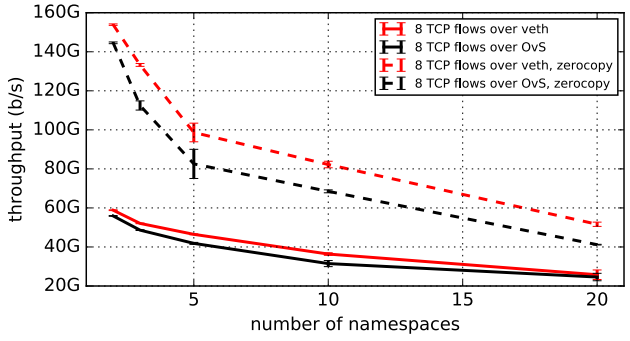


Fig. 7. Throughput with TCP over a chain of namespaces

Figure 7 plots the performance achieved by TCP flows going through namespace chains in these configurations. When OvS is used to connect pairs of namespaces, the performance degrades compared to veth. This is not surprising, as we expect the OvS code to be more complex than the veth one. Moreover, the veth code is part of the Linux kernel, whereas OvS is an external application running on the machine. In addition, in this particular scenario, using OvS is an overkill, as it is used to connect pairs of entities.

Independently from the type of the interconnection links, Figure 7 also shows that the performances evolves with the length of the chain. This decrease in throughput behaves much like the powerlaw $t_{\text{out}} = (1-l)^n t_{\text{in}}$, where t_{out} is the achieved throughput, l the percentage of loss of throughput induced by going through one namespace, n the length of the chain, and t_{in} the input throughput. For example, using the 8 TCP flows over veth, i.e. the red dashed line in Figure 7, we see that the performance loss induced by one additional namespace between $n = 2$ and $n = 3$ is about 11%, when the throughput decreases from approximately 155Gb/s to 138Gb/s. Therefore, by using this power law, the throughput achieved for a chain of $n = 5$ namespaces is $t_{\text{out}} = (1-0.11)^{5-1} \cdot 155 = 97.25$Gb/s, which is the value reported on Figure 7. Please note that we remove 1 to the chain length because we compare to the case where $n = 2$. This behavior also holds true when comparing the performances of veth links and OvS switches. Adding an OvS switch between two successive namespaces induces an additional loss of performance. The performance loss induced by the pair OvS switch/namespace can be modeled as $t_{\text{out}} = ((1-l_{\text{OvS}})(1-l_{\text{ns}}))^n t_{\text{in}}$ where l_{OvS} is the loss induced by the OvS switch, and l_{ns} is the loss induced by the use of a namespace. By using the value of the loss induced by an OvS switch as shown in Figure 5, using the formula we presented, we get the values illustrated by the 8 TCP flows over OvS in Figure 7. However, while these observations hold true for the beginning of the curve, it appears that the decline of performance flattens with an increase of namespaces, i.e. the loss of performance lessens with more namespaces. We believe that a full model of the throughput loss should also take into consideration hardware issues (number of cores, memory architecture, etc.), and, as a result, is more complex than the rule of thumb we presented here. For example, the loss is around 15% for the third namespace, but only 2% per additional namespace when the chain is around 20 namespaces long. Consequently, we leave the full analysis of the evolution of the throughput and its contributing factors when chaining VNFs as future work.

Now that we have analyzed the performance achieved by chaining namespaces, sections IV-C1 and IV-C2 will focus on the performances of real-world standard VNFs implemented within these namespaces, namely firewalls and traffic shaping. Following the results in this Section, we will chain the namespaces using veth pairs. Moreover, we will consider UDP flows exclusively, so as to measure the performance of these VNFs under constant load, without interferences from the TCP stack and/or protocol. For reference, Figure 8 depicts the performance of namespace chains crossed by UDP traffic.

1) Chains of Firewall VNFs: In order to measure the performance of firewalls VNFs within containers, we use the Linux `iptables` firewall, which allows for both stateless and stateful rules. In `iptables`, rules are added to user-defined chains that are part of tables. Each table is used for different kinds of rules, for example filtering, NATing, etc. In this experiment, we add a number of rules to a chain in the filtering table. We use both stateless rules, e.g. a rule matching an unused IP address, and stateful rules, e.g. count the amounts of bytes transferred for all connections. As a result, all the rules are checked and executed by `iptables`, thereby presenting a worst-case scenario.

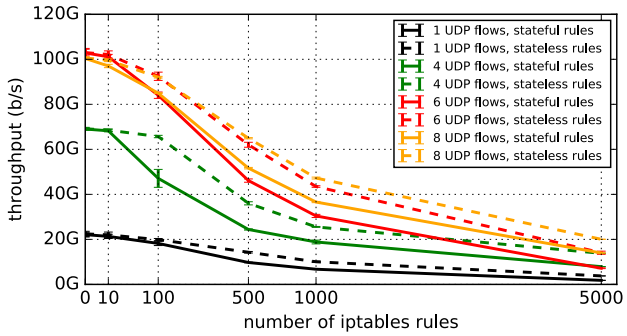


Fig. 9. Firewall VNF: performances for Linux `iptables`

First, we focus on the performance of `iptables` itself. Figure 9 plots the throughput achieved by a single instance of an `iptables` VNF, depending on the number and types of rules, and the number of UDP flows. It shows that stateless rules perform better than stateful rules, and that the firewalling appears to scale with the number of connections, up to our hardware-achievable maximum of 8.

Figure 9 delineates that, even though there is no sudden loss of performance, there is still a degradation of about 50% of the global throughput for 8 flows and 1000 rules. Would it be possible to achieve better performances if the same number of rules are spread over multiple instances of `iptables`, in other words if we divide the rules among multiple VNFs and chain them together? Figure 10 describes the cost of `iptables` rules spread over a namespace chain. In this scenario, we used exclusively stateful rules, to achieve the worst-case scenario possible. The total rules are spread over $n-1$ namespaces, i.e. there aren't any rules in the final namespace. It shows that, in general, independently of the number of rules, the throughput decreases as we spread the rules across an increasing number of namespaces. In most cases, the throughput achieved with chaining (Figure 10) is lower than the throughput achieved with a single instance of `iptables` (Figure 9) for the same total number of rules. In other words, the additional routing costs due to additional namespaces (as the packets need to be routed from a namespace to the next one) do not compensate for the loss of performance due to an increase in the number of `iptables` rules.

The specific case of performance improvement for 1000 rules and 4 flows (Figure 10b) can be easily explained by the specifics of our hardware environment: in this scenario the number of processes is equal to the number of processors, meaning that there are no scheduling issues.

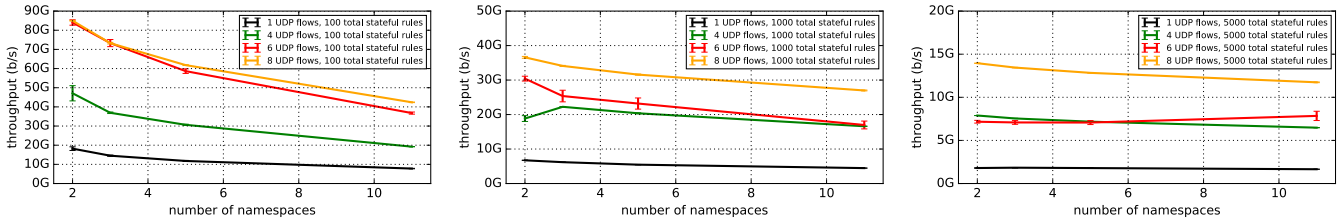
Comparing figures 10a, 10b and 10c together, we see that, as the total number of rules increases, the loss of performances due to an additional `iptables` VNF decreases. This is because the routing costs gets closer to the application cost, i.e. to the cost of running `iptables`. As a consequence, for a large number of rules, it may be beneficial to spread the rules over a chain of namespaces. However, since chains of VNFs are deployed in a data center following a tenant request, and since the basic infrastructure itself is already secured, it is unlikely that any tenant would require so many custom firewall rules.

2) Chains of Traffic Shaping VNFs: In order to measure the performance of traffic shapers as VNFs within containers, we use the Linux package "traffic control", `tc`, in order to set up a *hierarchical token bucket* (HTB) discipline. We set a hard-limit rate of 1 Tb/s and a large buffer. Figure 11 depicts the performances measured in this setup, according to the length of the chain of VNFs. These results are very similar to the 100 `iptables` rules results shown in Figure 10a. This is because, in both cases, the VNFs do not handle much work: 100 rules is a relatively low number of rules for a firewall, and the HTB shaper is quite simple. As a result, most of the performance loss induced in both experiments mostly display the overhead related to the use of additional namespaces, and not of the application.

V. Conclusion

When compared to traditional system virtualization techniques, i.e. fully-fledged virtual machines, Linux *containers*, which are built on top of *namespaces*, enable significant resource savings by isolating the application process, while sharing the rest of the operating system (kernel, libraries, etc.) with other processes concurrently running on the machine. This makes containers an appealing candidate for implementing NFV-based service chains in public (and private) data centers where each tenant requires their own set of network functions to be applied to their network traffic.

In this paper, we investigated the raw performance that can be achieved with container-based service chains, where the whole chain is hosted on a single physical server. We discussed the best approach to interconnect containers with one another,



(a) 100 total rules (b) 1000 total rules (c) 5000 total rules

Fig. 10. Firewall VNF chaining: UDP over a chain of `iptables` namespaces with stateful rules

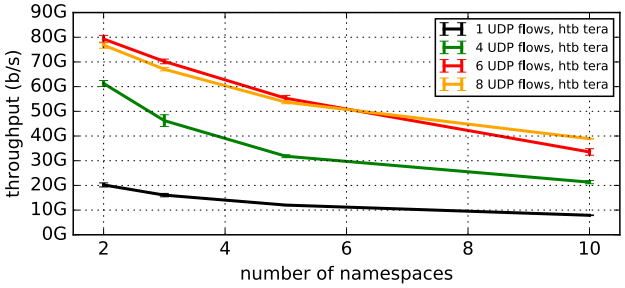


Fig. 11. Traffic shaping VNF chaining: UDP over a chain of `tc` namespaces running a hierarchical token bucket

that in our case turned out to be Linux virtual ethernet. We presented a first approximation of the performance loss model with respect to the chain length that is based on a power law function. This model, while satisfactory with chains of small lengths, is not precise enough when considering chains of tens of namespaces: there appears to be dependencies on the physical hardware that powers the network and the VNFs. Moreover, we considered chains of real-world VNFs, namely firewalls and traffic shapers. In this case, we showed that the networking cost is in general greater that the application cost, suggesting to group the services in the least number of namespaces, if possible.

As future work, we intend to pursue our efforts by investigating the impact on the performance as a function of the number of active chains. Another key point concerns the VNFs chained: it is more likely that a real-world scenario would involve multiple distinct VNFs, instead of homogeneous services. As a result, the application cost would differ from VNF to VNF, and optimizing the process over the whole chain will prove to be challenging. We are also interested in studying how chains, or part of chains, could be shared among tenants, i.e. how the management of a data center could minimize the amount of deployed VNFs by grouping identical tenant requirements inside the same namespace, and appropriately forwarding the flow to the next link of the service chain, or end host.

Acknowledgment

This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future". Program reference #ANR-11-LABX-0031-01.

References

[1] N. Yadav, J. Guichard, B. McConnell, C. Jacquenet, M. Smith, A. Chauhan, M. Boucadair, P. Quinn, R. Manur, T. Nadeau *et al.*, "Network service chaining problem statement," IETF Informational Internet Draft, Jan 2014.

[2] European Telecommunications Standards Institute, "Network functions virtualisation (NFV); use cases," Group Specification Network Functions Virtualisation (GS NFV) doc. 001 V1.1.1, Oct 2013.

[3] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu, "Research directions in network service chaining," in *IEEE SDN for Future Networks and Services (SDN4FNS)*, 2013, pp. 1–7.

[4] Docker, http://www.docker.com/.

[5] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending Networking into the Virtualization Layer," in *8h ACM Workshop on Hot Topics in Networks (HotNets)*, Oct 2009.

[6] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 459–473.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM Special Interest Group on Operating Systems (SIGOPS) Review*, vol. 37, no. 5, pp. 164–177, 2003.

[8] J. Blendin, J. Ruckert, N. Leymann, G. Schyguda, and D. Hausheer, "Position paper: software-defined network service chaining," in *3rd IEEE European Workshop on Software Defined Networks (EWSDN)*, 2014, pp. 109–114.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 38, no. 2, pp. 69–74, 2008.

[10] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 43, no. 4, 2013, pp. 27–38.

[11] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *3rd International Conference on Cloud Networking (CLOUDNET)*, Oct 2014, pp. 120–125.

[12] Open Cloud Blog, "Switching Performance – Chaining OVS bridges," 2014, http://www.opencloudblog.com/?p=386.

[13] OpenVZ, https://openvz.org/.

[14] L. Rizzo and G. Lettieri, "VALE, a switched ethernet for virtual machines," in *8th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012, pp. 61–72.

[15] Open vSwitch, "Frequently asked questions: How do I connect two bridges?" https://github.com/openvswitch/ovs/blob/master/FAQ.md#q-how-do-i-connect-two-bridges, May 2016.

[16] J. Pettit, "OvS performance with OpenStack Neutron," http://openvswitch.org/pipermail/discuss/2013-December/012383.html, Dec 2013.