PhantomSFC: A Fully Virtualized and Agnostic Service Function Chaining Architecture

Matheus S. Castanho[*], Cristina K. Dominicini[*†], Rodolfo S. Villaça[*], Magnos Martinello[*], Moises R. N. Ribeiro[*]
Federal University of Espírito Santo (UFES)[*], *Federal Institute of Education, Science and Technology of Espírito Santo (IFES)*[†]
Vitória, Brazil
mscastanho@gmail.com, cristina.dominicini@ifes.edu.br, rodolfo.villaca@ufes.br, magnos@inf.ufes.br, moises@ele.ufes.br

Abstract—**Service Providers (SP) have deployed virtualized network functions (VNFs) as key elements for handling traffic in the SP domain. Packets are steered to follow a certain order through a set of VNFs before reaching their destination. The provision of such chaining is called Service Function Chaining (SFC). However, current SFC implementations require complex management and are tailored to specific platforms or devices. In this paper, we propose PhantomSFC, a fully virtualized SFC architecture that aims to clearly decouple the service plane from the underlying network. The approach enables SFC to be network agnostic, allowing it to be deployed in multiple scenarios. Besides, SPs can scale resources allocated to SFC execution according to service demands. A proof-of-concept (PoC) prototype of the PhantomSFC architecture is built as a DPDK application to demonstrate the feasibility of our approach. Results show it can handle SFC operation with reasonable performance, considering latency, jitter, and throughput.**

Index Terms—**cloud, SFC, DPDK, NFV, and SDN.**

I. Introduction

Among the hottest topics in the field of computer networks today, Network Function Virtualization (NFV) is certainly one that deserves notice. It consists on an approach to migrate applications that were traditionally implemented in hardware to virtualized infrastructures, running in commodity off-the-shelf devices. This decreases acquisition and management costs, while providing more flexibility and scalability.

Network functions can be combined into chains, an ordered list of functions to be executed sequentially, providing composed services. Examples of such functions include firewalls, load balancers, and deep packet inspectors. This interconnection of functions is called Service Function Chaining (SFC).

Traditionally, SFC implementations rely on complex routing schemes to steer traffic through network functions [1]. This makes modifications to service functions difficult, usually requiring significant changes in network configuration, which is error-prone and cumbersome. Because these deployments are tightly coupled to network topology, they also tend to be restricted to a specific Service Provider (SP) domain, hardly being applicable to different scenarios. Besides, different third-party service functions have a low level of interoperability.

The research leading to these results received funding from the European Commission H2020 programme under grant agreement no. 688941 (FUTEBOL), as well from the Brazilian Ministry of Science, Technology, Innovation, and Communication (MCTIC) through RNP and CTIC. This work was partially supported by CAPES, CNPq and FAPES.

SFC has gained momentum in the past years [1]–[5] causing the Internet Engineering Task Force (IETF) to publish a problem statement for SFC defining key areas that working groups could investigate towards new SFC solutions, in the form of RFC 7498 [6]. As a product of such an effort, RFC 7665 [7] proposes a reference architecture for SFC, but lacks the definition of an encapsulation scheme. Such an encapsulation is proposed by a more recent standard, RFC 8300 [8], which presents the Network Service Header (NSH) protocol.

Although addressing some of the problems listed by RFC 7498, current NSH-based solutions have been developed for specific technologies or frameworks and require specific software or hardware that support the NSH protocol [9]–[11]. Therefore, it is hard to port them to different environments and truly decouple the service plane from the specificities of the dataplane implementation. Furthermore, due to the elastic nature of service demands, SPs need to efficiently allocate resources, avoiding over- and underprovisioning. This also applies to resources allocated to SFC tasks, such as classification, forwarding, and encapsulation. However, current solutions usually integrate the elements responsible for packet steering into infrastructure components, such as hardware and software switches [9]. Thus, it is necessary to provide alternatives to achieve greater flexibility and scalability not only to the service functions, but to the entire SFC architecture.

In order to enable a greater degree of scalability, flexibility, and independence from the underlying network, we propose PhantomSFC, a fully virtualized and agnostic SFC architecture. It builds upon RFCs 7665 and 8300, implementing all SFC components as virtualized network functions (VNFs), and decoupling the service plane from the network infrastructure. The main contribution of PhantomSFC is the ability to perform SFC in a transparent manner, without requiring the network to be aware of the chaining. To address performance issues inherited from virtualization, we harness the power of the Data Plane Development Kit (DPDK) [12].

This paper discusses the design of PhantomSFC and analyzes a prototype built to evaluate the viability of such approach. It is divided as follows: §II discusses related works; §III details PhantomSFC architecture; §IV describes our prototype and the design choices made during implementation; and §V presents tests and results. We finish in §VI with conclusions and future works.

II. Related Works

PhantomSFC focuses on the design and implementation of SFC mechanisms to enable the steering of flows through an ordered set of VNFs. Although not in the scope of this work, it can benefit from management and orchestration solutions [13], [14], and optimized placement strategies [15].

A few different approaches have been used to provide SFC [16], and herein we highlight three of them. The first consists on classifying packets based on packet's intrinsic information, such as header field values, at the end of each Service Function (SF). A central entity configures each SF with rules to forward packets to the next instance in the chain depending on the packet being handled, as implemented in [10]. The problem with this technique is that there is no notion of progression or end of path, since each SF only knows the next hop.

A second approach uses a series of packet encapsulations over the original packet [17], each addressing one SF in the chain. After packet processing, each SF removes the outmost packet encapsulation and uses the next set of headers to send it to the next SF. This process is repeated until all encapsulations are removed and the packet reaches the final destination. However, stacking many encapsulations can lead to packet fragmentation due to limited MTU sizes along the way.

Another option, is to create a service path identifier for each existing chain and attach it to each packet. Elements responsible for moving packets between SFs can, then, consult this information and decide what is the next hop at each stage into service path execution. This information can be included in existing packet fields, like MAC addresses [18], or in a new header, which is the approach adopted by NSH-based solutions [9], [11]. Using a specific header requires all elements acting over the packets to be aware of this new header and also careful consideration of MTU size to handle fragmentation.

OpenStack [10] implements SFC through Neutron, the module responsible for networking. It treats a service path as a list of pairs of Neutron ports, each pair containing ingress and egress ports. Service paths also have flow classifiers, which decide what packets should be handled by each corresponding path. Neutron uses this information to send packets from egress to ingress ports, following the sequence in the port pair list, thus providing SFC. OpenDaylight's [19] implementation is based on RFC 7665 and uses NSH for SFC encapsulation. SFC element functionality is integrated into other components, instead of being standalone entities. Traffic classification is implemented in two ways: using OpenFlow switches, such as OVS, or using iptables with NetFilterQueue. Rules are added to OVS switches during chain creation. These devices also steer packets through network functions. Fast Data (Fd.io) project presents another implementation [11], which is also based on RFCs 7665 and 8300. There, each architectural element specified by these documents are implemented as nodes for FD.io's graph-based virtual switch, VPP [20].

All these implementation approaches strongly rely on specific architectures or software components for which they were designed. This restricts their use to certain frameworks or technologies, without being portable for different scenarios. In contrast, PhantomSFC implements a fully virtualized approach that is agnostic to the underlying network and can be applied to a broader spectrum of infrastructures and scenarios.

III. PhantomSFC Proposal

This section presents the basic concepts, the architecture, and the use cases of our proposal, called PhantomSFC.

A. RFC 7665 and RFC 8300

Our proposed solution is based on IETF's RFC 7665 [7], which specifies a reference architecture for SFC, as illustrated in Fig. 1. It is composed by the following components:

1) **Classifier**: classifies input packets to choose which service chain should be executed;
2) **Service Function Forwarder (SFF)**: steers packets between SFs in the correct order for each path;
3) **Service Functions (SFs)**: performs some computation over a packet;
4) **Proxy**: supports SFs that are not aware of the SFC mechanisms (e.g. legacy service functions).

These elements should operate over an SFC encapsulation, which is a protocol to enable service function path execution. PhantomSFC uses NSH, standardized by RFC 8300 [8], as its SFC encapsulation protocol. NSH has been chosen for three main reasons: (i) it is compliant with RFC 7665 and independent of the transport encapsulation used; (ii) it allows metadata sharing between SFC elements, which is important to carry service management and configuration information; and (iii) being an RFC, it's likely that it will be improved and more widely adopted over time.

The NSH carries two chain identifiers: a 3-byte Service Path Identifier (SPI), and an 1-byte Service Index (SI), the former indicating the selected chain, and the latter the current position in the chain. The SI is decremented at each step of execution by each SF (or by a Proxy). These two fields are used to steer packets in the correct order through a chain.

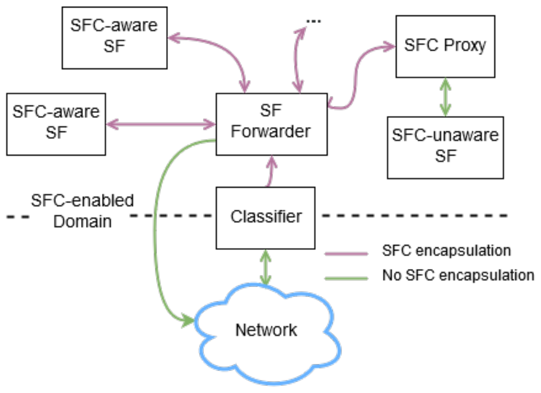


Fig. 1: Illustration of RFC 7665 reference architecture. Adapted from [7].

As shown in Fig 1, packets are classified upon arrival to the network by the Classifier. In this step a packet is matched against a set of pre-configured rules and a service path is chosen, if one exists for that packet. An NSH header containing the corresponding path information is added to the packet and it is sent to the next element in the architecture: the SFF. This element is responsible for deciding which SF should be executed at any given time for each path, following the correct order established for it. The SFF also handles service chain termination and final decapsulation of packets.

Service Functions (SFs) can be split into two categories: SFC-aware and SFC-unaware. The former receives NSH-encapsulated traffic from the SFF while the latter has no idea that it is part of an SFC environment and do not operate on packets with NSH headers. To support SFC-unaware SFs a Proxy is used between the SF and the Forwarder. The Proxy removes the NSH prior to sending packets to those SFs and adds it back again for packets bound to the SFF, also decrementing the SI after each SF execution.

B. *PhantomSFC*

Building upon the concept of NFV, we propose PhantomSFC, a fully virtualized SFC architecture that was designed to meet the following requirements:

1) **transport independent**: it should allow various transport schemes in the underlay (e.g., VXLAN and GRE);
2) **network agnostic**: it should operate over any kind of network, regardless of the topology and devices used;
3) **elastic**: it should be able to scale elements in and out, according to varying demands and workloads; and
4) **small footprint**: it should add small performance overhead, considering end-to-end latency and throughput.

To meet the first requirement and achieve independence from the packet transportation scheme, PhantomSFC relies on the reference architecture proposed by RFC 7665 and the NSH protocol (RFC 8300). Although other SFC implementations are already based on such RFCs, the components of the SFC architecture are usually understood as logical entities and the actions performed by each one are integrated into infrastructure elements, as explained earlier for the case of OpenDaylight and Fd.io. To decouple the service plane from the dataplane, PhantomSFC turns all SFC components into standalone and virtualized entities, each performing its own set of actions and using the underlay network only for communication, without being part of it. The advantage of such approach is that network elements, like routers and switches, don't need to be aware of the chaining. Instead, they just provide packet routing and forwarding, while SFC is done in a virtual environment. This way, PhantomSFC is network agnostic as defined in the second requirement, because it does not rely on specificities of the network infrastructure.

Fig. 2 illustrates the proposed architecture. All SFC components (Classifiers, Forwarders, and Proxies) are implemented as VNFs instantiated by a hypervisor alongside the SFs, and can be distributed across many different servers. Communication between instances in the same physical server is carried by a virtual switch, and the underlay network is used for communication across servers. The virtualization of SFC components tackles the scalability requirement, as each component may be implemented by a number of distributed instances that can be scaled according to the workloads. To allow dynamic chain configuration and instantiation, a logically-centered SFC Controller is used. It is responsible for registering and managing architectural elements as well as handling chain creation, removal and modification through configuration rules in Classifiers, Forwarders, and Proxies. Design of the control plane is out of the scope of this paper.

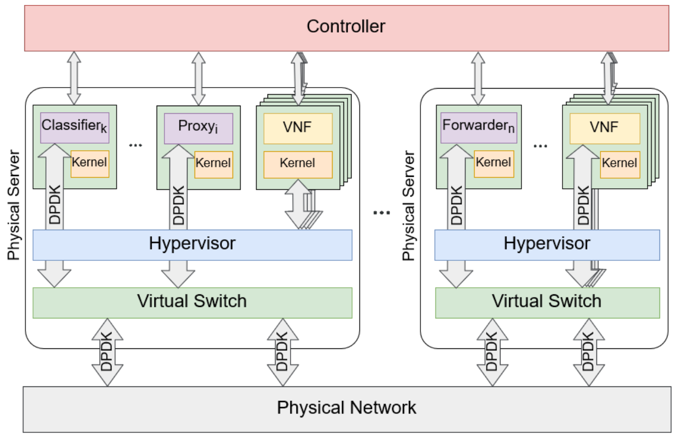


Fig. 2: Diagram of PhantomSFC architecture.

However, performance overhead is added when components are decoupled from network devices and implemented above a virtualization layer. In order to minimize this effect and meet the fourth requirement, PhantomSFC leverages the DPDK library for fast packet processing and kernel bypass. This technology was chosen because of its good performance [21] compared to others, like PF_ZRING [22] and Netmap [23].

C. *Use case scenarios*

Fig. 3 shows a common use case scenario for PhantomSFC. It consists of two endhosts communicating through an SP's cloud environment based on PhantomSFC. The endhosts should be totally unaware that a service chaining is being executed over the traffic exchanged between them.

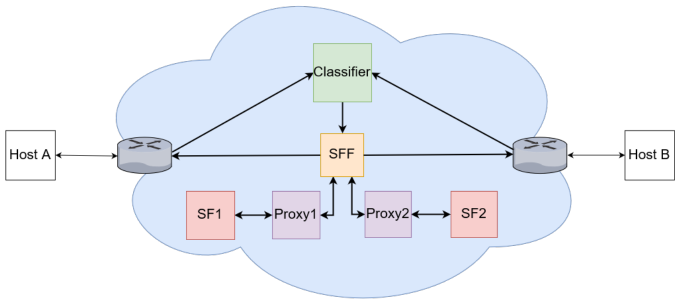


Fig. 3: Use case for PhantomSFC

Due to the independence between the PhantomSFC architecture and the dataplane implementation, SFC components may be either managed by the SP or the tenants. In the first case, the SP transparently provisions additional DPDK-enabled VMs that will be co-located with the tenant SFs in order to provide SFC functionalities. In the second case, the tenant itself requests DPDK-enabled VMs and deploy the SFC components to manage its own SFC environment.

This last use case can be considered an appealing application for SPs, since, to the best of our knowledge, existing solutions do not give this level of control to tenants. In current approaches, tenants cannot manage the SFC mechanisms, because they would have to know the internal workings of the cloud infrastructure and could even interfere in its isolation properties. On the other hand, with PhantomSFC, SPs can offer VNFs from a service catalog that implement SFC components that can be instantiated according to tenants' needs.

IV. Implementation

This section describes the implementation of the PhantomSFC components as DPDK applications and a proof-of-concept (PoC) prototype of the PhantomSFC architecture.

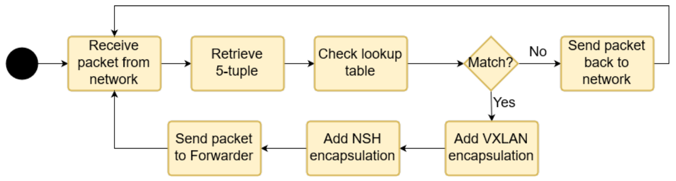
A. DPDK application

Although different in functionality, the three types of SFC components (Classifier, SFF or Proxy) share a common set of actions that are performed as part of a packet processing loop. This includes, but is not limited to, packet reception and transmission, addition and removal of NSH encapsulation and parsing of configuration files. To avoid code repetition, a single multi-purpose DPDK application was implemented, which handles all common initialization before calling the main loop, specific to each SFC component type.

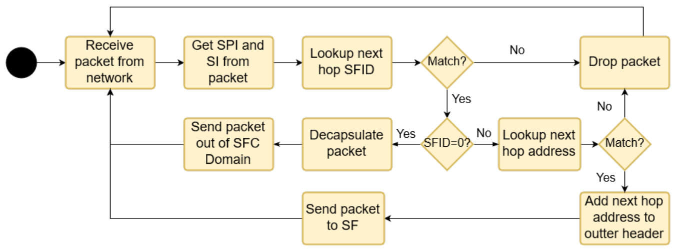
Classifier's actions can be summarized in three steps: classify packets based on a 5-tuple (source IP, destination IP, L4 protocol, source port, and destination port) to determine service path, encapsulate packets with NSH, and send them to an SFF. Beyond the NSH, a transport encapsulation is added to packets, to allow communication between each element in the architecture. In our implementation, VXLAN was used for this matter. Classifier functionality is described in Fig. 4a.

The SFF acts as a "switch for SFs". It is responsible for deciding what is the next hop for each packet, based on local information and the NSH, and removing NSH encapsulation when chain execution has finished. Packet forwarding is done based on a table matching NSH chain information (SPI and SI) to the corresponding next hop address. This table was split in two, adding an extra level of indirection as explained in [8], and the same was done in the Proxy. This prevents the same IP/MAC addresses from begin stored multiple times, which can increase lookup table sizes dramatically. Instead, each address is stored only once and is associated with an SF ID, unique to each SF instance. However, this approach comes at the cost of an extra lookup per packet. All lookup tables used were implemented using DPDK's rte_hash data structure, which is based on Cuckoo Hashing [24] algorithm. Fig. 4b summarizes all actions taken by the SFF.

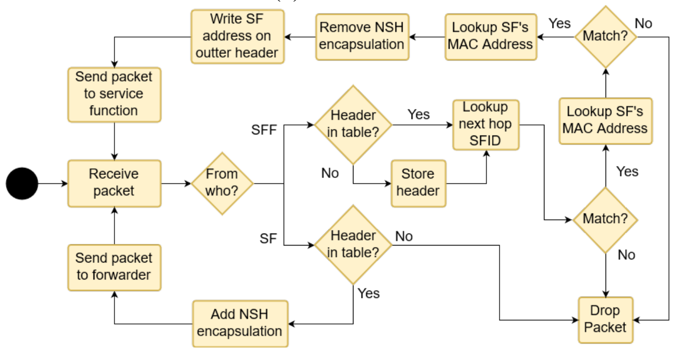
The Proxy intermediates communication between SFC-unaware SFs attached to it and the SFC domain. Just like the SFF, it needs to match NSH's SPI and SI values to SF ID and address of the corresponding SF. The difference lies in the removal of the NSH from packets before they are sent to SFC-unaware SFs. On their way back from the SFs, packets need to be encapsulated again with the same header they came in with. The Proxy also needs to decrement the SI field in the NSH before sending the packet back to the SFF. Fig. 4c shows a flow diagram showing its functionality.



(a) Classifier



(b) Forwarder



(c) Proxy

Fig. 4: Flow diagrams of SFC components.

Our current DPDK implementation considers the following scope: (i) **no loops or branches**: chains must be directed graphs without loops, to avoid proxy table mismatches, or branches, due to the lack of a branch synchronization mechanism; (ii) **static configuration**: all table configuration in each element is currently done through static files loaded during initialization, because the definition and implementation of an SFC control plane is not in the scope of this work and will be implemented in the future; and (iii) **single-core**: each DPDK instance uses one vCPU core for packet processing; a multicore implementation is left as future work.

B. Prototype

The prototype aims to be a PoC to demonstrate and evaluate the core functionalities of PhantomSFC. To this end, we replicate the scenario shown in Fig. 3, where two hosts (Hosts A and B) exchange packets and all communication between them goes through the SFC environment deployed.

In the prototype, we considered a single chain, which goes through two different service functions SF1 and SF2, in this

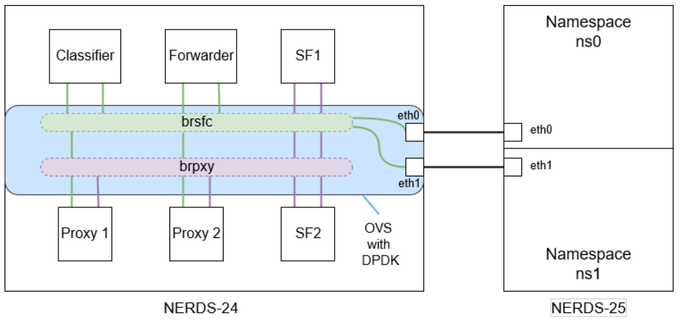


Fig. 5: Prototype infrastructure deployed in our testbed

order. Both SFs are SFC-unaware and are proxied by two separate SFC Proxies, one for each SF. The code run by these functions is a Python script providing a simple forwarding mechanism acting as a virtual loopback cable.

The physical setup is illustrated by Fig. 5 and consists of two servers: a device under test (DUT) running the entire PhantomSFC architecture consisting of 6 virtual machines (1 Classifier + 1 SFF + 2 Proxies + 2 SFs) and a traffic generator (TG), running the two endhosts in separate network namespaces. Both servers were Dell PowerEdge T430, with one Intel Xeon E5-2620 v3 @ 2.40GHz processor and one Intel I350 (quad-port, 1 GbE) NIC each. The DUT and TG were equipped with 64GB and 16GB of RAM, respectively.

The virtual infrastructure consists on 2 OVS bridges: $brsfc$ and $brpxy$. $brsfc$ provides all communication between SFC elements, while $brpxy$ is used to isolate proxy traffic to and from SFs. Finally, to connect DUT and TG two physical DPDK interfaces were added to $brsfc$. OpenFlow rules were used to send all incoming packets from these interfaces directly to the Classifier and to send packets out of the SFC domain after end of chain execution. Hugepage allocation to DPDK applications and RAM allocation to VMs was done according to the needs of each instance, as shown in Table I.

TABLE I: Distribution of resources

	Total Mem	DPDK Mem
OvS (host)	32GB (host)	8GB
Classifier	4GB	2GB
Forwarder	8GB	6GB
Proxy 1	4GB	2GB
Proxy 2	4GB	2GB
SF1	2GB	-
SF2	2GB	-

V. Experimental Validation

This section describes the functional and performance tests executed to evaluate the PhantomSFC prototype.

A. Functional Test

To create a visual representation of packets going through the SFC in Fig. 3 (SF1→SF2), traffic was monitored at four different points: transmission at the sender, SF1 and SF2; and reception at the receiver. The UDP traffic, with 1448-Byte packet size, was generated using iperf3, going from Host A to Host B at a constant rate of 900Mbps during approximately

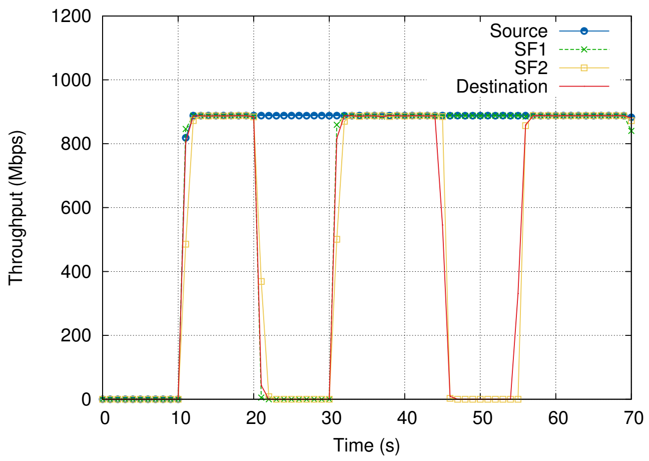


Fig. 6: Functional Test

60 seconds. If packets were indeed going through the SFs, we should see the same amount of traffic at each monitored point.

Additionally, to verify that SFs were executed in the correct order, during the experiment SF1 and SF2 were turned off for a short period of time and then turned on again, one at a time. This same test was repeated five times, and the average traffic at each monitored point during the test is shown in Fig. 6.

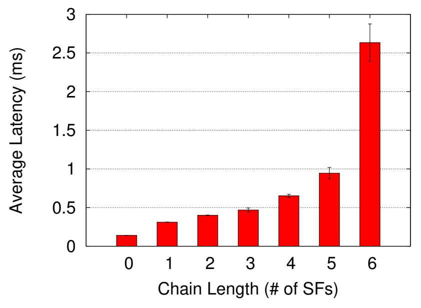
From Fig. 6 we can see that traffic is normal on all interfaces when both SFs are up on interval $(10, 20)$. During $(20, 30)$ SF1 is turned off, thus, the only traffic being shown are packets being sent from the source. This is expected, since SF1 is not forwarding packets to SF2 and the destination. Similarly, on $(45, 55)$ SF2 is turned off, but this time traffic still comes out of SF1, because it comes first in the chain, but no traffic is seen at the destination. This shows that packets were indeed steered through each SF in the expected order.

B. Performance Tests

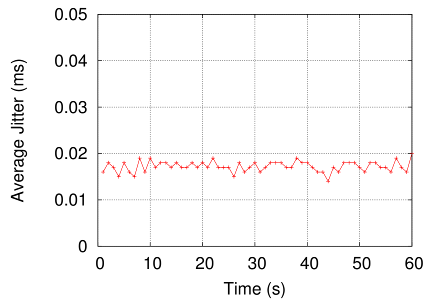
To investigate how PhantomSFC impacts end-to-end performance, we analyzed three metrics: latency growth by increasing chain length, jitter, and packet loss for different throughput rates. The bars in the graphs represent standard error.

Fig. 7a shows latency values measured with ping tool for different chain lengths. Each extra SF added to the infrastructure ran the same forwarding script as before, and was attached to a dedicated SFC Proxy. From Fig. 7a we can see that latency grows in a close-to-linear fashion for up to five functions in the chain, but a steep increase follows the addition of the sixth SF. This is due to the fact that all VMs in the prototype were allocated to the same physical server, competing for resources. Note that this bottleneck can be solved if the SFs and SFC components are placed in multiple servers in such a way to avoid the saturation of the physical host.

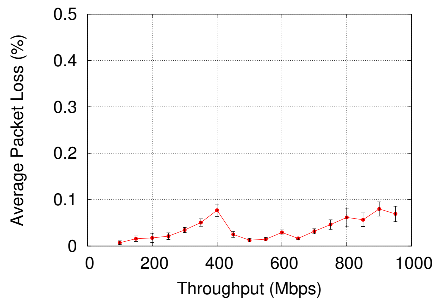
Jitter and packet loss were both measured for a chain composed of two SFs. Fig. 7b shows jitter measurements taken over a $60s$ interval of traffic being sent between two hosts at 900Mbps using iperf3. The maximum jitter variation observed was of $6\mu s$. In addition, we were able to achieve a packet loss smaller than 0.08% for throughput rates up to 950Mbps using 1448-Byte UDP packets. The remaining loss can be further



(a) Average latency vs. chain length.



(b) Average jitter at 900Mbps.



(c) Throughput vs. packet loss.

Fig. 7: Performance tests

reduced with a finer tuning of buffer sizes used in the SFC elements and distribution of VMs across multiple servers to lessen resource contention.

VI. Conclusion

Our proposal, called PhantomSFC, introduced a paradigm shift and a novel way of performing SFC. Current solutions are tightly coupled to specific frameworks and technologies, and require a reasonable amount of integration with the elements of the underlying network. PhantomSFC, on the other hand, shows that it is possible to implement SFC in a network agnostic manner by virtualizing the SFC components described by RFC 7665 and using NSH encapsulation.

The benefit of virtualizing the SFC components is twofold. Firstly, it enables the scalability of resources allocated to SFC execution according to service demands. Secondly, it allows the decoupling between the service plane and the dataplane implementation, enabling the SP to offer its tenants full control over the SFC mechanisms. However, these virtualization benefits comes at a performance cost. To tackle this issue, PhantomSFC implements SFC components using DPDK to enable fast packet processing and kernel bypass.

We described our proposal, explained design decisions made during the implementation of a PoC, and evaluated a prototype that replicates a common SFC scenario in an SP's environment. The experimental results show that our idea is feasible, and our prototype can handle SFC operation with reasonable performance, considering latency, jitter, and throughput.

We plan to extend this work considering the following areas: (i) DPDK application: use multiple CPU cores and NIC queues for packet processing; (iii) prototype: distribute instances across multiple servers, implement distributed SFC components for load balancing, replace OVS by SR-IOV [25] in the communication between VMs and NICs, and extend performance tests; (iv) control plane: define and implement a SFC control plane; and (iv) SFC functionality: add support for NSH metadata, and loops and branches in chains.

References

[1] P. Quinn and J. Guichard, "Service function chaining: Creating a service plane via network service headers," *Computer*, vol. 47, no. 11, pp. 38–44, 2014.

[2] M. Xia *et al.*, "Optical service chaining for network function virtualization," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 152–158, 2015.

[3] W. John *et al.*, "Research directions in Network Service Chaining," in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, 2013.

[4] S. Sahhaf *et al.*, "Network service chaining with optimized network function embedding supporting service decompositions," *Computer Networks*, vol. 93, pp. 492–505, 2015.

[5] H. Kitada *et al.*, "Service function chaining technology for future networks," *NTT Technical Review*, vol. 12, no. 8, 2014.

[6] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," IETF, RFC 7498, 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7498.txt

[7] J. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," IETF, RFC 7665, 2015.

[8] P. Quinn, U. Elzur, and C. Pignataro, "Network service header (nsh)," IETF, RFC 8300, 2018.

[9] OpenDaylight, "OpenDaylight Service Function Chaining Overview," 2017. [Online]. Available: http://docs.opendaylight.org/en/stable-nitrogen/user-guide/service-function-chaining.html

[10] Openstack, "OpenStack Open Source Cloud Computing Software," 2017. [Online]. Available: https://www.openstack.org/software/

[11] Fd.io, "FD.io - The Fast Data Project," 2017. [Online]. Available: https://fd.io/

[12] DPDK, "Data Plane Development Kit," 2017. [Online]. Available: http://www.dpdk.org/

[13] R. Mijumbi *et al.*, "Management and orchestration challenges in network functions virtualization," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, 2016.

[14] A. Gember-Jacobson *et al.*, "OpenNF: Enabling Innovation in Network Function Control," in *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*, 2014, pp. 163–174. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2619239.2626313

[15] J. Gil Herrera and J. F. Botero, "Resource Allocation in NFV: A Comprehensive Survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[16] S. Homma *et al.*, "Analysis on Forwarding Methods for Service Chaining," IETF, Internet-Draft, 2016.

[17] F. Clad *et al.*, "Segment Routing for Service Chaining," Work in progress, IETF, Internet-Draft, 2018.

[18] C. K. Dominicini *et al.*, "VirtPhy: Fully Programmable NFV Orchestration Architecture for Edge Data Centers," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 817–830, Dec 2017.

[19] OpenDaylight, "OpenDaylight," 2017. [Online]. Available: https://www.opendaylight.org/

[20] FD.io, "Vector Packet Processing (VPP)," 2018. [Online]. Available: https://fd.io/technology/

[21] S. Gallenmuller *et al.*, "Comparison of frameworks for high-performance packet IO," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, pp. 29–38.

[22] Ntop, "PF_RING," 2017. [Online]. Available: https://github.com/ntop/PF_RING

[23] R. Luigi, "The Netmap Project," 2017. [Online]. Available: https://github.com/luigirizzo/netmap

[24] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[25] Y. Dong *et al.*, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.