A Standardized Southbound API for VNF Management

Junaid Khalid, Mark Coatsworth, Aaron Gember-Jacobson, Aditya Akella
Department of Computer Sciences
University of Wisconsin-Madison
{junaid, coatsworth, agember, akella}@cs.wisc.edu

ABSTRACT

Network Function Virtualization (NFV) offers network operators great flexibility toward managing network functions, i.e. in-network appliances such as firewalls, load balancers and NATs. Several frameworks exist to this end; however VNF management is fragmented, and no standard management API exists. As a result, each framework uses a proprietary API which a network function must support to fully realize its benefits. This lack of standardization is a major barrier in the wider adoption of NFV. We propose a standard, framework-agnostic southbound API to facilitate faster adoption of NFV and enable innovation in the design of both management frameworks and network functions.

CCS Concepts

•Networks → Middle boxes / network appliances; Network manageability;

Keywords

Network functions virtualization; service chaining; state management

1. INTRODUCTION

Middleboxes (also known as network functions) are systems that perform sophisticated and often stateful packet processing, e.g. load balancers, caching proxies, intrusion detection systems, etc. In recent years, there has been a growing trend towards network function virtualization (NFV), in which network functions (NFs) run as software on generic compute resources, rather than dedicated hardware [7].

HotMiddlebox, August 22–26, 2016, Florianopolis, Brazil
© 2016 ACM. ISBN 978-1-4503-4424-1/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2940147.2940156

NFV provides network operators a greater degree of flexibility in handling application level performance that was not previously possible with monolithic hardware. One new opportunity is elastically scaling NFs [13, 23]—dynamically provisioning new instances as network traffic load increases, redistributing traffic among them, then freeing these resources when load decreases. Another is dynamically composing service chains [10]—run-time allocation, reordering and deallocation of a sequence of NF instances through which a particular flow traverses. Several frameworks [13, 12, 21, 23, 10, 11, 16, 19] have been proposed to fully achieve the potential of virtualization which is envisioned by NFV.

However, for a NF to work within these frameworks, NF developers are required to modify their code to support a framework-specific API. Lack of a standard API for NFs has been frequently cited as a primary hurdle in the adoption of NFV [2, 18]. This has made it difficult not only for network operators who stand to benefit from this technology, but also for NFV vendors who need to support all possible APIs.

To enable faster adoption of NFV, the European Telecommunications Standards Institute (ETSI) has been working on the standardization of NFV architecture. Their NFV Management and Orchestration (MANO) working group has defined a high level specification [9] for the NFV ecosystem. The MANO group is responsible for managing the complete life cycle of virtual network functions (VNFs), from handling cloud infrastructure to managing VNFs to creating services composed of VNFs. Although the MANO specification defines a high level API for service chaining, the lack of fine-grained details has caused confusion among NF vendors. As a result, vendors are coming up with their own interpretations. In addition, the MANO specification does not provide any standard for handling the internal state of NFs.

We argue that most proposed APIs for various aspects of NFV orchestration require VNFs to provide the same basic functionality. By standardizing this functionality and exposing a common management API, a wide range of orchestrators will be able to manage an NFV deployment. We argue the standard should be detailed enough to not cause any confusion, but also general enough to not dictate any specific implementation. We decouple the intention of op-

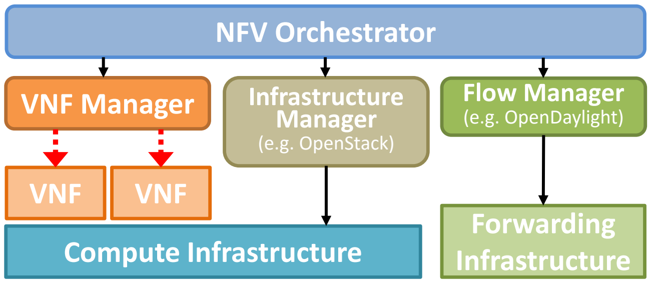

NFV Orchestrator
VNF Manager
Infrastructure Manager (e.g. OpenStack)
Flow Manager (e.g. OpenDaylight)
VNF
VNF
Compute Infrastructure
Forwarding Infrastructure

Figure 1: The red dotted arrows show the proposed API in the NFV architecture.

erations from the functional mechanisms that perform them. We leave the implementation of an underlying core mechanism to support the API operation to the NFs.

In this position paper, we propose the definition of a standard, framework-agnostic API to enable flexible control over NFs by decoupling the core mechanisms from the operational intent. Decoupling of intent from the implementation of mechanism enables NF vendors and management framework developers to innovate independently. In the rest of this paper, we start by enumerating the requirements for a generic and framework agnostic API, then we give an overview of existing NF management frameworks, followed by the proposed API design.

2. REQUIREMENTS FOR A STANDARD API

Based on the use cases of interest for NFV described in [8], we identify four core operations which a generic, framework-agnostic API should provide. Those four operations are described as follows:

- **State Management:** NFs are stateful: they create and manage internal state while processing network traffic. This state is critical to their functionality and must be made available when network traffic is moved from one instance to another [12, 13, 23]. State handling is necessary to support functions like elastic scaling (dynamically spinning up a new instance and redistributing traffic among these instances) as well as resource consolidation (moving all network traffic to a powerful instance to use resources more efficiently). Some frameworks [12, 13, 24, 22] also rely on control over the internal state to provide fault tolerance.
- **Service Chaining:** Network operators depend on NFs in service chains to meet high-level policy objectives such as performance, resource monitoring and security objectives [10, 21, 11]. For example, an ingress policy can dictate that all incoming web traffic passes through a firewall, followed by a caching proxy, and then a load balancer. Additionally, network operators may also want to dynamically add and remove NFs from service chains in response to changing traffic conditions. For example, a deep packet inspection (DPI) engine needs to be dynamically chained after a firewall, so it can further analyze traffic if the firewall detects an alarm. Similarly, a NF should be removed when it is no longer needed in order to reduce resource wastage and packet latency.
- **Bottleneck Detection:** Usage of network, CPU and memory resources by virtualized NFs varies a lot based on the type and configuration of NFs [5, 3, 6]. Bottleneck detection is required to help frameworks in making insightful decisions about chaining and scaling. Inefficient chaining or scaling can result in missing performance objectives and wasting resources.
- **Configuration:** Prior work [14] has looked at the standardization of configuration for NFs which is out of the scope for this paper.

3. EXISTING FRAMEWORKS

Researchers have proposed several mechanisms [13, 23, 12, 10, 21, 11, 16, 19] to enable flexible control over NFs. Unfortunately, these existing frameworks only address one or two of the requirements outlined in Section 2.

3.1 State Management

There are several frameworks [13, 23, 12, 16], designed to handle state by transferring and sharing it among different NF instances. These frameworks may differ in terms of how they track and move state around, but they all use *flowspace*: a set of packet header fields which define a particular flow, to request and filter state. To share and transfer state efficiently and avoid any possible bottleneck, most of them perform these operations in a peer to peer fashion.

These frameworks do differ with respect to how they handle in-transit packets. In Split/Merge [23], to migrate a flow, the controller suspends the traffic arriving at the NF. The packets arriving at the switch are temporarily buffered at the controller. Once the state is migrated, the controller updates the forwarding rules in the switch and injects any buffered packets in the switch. Any packet received by the old instance while migration is happening are dropped. OpenNF [13, 12] uses a similar approach: packets arriving at the old middlebox instance during a state transfer are buffered at the middlebox instance instead of being dropped. The old instance forwards the in-transit packets, along with the state, to the new instance. To mark the last in-transit packet, OpenNF injects a tracer packet in the switch after updating the forwarding rule. When the old NF receives the tracer packet, it signals to the new instance there are no more in-transit packets.

3.2 Service Chaining

Most of the service chaining frameworks [19, 11, 10, 21] rely on routing to steer traffic through a set of NFs in a specific order. These rely on software defined network (SDN) controllers to install fine grained forwarding rules that enable steering of traffic. This is inefficient. Most switches have limited space to store forwarding rules. Moreover, NFs may modify the packets, so there is no guaranteed one-to-one correlation between input and output traffic. A packet modified by a NF might no longer follow the forwarding rules and hence make it difficult to ensure service chaining.

SIMPLE [21], FlowTags [10] and NSH [15] solve these problems by adding extra tags to packets to provide additional necessary information. Tag based routing is used to reduce the required number of forwarding entries in the table at the switch. To enforce policies and ensure that packets traverse the correct set of NFs despite packet mangling NFs on the path, SIMPLE computes a similarity-based correlation at the controller to correlate input traffic with output traffic. FlowTags argues for integrating the tag generation and consumption, along with tracking the relationship between input and output traffic, into NFs. While SIMPLE enables steering traffic through a specific set of NFs to compose service chains, it does not provide the capability to dynamically add or remove a NF to/from the service chain. Whereas FlowTags also provide support for dynamically adding and removing network instances from a service chain. NSH specifies a protocol for creating and updating service chains, but it does not provide any details of an API for communicating with the NFs.

An MPTCP based approach is proposed [19] to add and remove a NF in an end to end path. They insert or delete a NF by creating a new path and tearing down the old one. This approach moves the task of selecting the service chain to the end host.

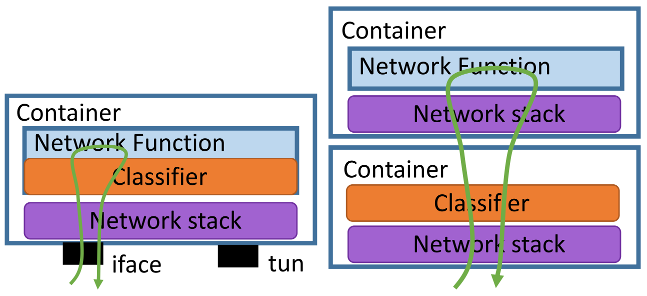
3.3 Bottleneck detection

Resource consumption of virtual network functions (VNFs) varies with the type of NF, traffic workload characteristics and configuration [5, 3, 6]. Techniques used in detecting CPU bottlenecks in traditional cloud infrastructure [1] cannot be employed in VNF settings. Metrics like CPU utilization or memory usage can be deceptive in detecting resource bottlenecks in case of VNFs. Researcher have shown that increased processing delay is a better indicator for bottleneck detections s [26, 25]. When a NF becomes a bottleneck, the packet processing time increases because of queue buildup and longer system calls. Measuring processing delay at the packet level is difficult, because: (1) due to session termination, traffic compression and packet modification, there is no one-to-one correspondence between input and output packet streams; and (2) measuring per packet processing delay adds significant overhead. To overcome these challenges in measuring processing delay, VND [26] analyzes round trip time (RTT) to identify bottlenecks. PerfSight [25] leverages the fact that an increase in processing delay is propositional to queue buildup. It collects and analyzes statistics about different buffers and queues to identify bottlenecks.

We argue that these existing frameworks have enough similarities in their API requirements that a common API can be designed with necessary requirements that support all these frameworks.

4. DESIGN OF APIS

We now present the design of an intent-based VNF API that simplifies the task of managing VNFs. The API allows a VNF Manager (Figure 1) to specify *what* VNFs should do



(a) Integrated classifier (b) Standalone classifier
Figure 2: Proposed network function architecture

with state and traffic, rather than *how* the state and traffic should be passed between VNFs. This allows an NF vendor, which knows its NF best, to select the most appropriate mechanisms for managing the NF's state, chaining the NF, and quantifying the NF's performance. Correspondingly, the VNF Manager can focus on making decisions that are best for the overall VNF deployment, without worrying about the intricacies of individual NFs. Furthermore, decoupling intent and mechanism enables NFs and the VNF Manager to evolve independently.

While our API would ideally be completely NF-agnostic, there are certain aspects of state management, service chaining, and bottleneck detection that are inherently tied to an NF's type and implementation. For example, the annotations that an NF may add to packets of a flow, to influence which downstream NFs process the flow, are dependent on the type of NF: e.g., a caching proxy may mark whether a request was a cache hit or miss, while an IDS may mark whether the flow matches the signature of a known vulnerability. Thus, while the functions contained in our API are common to all NFs, the values of some arguments and the contents of some responses are unique to individual NFs. To aid an operator in configuring a VNF Manager to appropriately specify/interpret these NF-specific arguments/responses, we expect NFs that implement our API to provide an informational document that contains a natural language description of the semantics of NF-specific arguments/responses. Table 1 shows an example of what this document contains. We discuss this in more detail below.

4.1 Service Chaining

Service chaining has two aspects: (1) setting up a sequence of NFs, and (2) providing contextual information for a packet/flow to another NF in a chain.

A naïve approach for setting up a chain of NFs is to have the VNF Manager disseminate the service chain information individually to each NF. However, this approach is inefficient. In cases when the NF mangles the packet header, the VNF Manager has to get the new flowspace identifier (e.g., IP address or MPLS label) of the output packet from an NF and relay it to the subsequent NF [21]. We believe the VNF Manager should just be responsible for providing the list of NFs in a service chain to the first NF in the service chain, and the first NF should be responsible for coordinating with

	Semantic information	Description
Service chaining	{label = status, values=[hit, miss]}	Flow's cache hit/miss status
Resource bottleneck	CPU usage	CPU utilization
	Per pkt. processing delay	Processing delay (μsec) measured using VND [26]

Table 1: An excerpt from a vendor's specification document of a caching proxy

the rest of the NFs to set up the service chain. Offloading the process of setting up service chains to NFs makes the control protocol for setting up the service chains independent of the VNF Manager[1] NF vendors can implement one or more protocols of their choosing (e.g., NSH [15]), with the caveat that all NFs in a chain must have at least one protocol in common. As part of our API, we provide a function that allows a VNF Manager to query which chaining protocols an NF supports:

```
list<protocol> getChainingProtocols()
```

The service chain setup API allows the VNF Manager to specify the list of NFs which a flow should traverse along with a *scope* which defines the flowspace. To setup a service chain, we define the following API call:

```
serviceChain(scope, <list>nexthop)
```

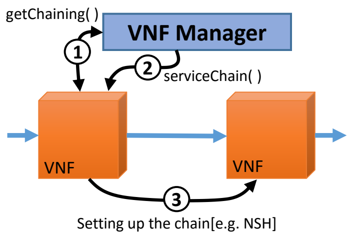
The *list* contains the identifiers of all the hops in the service chain and their order. These can be tags for tunnel endpoints, labels or host names, depending upon the containers' network stacks[2] and underlying network fabric. The calls required for setting up a service chain are shown in Figure 3a

Our API requires each NF to include a logical classifier an essential component of an NF, similar to NSH [15]. Figure 2 shows an overview of the NF architecture with a classifier. The classifier has two responsibilities:

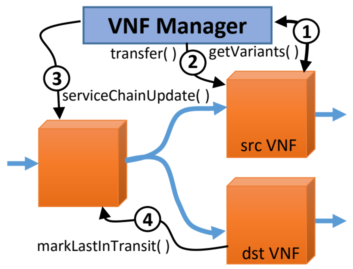
1. Classifier should annotate packets with an identifier (e.g., IP address or tag) for the next hop NF such that the container's network stack and the underlying fabric can route the packet to the appropriate next hop. The identifier to use depends on the implementation of the network stack and fabric: e.g., if the network stack uses IP addresses to determine where to route traffic, then the classifier must encapsulate the packet in a header containing the IP address of the next hop NF; if the network stack uses MAC addresses to determine where to route traffic, then the classifier must put the appropriate MAC address in the packet's Ethernet header.
2. Convey contextual information to other NFs in the chain by marking or tagging packets. This marking is used to transmit contextual information to the subsequent network instances like a signal about the last packet in-transit on the old path after moving the traffic to a new destination or information whether there was a cache hit or miss at the caching proxy.

We envision open-source classifier implementations will be available for NF vendors to integrate into their NFs.

[1]The VNF Orchestrator coordinates with the Flow Manager to install the appropriate forwarding rules.
[2]Container can be a software-container or a VM



(a) Service Chaining



(b) State Management

Figure 3: Interaction of VNF Manager with VNFs

If an NF's classifier does not support the forwarding primitive which the underlying network is using, a network operator/VNF Manager can use a standalone classifier instance as a wrapper to provide support for the required forwarding mechanism as shown in Figure 2b.

To support dynamically reconfigure the service chain by adding or removing NFs, we propose the following API:

```
serviceChainUpdate(scope,
<list>nexthop)
```

To add or remove an NF, the VNF Manager provides the updated list to the NF from which the subsequent service chain needs a modification, and it is the responsibility of that NF to update the service chain by adding or removing the NF.

The second aspect of service chaining is to convey the contextual information generated by an upstream NF to a downstream NF. For example, consider a service chain that has a caching proxy followed by a firewall. The firewall needs to know whether the flow was a hit or miss at the caching proxy, to make its block/allow decision. However, there is not a standard format to communicate the contextual information from one NF to another NF, nor is it possible for a NF to understand all possible inputs. To overcome this challenge, we require: (1) if an NF can consume some contextual information, the NF should consume the context in the form of labels defined in the configuration, and (2) an upstream NF must translate NF-specific contextual information into generic labels which could be consumed by a downstream NF. For example, the operator defines in the configuration of the firewall that the label 21 means a hit and label 22 is a miss. The semantics of the contextual information (i.e.,the type of labels and their acceptable values) which a particular NF can consume or generate must be provided by the NF vendor in the specification document. To translate the contextual information into consumable labels, we leverage our classifier. The VNF Manager adds rules in the classifier of the upstream NF to perform this translation. In our example, the classifier at the caching proxy translate the hit and miss information to their respective labels.

4.2 State Management

While existing state management frameworks require the VNF Manager to oversee state operations [13, 23], we argue that NFs should be responsible for orchestrating state operations, and the VNF Manager should only be responsible for indicating which operation to perform. This prevents the VNF Manager from becoming a bottleneck [12] and allows NF vendors to conduct state operations in a manner that is safe and efficient for their NFs.

Our API provides two functions for expressing state management intents:

```
transfer(destination, scope, variant)
replicate(destination, scope, variant)
```

The former enables a VNF Manager to communicate its intent that one NF instance (*destination*) immediately take over the responsibility of processing a particular set of traffic from another NF instance (*source*). The latter enables a VNF Manager to communicate its intent that an NF instance (*destination*) serve as a hot standby that *could* take over the processing of a particular set of traffic in the event another NF instance (*source*) fails. For both operations we leverage the fact that an NF's state generally pertains to either individual flows or a collection of flows, and we define the *scope* of an operation in terms of a flowspace—i.e., a set of flows identified based on source/destination IPs, transport protocol, source/destination ports, etc.

Operation variants. NF vendors may use any of the mechanisms discussed in Section 3.1, or their own mechanisms, to realize the above operations. For example, an NF may use a combination of moving and sharing state, with various consistency guarantees, to implement a transfer operation (Figure 3b shows the calls for transferring state.). Similarly, to implement replicate, an NF may copy state (or state deltas) after processing every packet, every n packets, packets with special flags (e.g., SYN and FIN packets), etc. For a given NF, different combinations of mechanisms have different trade-offs between NF performance, accuracy, and resource consumption. In other words, an NF vendor may implement multiple *variants* of transfer and replicate to give VNF Managers the flexibility to chose the variant that best meets a network operator's goals.

To help a VNF Manager choose the appropriate variant of an operation, each NF is expected to provide a list of variants and a numeric measure of the performance, accuracy, and resource consumption of each variant:

```
map<variant,[int,int,int]>
getVariants(operation)
```

The numeric measures can be compared across an NF's variants to determine the *relative* performance, accuracy, and resource consumption of a particular variant. Our API does not stipulate how performance, accuracy, and resource consumption are quantified, nor is there an expectation that the values provided by the NF are linearly related. For example, a variant whose performance and resource consumption measures are 2 and 1, respectively, may be ten times faster and consume half the resources of a variant whose performance and resource consumption measures are 1 and 2, respectively.

Handling traffic. In addition to making state available, a VNF Manager must also update the classification behavior of neighboring NFs in the chain such that the set of traffic identified in the `transfer` operation is sent to the destination NF, rather than the source NF. The `serviceChainUpdate` function (Section 4.1) can be used for this purpose. However, following the chain update, the destination NF may need to know that all in-transit traffic matching `scope` has been processed by the source NF or redirected to the destination NF [13]. Thus, our API includes a function that allows the destination NF to request that the classifiers of the neighboring NFs in the chain mark the last in-transit packet to source NF:

```
markLastInTransit(source, scope)
```

4.3 Resource Bottleneck

To support innovative and NF-specific mechanisms of detecting bottlenecks, we propose an API that is general enough to accommodate a variety of resource bottleneck indicators. Our API returns a list of key-value pairs with information about the performance/resource consumption of the NF, similar to SNMP:

```
map<metric,int> queryMetrics()
reportMetrics(interval, callback)
```

We leave the type of information, the technique for measuring and the granularity up to the NF vendors, as they know what indicators are best for their NFs. Vendors define the semantics of their metrics in the NF-specific information document we discussed at the beginning of this section.

The API supports both periodic event and polling based requests. In the case of a periodic event, the framework registers an event handler with the NF. The interval between two events can be either based on the number of packets (e.g. generate an event after the n^{th} packet) or it can be after fixed time intervals (e.g. generate an event after the k^{th} second). In the case of a polling based request, the framework actively polls the NF for the values of its metrics.

5. IMPLEMENTATION

Several implementation challenges exist for a standardized southbound API. VNF software must be modified to implement and expose the southbound API; moreover, VNF Manager must be modified to use these calls. These are not necessarily hard problems, but they have real world implications that affect the adoption of a new API.

5.1 Modifications to VNFs

In order for VNF software to adhere to a specific, standard management API, vendors must add new code that implements the proposed functions. These functions require a detailed analysis of internal application state and configuration, which can differ considerably between different VNF products.

Our previous work on StateAlyzr [17], and others [24] demonstrates that program analysis tools are well suited to this task. These tools can complete most of the work, which removes the menial engineering process and allows human developers to focus on the specific business logic for each NF.

5.2 Modifications to VNF Managers

The applications that communicate with instances must be able to speak the same language of the API. The NFV MANO specification describes an architecture in which VNF Managers are responsible for this management. Existing management frameworks such as OpenMANO [20] or CORD [4] can be extended to handle the calls we propose.

Fundamentally, all of these frameworks must have some mechanism for communicating with VNFs. Many solutions exist at present, but they are all fragmented and proprietary. Our proposed API allows all VNF software to communicate via a common interface, which managers only have to implement once before realizing all the gains of working in a standardized software space.

6. SUMMARY

Our standard API will make it easier for VNF vendors to implement a common set of functions which many VNF managers and other software orchestrators will be able to use. This is a critical step towards the widespread, production adoption of NFV. This also paves a path for the adoption of state management and service chaining frameworks.

7. REFERENCES

[1] AWS Auto Scaling. http://aws.amazon.com/autoscaling.

[2] Banerjee, A. Featured guest article: The commercialization of SDN and NFV. https://sdxcentral.com/articles/contributed/agile-virtualization-commercialization-ari-banerjee/2016/03.

[3] Beyene, Y., Faloutsos, M., and Madhyastha, H. V. SyFi: A systematic approach for estimating stateful firewall performance. In *PAM* (2012).

[4] CORD. http://opencord.org/.

[5] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward predictable performance in software packet-processing platforms. In *NSDI 12* (2012).

[6] Dreger, H., Feldmann, A., Paxson, V., and Sommer, R. Predicting the resource consumption of network intrusion detection systems. In *RAID* (2008).

[7] ETSI. Network functions virtualisation. https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[8] ETSI. Network functions virtualisation (NFV); use cases. http://etsi.org/deliver/etsi_gs/nfv/001_099/001/01.01.01_60/gs_nfv001v010101p.pdf, 2013.

[9] ETSI. Network functions virtualisation (NFV); management and orchestration. etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf, 2014.

[10] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *NSDI* (2014).

[11] Gember, A., Krishnamurthy, A., John, S. S., Grandl, R., Gao, X., Anand, A., Benson, T., Akella, A., and Sekar, V. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Tech. rep., University of Wisconsin-Madison, 2013.

[12] Gember-Jacobson, A., and Akella, A. Improving the safety, scalability, and efficiency of network function state transfers. In *HotMiddlebox* (2015).

[13] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. OpenNF: Enabling innovation in network function control. *SIGCOMM* (2014).

[14] IETF. Nec's simple middlebox configuration (simco). https://tools.ietf.org/html/rfc4540, 2006.

[15] IETF. Network service header. https://tools.ietf.org/id/draft-quinn-sfc-nsh-07.txt, 2015.

[16] Kablan, M., Caldwell, B., Han, R., Jamjoom, H., and Keller, E. Stateless network functions. In *HotMiddlebox* (2015).

[17] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., and Akella, A. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr. In *NSDI* (2016).

[18] Mishra, S. NFV orchestration: Challenges in telecom deployments. youtube.com/watch?v=YHJQpdjCFJ.

[19] Nicutar, C., Paasch, C., Bagnulo, M., and Raiciu, C. Evolving the internet with connection acrobatics. In *HotMiddlebox* (2013).

[20] OpenMANO. github.com/nfvlabs/openmano.

[21] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM* (2013).

[22] Rajagopalan, S., Williams, D., and Jamjoom, H. Pico Replication: A high availability framework for middleboxes. In *SoCC* (2013).

[23] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI* (2013).

[24] Sherry, J., Gao, P., Basu, S., Panda, A., Krishnamurthy, A., Macciocco, C., Manesh, M., Martins, J., Ratnasamy, S., and Shenker, L. R. S. Rollback recovery for middleboxes. In *SIGCOMM* (2015).

[25] Wu, W., He, K., and Akella, A. PerfSight: Performance diagnosis for software dataplanes. In *IMC* (2015).

[26] Wu, W., Wang, G., Akella, A., and Shaikh, A. Virtual network diagnosis as a service. In *SoCC* (2013).