Scylla: A Language for Virtual Network Functions Orchestration in Enterprise WLANs

Roberto Riggio*, Imen Grida Ben Yahia§, Steven Latré‡, Tinku Rasheed*
*CREATE-NET, Trento, Italy; E–Mail: rriggio,trasheed@create-net.org
§Orange Labs, Paris, France; E–Mail: imen.gridabenyahia@orange.com
‡University of Antwerp, Antwerp, Belgium; E–Mail: steven.latre@uantwerpen.be

***Abstract*—Network Function Virtualization (NFV) is set to disrupt the current networking ecosystem by turning vertically–integrated middleboxes into software modules running on general purpose virtualized platforms. NFV will play a key role in future wireless and mobile networks where significant cost reductions can be obtained by virtualizing different layers and functions of the radio access and core network. Such goal raises several challenges in terms of both functional decomposition of the radio nodes and for the management and orchestration of the resulting network. In this work we present *Scylla* a high–level declarative language for programming network functions that allows programmers to implement per–flow custom packet processing. We also introduce a set of programming abstractions modeling the fundamental aspects of VNF orchestration. Finally, we present a proof–of–concept Controller and an SDK implementing the proposed abstractions.**

I. Introduction

Network operators are currently transitioning from hardware–based middlebox models, where network functions, such as firewalling, load balancing, and caching, are implemented as vertically integrated solutions, to Network Function Virtualization (NFV) where the same operations are performed by software instances running on general purpose virtualized networking and computing infrastructures.

Current SDN/NFV solutions already allow operators to dynamically deploy network functions as virtual machines (VMs) and to steer traffic through them using OpenFlow. Nevertheless, the progressive process of network softwarization enabled by NFV allows for a dramatic refactoring of network functionalities moving beyond the architectural limitations imposed by physical middleboxes. NFV enables network operators to implement custom packet processing and to improve code reuse and modularity by turning common operations such as filtering, load–balancing, and encryption into *libraries* that can be shared across different network functions.

While several attempts to apply such concept to wireless and mobile networks can already be found in the literature [1], [2], few works address the programming abstractions for NFV orchestration in wireless networks. Likewise, Radio Access Network (RAN) and core network have been treated, for the most part, as two separate problems despite the fact that resource allocation decisions in the RAN affect the status of the core network and vice–versa.

Research leading to these results received funding from the European Union's H2020 Research and Innovation Action under Grant Agreements H2020-ICT-644843 (VITAL) and H2020-ICT-671639 (COHERENT).

978-1-5090-0223-8/16/$31.00 © 2016 IEEE

In this work we present *Scylla* a high–level declarative language for programming network functions that allows programmers to implement per–flow custom packet processing as well as the associated management and orchestration logic. *Scylla* blurs the line between radio access and core network by introducing the concept of *Programmable Network Fabric* (*PNF*). The *PNF* builds upon a single platform consisting of general purpose hardware (e.g., x86) and operating systems in order to deliver three types of virtualized network resources, namely: forwarding nodes (i.e., OpenFlow–enabled switches), packet processing nodes, and radio processing nodes.

Scylla allows to deploy complex network services by chaining different packet processing functions, named as *Light Virtual Network Functions* (*LVNF*). In an Enterprise WLAN this allows for example to offload part of the IEEE 802.11 MAC to *LVNF*s that can be dynamically scaled according to the network load. Examples include *Probe Request*s offloading, to provide protection against common IEEE 802.11 Denial of Service attacks [3], [4], as well as *Duplicates Filtering* to ease the load on the backhaul in highly noisy environments where a significant number of retransmission are common.

This paper extends our previous work on programming abstractions for Software–Defined Wireless Networks [5] with additional primitives for VNF orchestration. The proposed abstractions tackle VNF state management, layer management, and network state collection. We realized these abstractions in a proof–of–concept *Programmable Network Fabric Controller* (*PNFC*) and a Python–based Software Development Kit (SDK). Although our discussion will focus mainly on WiFi due to the fact that the proof–of–concept currently supports only this technology, we believe that the proposed abstractions can meet the requirements of current and future mobile networks. The entire software stack, including: data–path, *PNFC*, and SDK are released under BSD license[1].

The reminder of this paper is structured as follows. Section II discusses the related work. The programming abstractions together with the rationale behind their design are introduced in Sec. III. The proof–of–concept implementation details and the *Scylla* SDK are presented in, respectively, Sec. IV and Sec. V. Section VI reports on the field evaluation. Finally, Sec. VII draws the conclusions pointing out future research directions.

[1]http://empower.create-net.org/

II. Related Work

Standardization. The European Telecommunications Standards Institute (ETSI) has recently tackled the NFV concept [6] while the OPNFV project [7] is working towards an open source carrier grade platform for NFV Management and Orchestration. OpenMANO [8] and OpenBaton [9] have similar goals. In parallel, the IETF Service Function Chaining (SFC) working group is investigating various aspects of SFC in the mobile and in the data–center networking domains [10].

Data–planes. State–of–the–art solutions on programmable data–plane focus on improving raw packet processing speed [11], [12], [13], [14] but do not tackle VNF orchestration nor they provide a viable NFV control plane. Similarly, the concept of making wireless systems more modular has been investigated in several works like OpenRadio [2] and MAClets [1]. *Scylla* on the other hand provides network programmers with a set of high–level programming abstractions for VNF management and orchestration. Such abstractions allow network developers to specify the logical sequence of VNF as a certain flow must traverse leaving to a runtime system the task of configuring the network.

Cloud Computing Platforms. Traditional cloud computing platforms such as OpenStack [15] have not been designed with VNF orchestration in mind and, as a result, their API fails to capture its requirements. For example, VNFs deployed using traditional VMs only allow cloning as migration method. This results in a significant waste of memory/networking resources in that unneeded state has to be migrated alongside the actual VNF [16]. Same considerations also apply to solutions based on Docker and/or process migration.

Middlebox Management. Systems like OpenNF [17] and its derivatives [18], [19] focus on providing a platform for consistent VNF migration, however their focus is on maintaining backward compatibility with currently available VNFs such as Bro [20] for IDS and Squid [21] for caching web proxies. Conversely in this work we set to explore the possibilities opened by a clean–slate design which encourages code reuse and modularity. Similar considerations can be made also for Split/Merge [22], CoMB [23], and XoMB [24]

Programming Abstractions. Many works have recently focused on programming languages for SDNs [25], [26], [27], [28], [29], [30], [31], [32]. Most of these languages do provide high–level interfaces to an OpenFlow–enabled network, however they do not provide mechanisms for managing or orchestrating VNFs, nor do they support VNF migration.

VNF Placement. The amount of literature on components placement and on virtual network embedding (VNE) is extensive. Fundamental works on VNE include VINEYard [33] and PolyVINE [34]. For a comprehensive survey on VNE algorithms we point the reader to [35]. Similarly, the amount of literature on component placement is humbling [36], [37], [38]. In [39] a joint node and link mapping algorithm is proposed. While the problem of dynamic VNF placement in wired and wireless networks is tackled in [40], [41] and [42] respectively. *Scylla* complements the works above by providing a programmable control plane for VNF orchestration.

III. Requirements and Design

A. Overview

In this section we set to identify the common aspects of VNF orchestration. Notice how, throughout the paper, we will use the term *Network Service* to refer to a collection of VNFs and the associated management and orchestration logic. It is worth stressing that, in this work, we do not aim at proposing novel VNF placement solutions. Conversely, we aim at identifying the invariants of NFV management and orchestration clearly separating policies from mechanisms and at putting the former in the hands of *Network Service* developers through a set of high–level and open primitives.

The abstractions presented in this paper extend and complement our prior work on control and coordination of wireless and mobile networks [5]. We do so by generalizing the concept of radio access as a generic VNF and by extending the programming model with new primitives for NFV management and orchestration. In particular, in this paper we address: layer management (i.e. how to deploy a *Network Service*), state management (i.e. how to orchestrate VNFs), and network discovery (i.e. how to expose the network status to the *Network Service* programmers). Let us analyze the requirements imposed by each of them on the abstractions:

- *Layer Management*. Programmers should simply specify the logical sequence of VNF the traffic should traverse, leaving to an underlying runtime system the task of deciding how the traffic should be routed. This is consistent with the recent trends on Intent–based networking where network applications express their goals, e.g. traffic should go through a firewall before leveling the network, rather than specifying how to implement them, e.g. the traffic should be routed along a certain path.
- *State Management*. Programmers shall not be exposed with the details of handling the state of VNFs, nor they should deal with the details of polling the network elements for statistics. Conversely, each VNF shall define the requirements for the orchestration layer which is then in charge of deploying/scaling/migrating these functions according to such constraints.
- *Network Monitoring*. The abstractions shall allow network developers to gather the status of the network functions using high–level querying primitives. Such information shall include network statistics and topology changes, e.g. node going offline or links becoming congested.

In the next subsections we shall introduce the three fundamental programming abstractions introduced by *Scylla* in order to support the above requirements. Figure 1 depicts the relationship between the previously introduced radio abstractions (Light Virtual Access Point, Resource Blocks, Channel Quality Map, and Radio Port) and the NFV abstractions (Light Virtual Network Function, Virtual Port, and Network Graph) using an UML class–diagram. We will now briefly summarize the main features of the radio abstractions pointing the reader to [5] for a more exhaustive description.

We name as Virtual Execution Engines (*VEE*s), all the network elements in the *PNF*, i.e. switches, computing, and radio nodes. We name Wireless Termination Points (*WTP*s) as the physical points of attachment in the RAN (e.g. WiFi

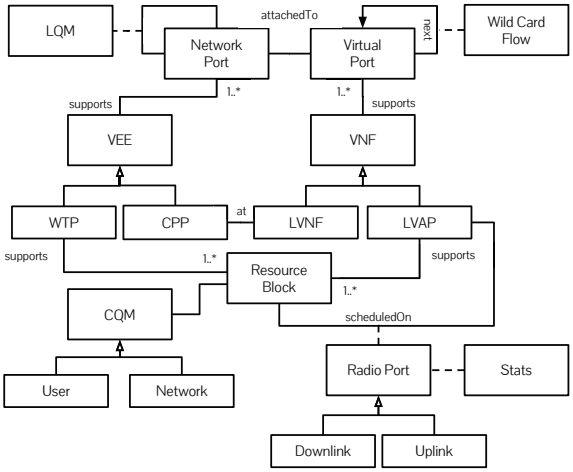


Fig. 1: The programming abstractions object model. Methods and properties are omitted in order to improve readability.

Access Points or LTE eNodeBs) and Click Packet Processors (*CPP*s) the forwarding nodes with computational capacity. These nodes are essentially programmable switches. As the name suggests, *CPP*s leverage on multiple instances of the Click Modular Router [43] in order to implement packet processing. Two types of VNFs are currently supported by *Scylla*, namely the Light Virtual Network Function (*LVNF*), implementing custom packet processing, and the Light Virtual Access Point (*LVAP*), implementing radio access functions.

In the *Scylla* programming model, the *Resource Block* represents the minimum amount of wireless resource that can be assigned to a wireless client. *Resource Block*s are identified by a frequency band, a time interval, and the *WTP* at which they are available. *LVAP*s represent the state of wireless client on a set of *Resource Block*s. *WTP*s and *LVAP*s support a possibly different set of *Resource Block*s. A non–empty intersection between the two sets must exist in order to allow communication. A relationship exists between *LVAP*s and *WTP*s and between *WTP*s modeling the link quality between the two entities. This relationship is captured by the *Channel Quality Map*s. Finally, the *Radio Port* abstraction models the dynamic and reconfigurable characteristics of the link between *WTP*s and *LVAP*s on a set of *Resource Block*s, e.g. the MCS or the transmission power.

B. Light Virtual Network Function

The *LVNF* is a generalization of the *LVAP* abstraction introduced in [5]. However, unlike the *LVAP* abstraction, which was designed in order to provide network developers with a high–level interface for wireless client state management, the *LVNF* abstraction allows arbitrary packet processing blocks to be exposed to the *Network Service* developer trough an high–level declarative interface. Inspired by Click, *Scylla LVNF*s can be composed in a forwarding graph in order to implement complex operations on a specific subset of the traffic.

*LVNF*s are instantiated starting from VNF templates called *Image*s. Each network function corresponds to an *Image* and consists of a Click script together with some additional information such as the number of input/output ports used

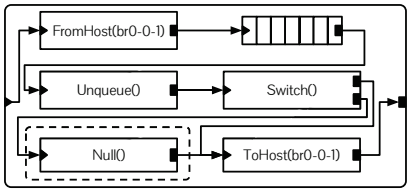


Fig. 2: The complete Click script implementing the Null *LVNF* as deployed on the *CPP*. The user–defined portion is in the dashed box.

by the *LVNF*, and the list of Click handlers[2] exposed by the *Image*. Handlers are used in order to manipulate the internal state of the *LVNF*. For example, in the case of a Firewall *LVNF*, specific handlers shall be defined in order to allow the network operator to add/remove firewall rules. In the ETSI terminology, the handlers essentially provide the interface for the Operator's Element Management Systems (EMS) [6].

Deploying a VNF requires handling several low–level details, such as creating new virtual network interfaces in the *CPP*, configuring the local switch, and start/stopping the software processes implementing the actual network function. Such details are transparently handled by the *Scylla* runtime system. For example consider the following *Scylla Image* which does not perform any modification on the traffic:

```
in_0 -> Null() -> out_0
```

Listing 1: A *Scylla Image* implementing a null network function.

As it can be seen, the *Image* consists of a very simple Click script which uses one input and one output port. Notice how `in_0` and `out_0` are the names of the two Click elements in charge of the network I/O. Upon deployment, the *Image* will be extended by the *Scylla* runtime with the missing elements declaration plus some additional boilerplate code required for VNF migration. The final Click script deployed on the *CPP* is depicted in Fig. 2. The virtual network interface $br0-0-0$ is dynamically created on the target *CPP* before starting the *LVNF*. The interface name is dynamically generated in such a way to avoid collision with other *LVNF*s running on the same *CPP*. As it will be clear in the next subsection, *Network Service* developers need not consider the actual virtual network interface names, instead they can use the *Virtual Port* abstraction in order to declare the sequence of *LVNF* a certain portion of the flow–space shall traverse leaving to the underlying runtime system the task to compute the route and to configure the switches.

*Scylla LVNF*s support different operating modes. Possible events and the corresponding transitions between modes are implemented by the finite state machine (FSM) depicted in Fig. 3. *LVNF* automatically transition to the *Running* mode once deployed on a *CPP* (after the virtual network interface has been created and the click process has been started). In this mode all incoming traffic is processed by the Click script provided by the *Network Service* developer.

There are however situations where it may be useful to pause/resume an *LVNF* as a method, for example, to prevent

[2]Handlers are access points through which users can interact with elements in a running Click router or with the router as a whole.

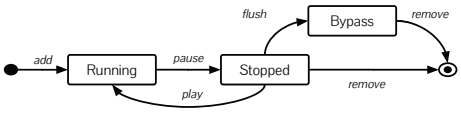


Fig. 3: The *LVNF* finite state machine.

updates to its internal state prior to a migration. Upon receiving the *pause* trigger, the *LVNF* transitions to the *Stopped* state. In this state all incoming traffic is buffered (FIFO) in the *LVNF* internal queue. The capacity of the Queue element used to buffer the incoming traffic is finite (by default it is set to 1000 packets) and when the buffer is full incoming traffic is dropped. Once in the *Stopped* state an *LVNF* can either transition back to the *Running* state or, upon receiving the *flush* trigger, can transition to the *Bypass* State. In this state all incoming traffic is redirected by the Switch element to the output port without further processing. Finally, the *remove* trigger terminates the *LVNF*.

The FSM above allow us to support basic statefull *LVNF* migration. In particular, when an *LVNF* migration is triggered, the runtime will *pause* the execution of the *LVNF* at the source *CPP* causing all incoming traffic to be buffered. The runtime will then spawn a new *LVNF* at the target *CPP* and move the state of the *LVNF* from the source *CPP* to the target *CPP* (details on how the state is moved are given in Sec. V-A). Finally, the forwarding rules in the switching fabric can be updated in order to: (i) redirect the new traffic to the target *CPP*; and (ii) forward the traffic from the source *CPP* to the target *CPP*. This will result in buffered packets at the source *CPP* to be redirected to the target *CPP* when the *flush* method is invoked on the *LVNF* on the source *CPP*.

Albeit this approach guarantees that no traffic is lost during the migration, in–order packet processing is not ensured. Moreover, this migration scheme also results in a slightly increased latency in that packet processing is halted between the *pause* and the *flush* calls. We acknowledge that out-of–order packet delivery as well as the increased latency may disrupt the operations of certain network functions [44]. Nevertheless, we would like to stress how the focus of this work is on the high–level programming primitives exposed to the *Network Service* developers and not on their actual implementation, which we still consider as preliminary.

C. Virtual Port

*LVNF*s, as well as *Image*s, does not define which kind of traffic they should process. *Network Service* developers can define the logical sequence of *LVNF*s a certain portion of the flow–space must traverse, using the *Virtual Port* abstraction. *LVNF*s can have one or more *Virtual Port*s. Each *Virtual Port*, is associated to one, and only one, virtual network interface in a *CPP* which in time is attached one *Network Port* in the local software switch, e.g. an OpenVSwitch instance.

Figure 4 sketches the reference architecture for a *VEE* illustrating the relationship between *Network Port*s and *Virtual Port*s. Notice how, *LVAP*s share the same virtual interface. This derives from the fact that *LVAP*s (which we remind the reader are as many as the number of wireless clients associated to a given *WTP*) are expected to be migrated between *WTP*s more often than regular *LVNF*s. This design choice allow us to avoid adding and removing a port for the local software switch every time a handover occurs.

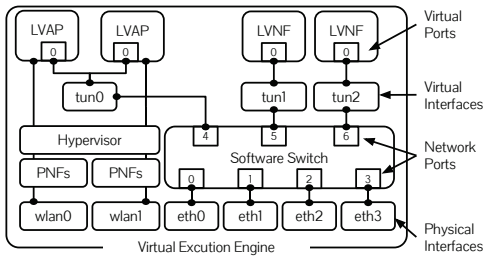


Fig. 4: The reference architecture for a *VEE*.

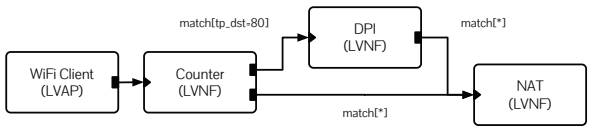


Fig. 5: A simple *LVNF*s chain illustrating the concept of *Virtual Port*.

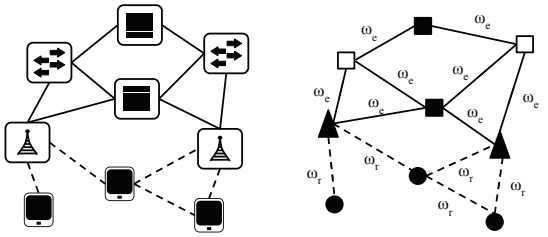
Network Service developers do not need to care about the virtual interface name on a *CPP* . Instead, they can perform *LVNF* chaining by using their *Virtual Port*s. Figure 5 depicts a sample *LVNF*s chain in order to better illustrate the *Virtual Port* concept. In this example, the traffic generated by a wireless client, or more specifically by the *LVAP* mapping a wireless client, is redirected to a traffic monitoring *LVNF* (Counter). Then all HTTP traffic is redirected to a Deep Packet Inspection *LVNF*s (DPI). Finally, all traffic is redirected to a gateway *LVNF* (NAT). The translation between *Virtual Port*s and *Network Port*s as well as the computation of the optimal route and the configuration of the switches is transparently handled by the *Scylla* runtime.

D. Network Graph

The Network Graph extends the *Channel Quality Map* (*CQM*) presented in [5] with the full view of the core network state. In this section we shall briefly summarize the main feature of the *CQM* before moving to the *Network Graph*.

The *CQM* abstraction provides network programmers with a full view of the network state in terms of channel quality between *LVAP*s and *WTP*s over the available *Resource Block*s. Let $G_r = (V_r, E_r)$ be a directed graph, where $V_r = V_{WTP} \cup V_{LVAP}$ is the set of $v_1 = |V_{WTP}|$ *WTP*s and $v_2 = |V_{LVAP}|$ *LVAP*s in the network, and E_r is the set of edges or links. An edge $e^r_{n,m,i} \in E_r$ with $n, m \in V_r$ exists if m is within the sensing range of n over the *Resource Block* i. A weight $\omega_r(e_{n,m,i})$ is assigned to each link $e^c_{n,m,i} \in E_r : \omega_r(e_{n,m,i}) \in \mathbb{N}^+$ modeling the channel quality of the link between the two nodes (e.g. using the RSSI).

In this work we extend the *CQM* with information about the core network. In particular, let $G_c = (N_c, E_c)$ be a directed graph modeling the core network, where N_c is the set of $n = |N_c|$ network elements (*CPP*s and switches) and E_c is the set of edges or links. An edge $e^c_{n,m} \in E_c$ if and only if a point–to–point connection exists between $n \in N_c$ and $m \in N_c$. With respect to the core network, links are actual wiring



(a) Network topology. (b) Network graph.

Fig. 6: Sample physical network (a) and network graph (b).

media, e.g., an Ethernet cable interconnecting the two nodes. A single weight $\omega_e^s(e_{n,m})$ is assigned to each link $e_{n,m}^c \in E_c$: $\omega_e^s(e_{n,m}^c) \in \mathbb{N}^+$ modeling the link load.

Figure 6 depicts a sample radio access and core network and the associated network graph. More specifically, Fig. 6a shows valid links between wireless clients and *WTP*s (dashed lines) as well as wired links interconnecting *WTP*s, *CPP*s, and pure forwarding nodes (solid lines), while Fig. 6b shows the Network Graph associated with the topology.

IV. Implementation Details

A. Overview

The proposed abstractions have been implemented over the *EmPOWER* platform. *EmPOWER* is an SDN/NFV framework consisting in: (i) a *Programmable Network Fabric Controller* (*PNFC*), (ii) a programmable data–plane, and (iii) a Python–based SDK. In this section we describe in detail the first two components while the features of the SDK are described in the next section. Notice that the prototype currently targets only wireless access networks based on the 802.11 family of standards. A high level view of the *EmPOWER* system architecture is depicted in Fig. 7. The architecture is conceptually divided into three layers. The bottom layer consists of the physical and virtualized resources composing the *PNF*. In the second layer we have the *PNFC* which is in charge of the physical and virtual resources available in the *PNF*. Finally, in the third layer we have the actual *Network Service* slices.

Virtual Network Operators use their Operational Support System (OSS) and the Business Support System (BSS) in order to manage and operate their *Network Service* slices. From an architectural standpoint, the *Network Service* slice creation resides in the Orchestrator which is in charge of deciding whether a particular *Network Service* can be accepted or if it must be refused. If a request is accepted, then the Orchestrator is in charge of mapping the request onto the *PNF* by deploying the *LVNF*s on the selected nodes. Information about all supported *LVNF*s and *Network Service*s are maintained in the *LVNF* and *Network Service* catalogs. The Infrastructure Catalog holds information about available physical and virtualized resources. Finally, the list of placement algorithms is held by the Placement Policy Catalog.

B. Programmable Network Fabric Controller

The *PNFC* supports multiple *Network Service*s over the same physical infrastructure. Management and orchestration applications run on top of the *PNFC* in their own slice of resources and exploit the *Scylla* programming primitives through either a REST API or the native Python API. Example of control applications include: mobility management, traffic engineering, and *LVNF* scaling/consolidation. Communications between *WTP*s/*CPP*s and the *PNFC* take place over persistent TCP connections.

C. Programmable Network Fabric

As already mentioned in Sec. III our architecture currently accounts for three kinds of *PNF* resources, namely: basic forwarding nodes (i.e. OpenFlow switches), packet processing nodes (*CPP*s), and radio nodes (*WTP*s).

Each *CPP*/*WTP* includes an OpenVSwitch instance, one or more *LVNF*/*LVAP*, and one Agent. The latter is in charge of monitoring the status of each *LVNF*/*LVAP* as well as of handling the requests coming from the controller. In the current implementation the monitoring features includes: number of packets/bytes transmitted and received on each virtual interface as well as the amount of resources utilized (cpu time, and memory) by each *LVNF*.

*CPP*s are built upon the Soekris 6501-70 platform consisting in single 1.6 GHz Intel Atom CPU, 2 Gbyte if SDRAM, and 12 Gigabit Ethernet Ports. *CPP*s run Ubuntu 15.04 Server as operating system and a patched version of the Click Modular Router [43] supporting element state serialization. *WTP*s exploit the PCEngines ALIX (x86) embedded platform and run the OpenWRT operating system.

V. Software Development Kit

The Python–based SDK introduced in [5] has been extended in order to support the new primitives introduced in Sec. III. In this section we shall briefly summarize some of the SDK's most interesting new features. The list of new primitives can be found in Table I. For a comprehensive list of all the primitives supported by the SDK we refer the reader to [5].

Two types of primitives are supported by the SDK: *Queries* and *Events*. In the former case (*Queries*), *VEE*s are periodically polled for a specific information, e.g. the number of packets transmitted or received by an *LVNF*. In the latter case (*Events*) a thread is created at one or more *VEE*. Such thread is identified by a firing condition, e.g. the CPU Utilization of an *LVNF* going above a certain threshold. When such condition is verified a message is generated by the *VEE*. A termination condition can also be specified.

All primitives are non–blocking and, as such, they immediately return control to the calling network application. An optional *callback* method can be provided to all primitives specifying a method to be executed when the primitive returns a result. A Python dictionary, whose structure depends on the actual primitive, is passed as parameter to the callback.

A. Light Virtual Network Function

The *LVNF* abstraction is exposed to the *Network Service* developers through a Python object mapping properties to operations. These properties are: (i) the *Image* used by the *LVNF*, (ii) the list of *Virtual Port*s supported by the *LVNF*, and (iii) the *CPP* that is hosting the *LVNF*. Programmers can access the *Virtual Port*s supported by an *LVNF* using the `ports` property. Similarly, performing an *LVNF* migration

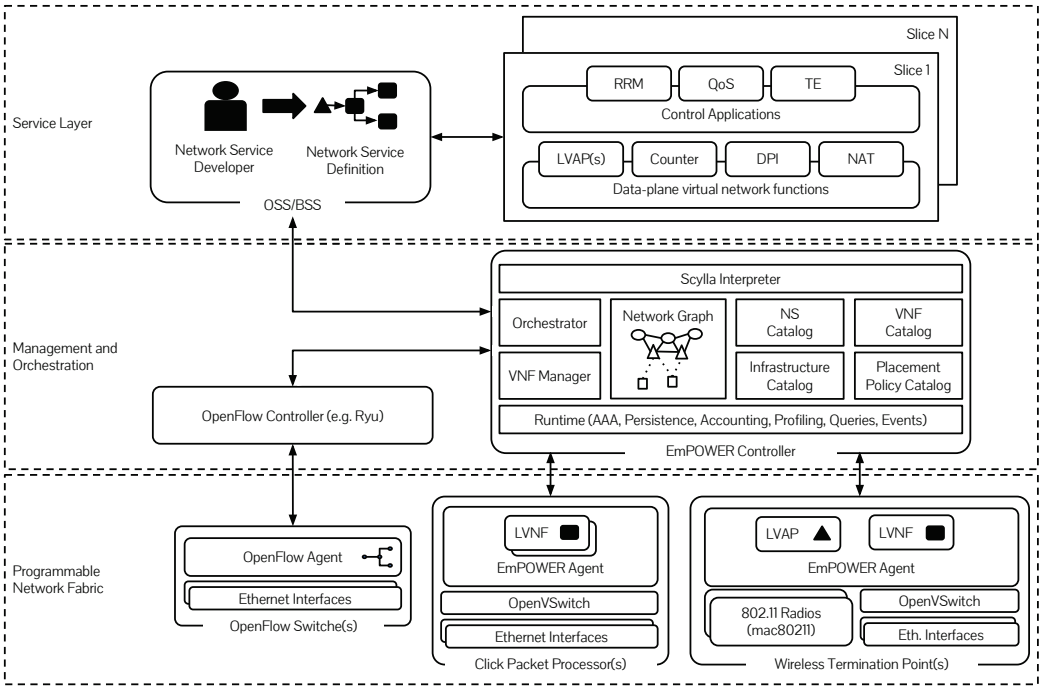


Fig. 7: The *EmPOWER* system architecture.

Primitive	*Parameters*	*Type*	*Description*	*Section*
`packets/bytes_count`	lvnf, every	Query	Packets/bytes transmitted or received by an *LVNF*	V-A
`cpu/mem_util`	lvnf, relation, value	Event	Callback when a condition on CPU/Memory utilization is verified	V-A
`lqm`	addrs, every	Query	Returns packet counters from a *CPP*	V-C

TABLE I: *LVNF* programming primitives in the Python–based SDK.

is a matter of assigning a new *CPP* to the `cpp` property of an *LVNF* object. In the following example a new *LVNF* is spawned at one *CPP* using a previously defined *Image*.

```
# Click script
VNF = """in_0
-> Classifier(12/bbbb)
-> Strip(14)
-> dupe::WifiDupeFilter()
-> WifiDecap()
-> out_0;"""

# Handlers
handlers = [("dupes_table", "dupe.dupes_table")]

# Create Image
img = Image(vnf=VNF, nb_ports=1, handlers=handlers)

# Create and spawn LVNF
lvnf = LVNF(img)
lvnf.cpp = cpp
```

Listing 2: Spawning a new LVNF.

In particular, the listing above implements a de–duplication *LVNF* for 802.11–based WLANs. In legacy systems this network function is typically implemented by the WiFi AP or by the WiFi controller. This *LVNF* checks if the incoming traffic is LWAPP–encapsulated. The Lightweight Access Point Protocol (LWAPP) is a protocol used to transport WiFi traffic over either L2 or L3 PDUs. Non–LWAPP traffic is discarded, while legit frames are stripped of their Ethernet header and passed to the duplicate filtering element (*WiFiDupeFilter*). Unique WiFi frames are converted to Ethernet frames (*WiFiDecap*).

The *Image* constructor allows to define a list of handlers. Each handler is defined as a 2–tuple where the first entry is the handler short–name (e.g. *dupes_table*) while the second entry is the actual Click handler (e.g. *dupe.dupes_table*). As for the other primitives, also *LVNF*s' handlers are non–blocking and requires a callback method to be specified by the *Network Service* developer. The following statement accesses the *LVNF* duplicates table and then calls the specified callback method when the handler returns a result:

```
lvnf.dupes_table(callback=dupes_callback)
```

Listing 3: Accessing an *LVNF* handler (read).

The callback method will receive a Python dictionary as parameter similar to the one reported in the listing below. For each wireless station the following information are reported: (i) the number of duplicate frames, (ii) the number of entries in the list, and the last N sequence numbers.

```
{
    "A0:D3:C1:A8:E4:C3": [215, 3, 1323, 2324, 1325],
    "5C:E0:C5:AC:B4:A3": [1231, 3, 201, 202, 203]
}
```

Listing 4: *LVNF* handler output.

Handlers can also be accessed in write mode allowing the *Network Service* developers to modify the internal state of an *LVNF*. For example, the following listing adds a new entry in the *LVNF* duplicates table.

```
lvnf.dupes_table(value=["a0:d3:c1:a8:e4:c3", \
                        215, 3, 1323, 2324, 1325])
```

Listing 5: Accessing an *LVNF* handler (write).

Migration is supported by leveraging on the Click "hotswaping" feature. Such feature allows to replace the Click configuration with a new one while preserving the state of some elements (e.g. the duplicates table). We extended this feature by allowing Click configurations to be removed from one *CPP* and re–created on another one. This is achieved by serializing the state of the Click elements with a custom serialization function implemented using Boost [45]. This approach allowed us to dramatically reduce the state to be migrated across *CPP*s. The actual amount of state depends on the *LVNF*, and can range from few bytes for simple statistics gathering *LVNF*s to several MBytes for more complex *LVNF*s.

The `cpu_util` primitive allows the programmer to trigger a callback the first time the CPU utilization of an *LVAP* verifies a certain. For example:

```
cpu_util(lvnfs='11:22:33:44:55:66',
         relation='GT',
         value=80,
         tenant_name='Guests',
         callback=cpu_callback)
```

Listing 6: Create an CPU Utilization trigger.

After the trigger has fired the first time and as long as the CPU utilization remains above 80%, the callback method is not called again. Specifying `ff:ff:ff:ff:ff:ff` as `lvnfs` will trigger the callback when the CPU Utilization of any *LVNF*s is above 80%. Similarly, the `mem_util` primitive allows developers to get a callback when the memory utilization of a certain *LVNF* verifies a certain condition.

Finally, the `packets/bytes_count` primitives allow programmers to track the traffic transmitted and received by a certain *LVNF Virtual Port*. Their details have been omitted due to space constraints.

B. Virtual Port

The *Virtual Port*s supported by an *LVNF* can be accessed from the `ports` property. *Virtual Port*s can be concatenated using their `next` property which maps a certain portion of the flow space to another *Virtual Port* (typically belonging to a different *LVNF*). For example the following routine implements the VNF chaining depicted in Fig. 5:

```
# All traffic
other = make_wild_card_flow()
# HTTP traffic
http = make_wild_card_flow()
http['tp_dst'] = 80
# LVNF Chaining
lvap.ports[0].next[other] = lvnf_count.ports[0]
lvnf_count.ports[0].next[http] = lvnf_dpi.ports[0]
lvnf_count.ports[0].next[other] = lvnf_nat.ports[0]
lvnf_dpi.ports[0].next[other] = lvnf_nat.ports[0]
```

Listing 7: LVNF Chaining.

This essentially implements an intent–based interface for VNF composition allowing *Network Service* developers to specify the logical sequence of *LVNF*s the traffic should traverse. The intent is eventually conveyed to a network controller which is responsible for implementing it by computing the optimal route and by configuring the individual network elements (i.e. switches and routers). Currently the *Scylla* runtime supports two network controllers: Ryu [46] and ONOS [47]. In the former case (Ryu) a new module has been developed implementing the intent–based interface. In the latter Case (ONOS) the embedded intent–based interface has been used.

C. Network Graph

The *Network Graph* is exposed to the network programmer by means of three data structures: the *User Channel Quality Map* (*UCQM*), the *Network Channel Quality Map* (*NCQM*), and the *Link Quality Map* (*LQM*). The first two data structures are 3–dimensional matrices where each entry is the channel quality over one *Resource Block* between: an *LVAP* and a *WTP* in the case of the *UCQM*; and between two*WTP*s in the case of the *NCQM*. Similarly, the *LQM* is 2–dimensional matrix where each entry is the load of the between two *CPP*s. In the current implementation we use the link bandwidth computed starting from the Openflow port statistic primitive as the measure of the link load. The code below periodically queries the specified *CPP* for all its neighboring *CPP*s:

```
lqm(addrs='5C:E0:C5:AC:B4:A3',
    every=5000,
    tenant_name='Guests',
    callback=lqm_callback)
```

Listing 8: LQM query creation.

The query is executed periodically with the period set by `every` parameter (in ms). Specifying $every = -1$ will result in a single query being issued. In the above example specifying `ff:ff:ff:ff:ff:ff` will return the link statistics of every *CPP* in the slice. The report includes the packets/bytes transmitted/received on that port. If $every > 0$, the primitive will compute also the average bit–rate and packet–rate in the last observation window.

VI. Evaluation

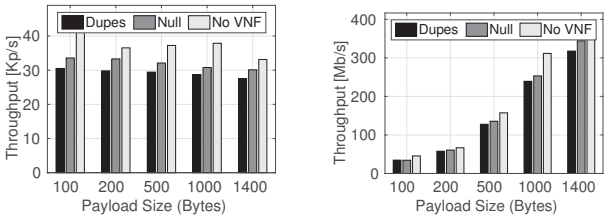
In this section we evaluate the performance of the duplicate filtering *LVNF* described in Sec. V-A. Our evaluation is divided into two parts. In the first part we report on the *LVNF* throughput and latency, while in the second part we quantify the *LVNF* migration overhead.

A. Throughput and Latency

In this scenario we deploy a single duplicate filtering *LVNF*. Two high– end server–class machines connected to two different ports of the *CPP* hosting the *LVNF* are then used as the endpoints of a saturated synthetic UDP stream. UDP packets are encapsulated into WiFi frames and then into the LWAPP frames by the transmitter. Traffic is then dispatched to the *CPP* where it is processed by the duplicates filtering *LVNF*. Finally, the processed stream is delivered to the receiver. Measurements are taken using different payload sizes (100 and 1400 bytes) and with either an empty duplicates

Scenario	*Mb/s*	*C.I. (95%)*	*Kp/s*
100 bytes (empty table)	34.6	0.2	30.5
100 bytes (10k entries)	33.1	1.1	29.1
1400 bytes (empty table)	317.6	7.7	27.5
1400 bytes (10k entries)	318.4	3.9	27.6

TABLE II: Basic packet processing performance.



(a) Throughput (packet/s). (b) Throughput (bit/s).

Fig. 8: Performance of the duplicate filtering *LVNF* as a function of the payload size (bytes).

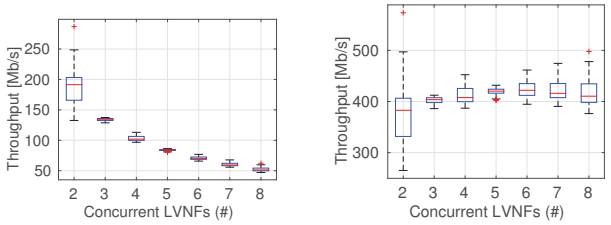
table, i.e. the data structure keeping track of the last sequence numbers seen from a particular client, and a duplicates table with 10000 entries. Each measurement is 60 seconds long and has been repeated 10 times. Results are shown in Table II. As it can be seen the *LVNF* performance essentially does not depend on size of the duplicate table.

We also study the impact of *LVNF* processing on end–to–end performance. Figure 8 shows the system performance for different payload sizes using three different setups, namely: with the duplicate filtering *LVNF*, with an *LVNF* which does not perform any manipulation, and without any *LVNF*. As it can be seen a single low–power Atom CPU running at 1.6 GHz can easily process WiFi frames at 300 Mb/s when the frames are larger than 1400 bytes. Moreover, the performance penalty introduced by the *LVNF* processing is also negligible.

We also evaluate the performance of multiple concurrent duplicates filtering *LVNF*s running on a single *CPP*. The network setup is the same as the one used in the previous experiment. However, in this scenario the server is generating an increasing number of UDP streams (equal to the number of *LVNF*s), each stream is forwarded to a different *LVNF* instance over the same Ethernet port. The payload length is kept constant at 1400 bytes. Each measurement is 60 seconds long and has been repeated 10 times. As it can be seen from Figure 9, the average *LVNF* throughput decrease sub–linearly with the number of concurrent *LVNF*s. However, the aggregated throughput remains essentially constant as the number of concurrent *LVNF*s increases.

B. Migration Overhead

In this section we evaluate the performance of our *LVNF* migration strategy by implementing an application that periodically migrates the duplicate filtering *LVNF* between two *CPP*s. Unlike the previous scenarios where UDP flows were used, in this case we use TCP traffic in that it is in general more sensitive to packet loss and/or out–of–order delivery. We use iperf [48] to generate a saturated TCP stream between the two servers. Traffic is encapsulated at the transmitter into WiFi frames and then into LWAPP packets. The duplicate



(a) *LVNF* throughput. (b) Aggregated Throughput.

Fig. 9: Performance of the duplicate filtering *LVNF* for an increasing number of concurrent instances.

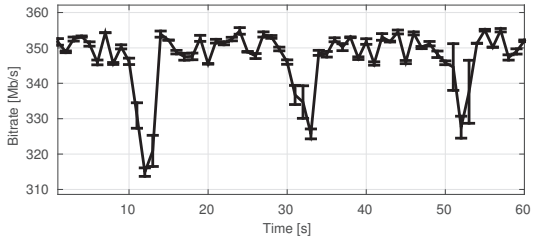


Fig. 10: Impact of *LVNF* migration on TCP throughput.

filtering *LVNF* is migrated every 20s between the two *CPP*s. Figure 10 shows the instantaneous throughput achieved by the TCP connection. As it can be seen from the figure, there is a sensible decrease in performance when the migration is occurring (at seconds 10, 30, and 50), however in each of these case the performance degradation is in the order of 25/30 Mb/s.

VII. Conclusions

In this paper we presented *Scylla*, a high–level declarative language for programming network functions that allows programmers to implement per–flow custom packet processing. We implemented the proposed abstractions on top of the *EmPOWER* SDN/NFV platform consisting of a proof–of–concept controller and a Python–based SDK. We used the SDK to implement a number of applications for MAC offloading in Enterprise WLANs. The entire stack including the datapath implementation, the controller, and the Python–based SDK have been released under a permissive BSD license making it available to the broad research community.

In its current implementation, our framework can accommodate only Click–based *LVNF*s. While this feature allows for high throughput and low–latency data–plane, it also imposes several constraints on the *Network Service* developers, above all, the fact that Click must be used as packet processing engine. As future work, we plan to remove this constraint by extending the *LVNF* concept to third–party network functions. In parallel we also plan to implement a broader set of network functions and to extend and validate the programming abstractions to 4G/5G mobile networks. Finally, we intend to enhance the *Scylla* language with new constructs aimed at allowing *Network Service*s developers to express processing and forwarding constraints (e.g. minimum throughput or maximum latency). We expect such features to play a key role in the design and implementation of service aware state management strategies such as VNF consolidation and scaling.

REFERENCES

[1] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, and I. Tinnirello, "MAClets: active MAC protocols over hard-coded devices," in *Proc. ACM CoNEXT*, Nice, France, 2012.

[2] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "OpenRadio: a programmable wireless dataplane," in *Proc. of ACM HotSDN*, Helsinki, Finland, 2012.

[3] F. Ferreri, M. Bernaschi, and L. Valcamonici, "Access points vulnerabilities to dos attacks in 802.11 networks," in *Proc. of IEEE WCNC*, Atlanta, GA, USA, 2004.

[4] K. Bicakci and B. Tavli, "Denial-of-service attacks and countermeasures in ieee 802.11 wireless networks," *Comput. Stand. Interfaces*, vol. 31, no. 5, pp. 931–941, Sep. 2009.

[5] R. Riggio, M. Marina, J. Schulz-Zander, S. Kuklinski, and T. Rasheed, "Programming abstractions for software-defined wireless networks," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 2, pp. 146–162, June 2015.

[6] European Telecommunications Standards Institute (ETSI), "Etsi gs nfv 002 network functions virtualisation (nfv); architectural framework," December 2014.

[7] "OPNFV." [Online]. Available: https://www.opnfv.org/

[8] "OpenMano." [Online]. Available: https://github.com/nfvlabs/openmano

[9] "OpenBaton." [Online]. Available: http://openbaton.github.io/

[10] "SFC: Service Function Chaining." [Online]. Available: https://datatracker.ietf.org/wg/sfc/charter/

[11] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proc. of ACM SOSP*, Big Sky, Montana, USA, 2009.

[12] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A gpu-accelerated software router," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, Aug. 2010.

[13] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 1, pp. 34–47, March 2015.

[14] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *Proc. of USENIX NSDI 14*, Seattle, WA, USA, 2014.

[15] "OpenStack." [Online]. Available: http://www.openstack.org/

[16] 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," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[17] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proc. of ACM SIGCOMM*, Chicago, Illinois, USA, 2014.

[18] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed vnf state management," in *Proc. of ACM HotMiddlebox*, London, United Kingdom, 2015.

[19] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *Proc. of ACM HotMiddlebox*, London, United Kingdom, 2015.

[20] "The Bro Network Security Monitor." [Online]. Available: https://www.bro.org/

[21] "The Squid Caching Proxy." [Online]. Available: http://www.squid-cache.org/

[22] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. of USENIX NSDI*, Lombard, IL, USA, 2013.

[23] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. of USENIX NSDI*, San Jose, CA, USA, 2012.

[24] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xomb: Extensible open middleboxes with commodity servers," in *Proc. of ACM/IEEE ABCS*, Austin, Texas, USA, 2012.

[25] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proc. of ACM WREN*, Barcelona, Spain, 2009.

[26] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proc. of USENIX NSDI*, Lombard, IL, USA, 2013.

[27] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *Proc. of ACM PADL*, Austin, TX, USA, 2011.

[28] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[29] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proc. of ACM HotSDN*, Helsinki, Finland, 2012.

[30] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proc. of ACM WREN*, Barcelona, Spain, 2009.

[31] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. of ACM POPL*, Philadelphia, PE, USA, 2012.

[32] M. Casado, N. Foster, and A. Guha, "Abstractions for Software-defined Networks," *Commun. ACM*, vol. 57, no. 10, pp. 86–95, Sep. 2014.

[33] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual network embedding algorithms with coordinated node and link mapping," *Networking, IEEE/ACM Transactions on*, vol. 20, no. 1, pp. 206 –219, February 2012.

[34] M. Chowdhury, F. Samuel, and R. Boutaba, "Polyvine: policy-based virtual network embedding across multiple domains," in *Proc. of ACM VISA*, New Delhi, India, 2010.

[35] A. Fischer, J. Botero, M. Till Beck, H. de Meer, and X. Hesselbach, "Virtual network embedding: A survey," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 4, pp. 1888–1906, Fourth 2013.

[36] D. Breitgand, A. Epstein, A. Glikson, A. Israel, and D. Raz, "Network aware virtual machine and image placement in a cloud," in *Proc. of IEEE CNSM*, Zurich, Switzerland, 2013.

[37] M. Barshan, H. Moens, and F. De Turck, "Design and evaluation of a scalable hierarchical application component placement algorithm for cloud resource allocation," in *Proc. of IEEE CNSM*, Rio de Janeiro, Brazil, 2014.

[38] M. Barshan, H. Moens, S. Latre, and F. De Turck, "Algorithms for efficient data management of component-based applications in cloud environments," in *Proc. of IEEE NOMS*, Krakow, Poland, 2014.

[39] R. Guerzoni, R. Trivisonno, I. Vaishnavi, Z. Despotovic, A. Hecker, S. Beker, and D. Soldani, "A novel approach to virtual networks embedding for sdn management and orchestration," in *Proc. of IEEE NOMS*, Krakow, Poland, 2014.

[40] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The dynamic placement of virtual network functions," in *Proc. of IEEE NOMS*, Krakow, Poland, 2014.

[41] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *Proc. of IEEE CNSM*, Rio de Janeiro, Brazil, 2014.

[42] R. Riggio, A. Bradai, T. Rasheed, J. Schulz-Zander, S. Kuklinski, and T. Ahmed, "Virtual network functions orchestration in wireless networks," in *Proc. of IEEE CNSM*, Barcelona, Spain, 2015.

[43] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[44] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," in *Proc. of ACM SIGCOMM*, ser. SIGCOMM '15, London, United Kingdom, 2015.

[45] "Boost C++ Libraries." [Online]. Available: http://boost.org/

[46] "Ryu." [Online]. Available: https://osrg.github.io/ryu/

[47] "ONOS." [Online]. Available: http://onosproject.org/

[48] "Iperf." [Online]. Available: http://iperf.sourceforge.net/