Composing data and control functions to ease virtual networks programmability

Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla

Composing data and control functions to ease virtual networks programmability

Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla
University of Toulouse, IRIT
118 Route de Narbonne, F-31062 Toulouse, France
Email: {FirstName.LastName}@irit.fr

***Abstract*—This paper presents a new domain specific language, called *AirNet*, to design and control virtual networks. The central feature of this language is to rely on network abstractions in order to spare operators the trouble of dealing with the complex and dynamic nature of the physical infrastructure. One novelty of this language is to integrate a network abstraction model that offers a clear separation between simple transport functions and advanced network services. These services are classified into three main categories: *static control* functions, *dynamic control* functions and *data* functions. In addition, we provide an easy and elegant way for programming these functions using the decorator design pattern. The AirNet language is supported by a runtime system handling, in particular, the virtual-to-physical mapping. Despite the fact that it is still in a prototype stage, this runtime has been successfully tested on several use cases.**

I. Introduction

Software Defined Networking (SDN) is one of the latest attempts that aims to make current networks more flexible, so that they can quickly adapt to today's evolving needs. Unfortunately, current SDN controllers are difficult to use in practice, mainly because they provide only low-level programming interfaces that: *i)* prevent network operators from writing separate control modules that can be composed and *ii)* require operators to deal directly with the complex and dynamic nature of the physical infrastructure. By allowing the use of network abstractions, network virtualization has emerged as the most appropriate solution to address these limits.

Currently, there are two main approaches to abstract the physical infrastructure: *i)* the overlay network model which consists in overlaying a virtual network of multiple switches on top of a shared physical infrastructure [1] and *ii)* the single switch abstraction model [2] which abstracts the entire network view into a single logical switch. In previous work [3], we have shown that these two models present some drawbacks, among which: the one big switch model forces network administrators to always use a single router to abstract their physical infrastructure, while the overlay network model does not consider any distinction or logical boundaries between in-network functions and packet transport functions, despite the fact that these two auxiliary policies solve two different problems.

In this paper, we present AirNet, a new high-level language for programming SDN platforms, whose main novelty is to integrate an abstraction model that explicitly identifies two kinds of virtual units: *i) Fabrics* to abstract packet transport functions and *ii) Edges* to support, on top of host-network interfaces, richer in-network functions (firewall, load balancing, caching, etc.). Thus, unlike existing work, AirNet's abstraction model allows to ease the configuration, control and management of the physical infrastructure, while also allowing fine-grained control in order to be able to respond to different types of constraints, whether physical or logical. Additionally, we have designed and implemented a hypervisor that supports this model and achieves its mapping on a physical infrastructure.

The remainder of this paper is organized as follows: in section II, existing works are briefly presented. In section III we start by presenting our network abstraction model, then we give an overview of AirNet's key elements. Its implementation is described in section IV and a program example is exposed through a complete use case in section V. Finally, we conclude and shortly present ongoing work.

II. Related Work

Early works have addressed issues related to the low-level nature of programming interfaces and their inability to build control modules that compose. The *FML* language [4] is one of the very first, it allows to specify policies about flows, where a policy is a set of statements, each representing a simple *if-then* relationship. *Frenetic* [5] is a high-level language that pushes programming abstractions one-step further. *Frenetic* is implemented as a Python library and comprises two integrated sub-languages: *i)* a declarative query language that allows administrators to read the state of the network and *ii)* a general-purpose library for specifying packet forwarding rules.

Additional recent proposals introduced features that allow to build more realistic and sophisticated control programs. Indeed, languages such as *Procera* [6] and *NetCore* [7] offer the possibility to query traffic history, as well as the controller's state, thereby enabling network administrators to construct dynamic policies that can automatically react to conditions like authentication or bandwidth use.

Traffic isolation issues were also addressed in works like *FlowVisor* [8] and *Splendid Isolation* [9]. *FlowVisor* is a software slicing layer placed between the control and data plane. This slicing layer divides the data plane into several slices completely isolated, where each slice has its own and distinct control program. Following the same idea, Gutz *et al.* proposed *splendid isolation* which is a language that allows, on one side, to define network slices, and on the other, to formally verify isolation between these slices.

Recently, Monsanto *et al.* proposed the *Pyretic* language [10], which introduced two main programming abstractions that have greatly simplified the creation of modular control programs. First, they provide, in addition to the existing parallel composition operator, a new sequential one that allows to apply a succession of functions on the same packet flow. Second, they enable network administrators to apply their control policies over abstract topologies, thus constraining what a module can see and do.

Although these related work provide advanced network control languages, none of them make an explicit distinction between transport and in-network functions, property that we think is essential for the program's modularity and reusability.

III. The AirNet language

In this section, we present AirNet's programming pattern and its key instructions, which are summarized in Fig. 1. Each AirNet program contains three major parts. The first part deals with the design of the virtual network. We have chosen a fully declarative approach: one primitive for each virtual unit that has to be added to the network (addHost, addEdge, etc.). The second part contains control policies that will be applied over the defined virtual network (in the following subsections, we describe in more details the primitives used to specify and compose these control policies). Finally, the third part is a separate initialization module through which the administrator defines the mappings existing between virtual units and switches present at the physical level (associating IP addresses to hosts, or mapping an edge to multiple physical switches are examples of such mapping instructions).

```
Virtual Network Design:
  addHost(name)
  addNetwork(name)
  addEdge(name, ports)
  addFabric(name, ports)
  addLink((name, port), (name, port))
Edge Primitives:
  filters: match(h=v) | all_packets
  actions: forward(dst) | modify(h=v) | tag(label) | drop
  network functions: @DynamicControlFct | @DataFct
Fabric Primitives:
  catch(flow) | carry(dst, requirements=None)
Composition Operators:
  parallel composition: "+"
  sequential composition: ">>"
```

Fig. 1. AirNet's key primitives

A. Edge & Fabric network abstraction model

Explicitly distinguishing between edge and core functions was also used by Casado *et al.* in a proposal for extending current SDN infrastructures [11]. We propose to integrate this concept in our network abstraction model (*Fig.* 2), thereby *lifting it up* at the language level. Network operators will thus build their virtual networks using three types of network abstractions:

- *Edges* which are general-purpose processing devices used to support the execution of in-network functions.
- *Fabrics* which are more restricted processing devices used to deal with packet transport issues.

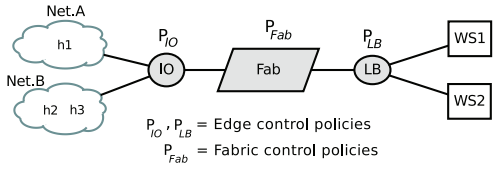

Fig. 2. A simple virtual network using the edge-fabric model

- *Hosts and Networks* which are abstractions used to represent sources and destinations of data flows.

As a consequence, the programming paradigm that we are advocating through this edge-fabric abstraction model is as follows: edges will receive incoming flows, apply appropriate control policies that have been previously defined by network operators (using specific primitives that we will present in the next section), then redirect flows towards a fabric. From this point, it is the fabric's responsibility to carry flows to another border edge in order to be delivered to its final destination (host or network).

B. Edge primitives: filters and actions

Edge primitives are divided into three main groups: *Filters*, *Actions* and *Network Functions*. The language's main filter is the *match(h=v)* primitive that returns a set of packets that have a field *h* in their header matching the value *v*. Actions are applied on sets of packets that are returned by installed filters. *Drop*, *forward* and *modify* are standard actions found in most network control languages. As for the *tag* action, it attaches a label onto incoming packets, label that is used by fabrics to identify and process a packet.

As a simple example, consider the following policy applied to the virtual network of *Fig.* 2, that matches all in web flows (independently of the destination address) on edge *IO* and forwards them to the fabric with a *in_web_flows* tag:

```
match(edge="IO", tp_dst=80) >>
    tag("in_web_flows") >> forward("Fab")
```

This policy uses the match instruction to capture all in web flows (*i.e.*, TCP destination port equals to 80), then it tags these flows as *in_web_flows* by sequentially combining the *match* with a *tag* instruction. Finally, the result is passed to the *forward* action that transfers the packet to the output port leading to the fabric.

C. Edge primitives: network functions

As we have just seen, filters and actions allow network operators to write simple and static control applications. However, we believe that a control language should provide more powerful instructions in order to allow operators to write sophisticated and realistic control applications that can meet a variety of requirements. To fulfill this goal, we have identified two types of advanced network functionalities:

1) Functions that implement complex processing that cannot be performed by the language's basic set of actions (*i.e.*, *forward*, *modify*, *drop*). Encryption, compression or transcoding are examples of such functions.

2) Functions that implement a decision making process capable of generating, at runtime, new policies that change the overall control program behavior. A typical example is a forwarding decision emanating from some deep packet analysis that cannot be specified through the *match* instruction.

We have integrated these two types of functions to AirNet by using the decorator design pattern. Programmers will therefore be able to transform their own Python functions by simply applying these decorators, thereby being able to compose them with other AirNet primitives to build advanced policies.

The first type of function uses the `DataFct` decorator and relies on entire network packets to accomplish its task and return the modified packets. The second type of function uses the `DynamicControlFct` decorator and can rely either on entire network packets or network statistics to generate, at runtime, new policies. Thus, each time a new packet or statistic is available, it is passed to the *decorated* function, which behaves like a callback function that returns, in the end, either a modified packet or a new policy.

```
@DataFct(limit=number, split=[h=v])
@DynamicControlFct(data="packet", limit=number, split=[h=v])
@DynamicControlFct(data="stat", every=seconds, split=[h=v])
```

As shown above, the `DataFct` decorator takes always two parameters: the first one is *limit*, it defines how many packets (from the matched flow) must be redirected to the network function. If *limit* is set to `None`, it means that all packets need to be redirected to the network function. The second parameter is *split*, it allows to discriminate between packets that are sent to the network function. The *split* parameter is a list of headers (*e.g*, `split=["nw_src","tp_dst"]`) that is used by our runtime as a *key* to identify subflows on which the limit parameter applies. If *split* is set to `None`, it means that the limit parameter applies on the entire flow. As for the `DynamicControlFct` decorator, the *data* parameter specifies whether to retrieve entire network packets or statistics related to the number of received bytes and packets. If network packets are used then the *limit* and *split* parameters apply (alike data functions), whereas for statistics, the *limit* parameter is replaced by a polling period specified thanks to the *every* parameter.

In the remainder of this section we will give a few usage examples of data and dynamic control functions.

Let us assume that on edge *IO*, on top of forwarding web flows, a network operator wants to do some advanced processing on the packet's payload before forwarding it to the fabric, in this case compression. This policy can be specified by relying on a specific *compress* function and by sequentially composing it with the other edge primitives:

```
match(edge="IO", tp_dst=80) >>
    compress() >> tag("in_web_flows") >> forward("Fab")
```

The compress function can be simply defined as follows:

```
from tools import cpr
@DataFct(limit=None, split=None)
def compress(packet):
  cpr(packet)
  return packet
```

Here, the parameters *limit* and *split* are set to None, meaning that all packets of the flow must be redirected to the *compress* function and that no discrimination is needed. Also, it is important to stress that the network function must return the modified packet since its result is an input for the next instruction in the composition chain.

Regarding dynamic control functions, we will take a first example of a simplified web application firewall that is implemented within the *waf* function:

```
match(edge="IO", tp_dst=80) >> waf()
```

As shown below, the *waf* function analyses the first packet of a flow and returns a new policy: if it detects malware, a policy dropping all packets from that malicious network address is returned, otherwise, a forwarding policy is returned (note that the *split* parameter is used here to discriminate packets based on the network source address; the *waf* function will thus apply for each first packet coming from a different source address).

```
from tools import DPI
@DynamicControlFct(data="packet", limit=1, split=["nw_src"])
def waf(packet):
  ip = packet.find("ip")
  my_match = match(nw_src=ip.srcip, tp_dst=80)
  if DPI(packet):
    return my_match >> tag("in_web_flows") >> forward("Fab")
  else: return my_match >> drop
```

D. Fabric primitives

Fabrics provide two main primitives that are *catch* and *carry*. The first primitive captures an incoming flow on one of the fabric's ports. Data flows are identified based on a label that has been inserted beforehand by an edge. The second one *carry* transports a flow from an input port to an output port; it also allows to specify some forwarding requirements such as maximum delay to guarantee or minimum bandwidth to offer.

Back to the previous virtual network example (cf. *Fig*.2), we suppose that the administrator wishes to handle both incoming web and ssh flows. The following program represents the transport policy carrying these input flows from edge *IO* to edge *LB*, and vice versa. Since these flows have the same destination, we used the parallel operator to compose the catch primitives and thus have a more concise policy (although we could have also used one policy for each type of flow):

```
(catch(fabric="Fab", src="IO", flow="in_web_flows") +
 catch(fabric="Fab", src="IO", flow="in_ssh_flows")) >>
    carry("LB")
(catch(fabric="Fab", src="LB", flow="out_web_flows") +
 catch(fabric="Fab", src="LB", flow="out_ssh_flows")) >>
    carry("IO")
```

IV. Implementation

In this section we give a brief description of our implementation of the AirNet language which consists of a domain-specific language embedded in Python, as well as a runtime system executed on top of an SDN controller. At the moment, the runtime system, which we name "AirNet Hypervisor", is only a prototype, nevertheless it has been successfully tested on several use cases (using various virtual and physical topologies), including the ones mentioned in this article, using the POX controller and the Mininet network emulator.

A. Filters and actions

The excution of any AirNet program passes through two main phases. The first one is completely independent of the physical infrastructure and deals mainly with composition issues between the different virtual policies. Conversely, the second one is highly correlated to the the physical infrastructure since it generates OpenFlow [12] rules that have to be installed on real network devices.

At the very beginning of the first phase, edge policies are combined, sequentially and in parallel, to form an *edge composition policy*. One difficulty of this process lies in the overlapping rules that can exist. To resolve this problem, rules are placed into containers called *classifiers* that allow us to order the rules by priority and solve match intersection issues. Indeed, when classifiers are generated, rules are combined, and if an intersection exists between two rules, a new one is created, with a match corresponding to the result of the intersection, and a set of actions containing the union of the two original rules actions.

Regarding fabric policies, they are also composed together to form a *fabric composition policy*. In this case, this is a simple process that constructs one forwarding table per fabric, containing incoming flows and their respective edge destination.

The second phase consists in compiling rules stored into classifiers according to the target physical infrastructure and the operator's mapping instructions. For each edge rule that applies on an virtual edge, the runtime fetches the edge's corresponding physical switches, then transforms this virtual rule into one or several physical rules. This transformation includes, for instance, replacing symbolic identifiers present in the program by the corresponding low-level parameters.

As for fabric rules, it is not a trivial algorithm since we cannot just rely on the fabric's forwarding table to install flow rules, mainly because an edge can map to more than one physical switch, and hosts that are connected to this edge can be in reality connected to different physical switches. Thus, we need to split flows that are carried to an egress edge into several flows according to the final destination. In other words, we only deliver (following a shortest path algorithm) to border switches the flows that are intended for a destination directly connected to that switch, and not all the flows arriving to the virtual egress edge.

B. Network functions

In this section we present, through a set of algorithms written in Python, how network functions work and the main ideas behind their implementation. Given that network functions cannot be executed on standard OpenFlow switches, we have implemented them within the controller. Note that this two-tiered architecture is completely transparent to administrators.

We can divide the life cycle of a network function within the controller into three main phases: *i)* initialization at deployment time, *ii)* processing incoming packets at runtime before the limit parameter has been reached, and *iii)* processing incoming packets at runtime once the limit is attained (note that the *every* parameter is handled almost in the same manner).

1) First phase: initialization

This first phase has two main objectives. The first one is to install data path rules sending packets and statistics to the controller, and the second one is to create data structures named *buckets* that will receive and process this data at runtime on the controller. The algorithm in Fig. 3 describes the main steps of this phase.

```
def init_net_functions():
  for net_rule in net_function_rules:
    create_bucket(net_rule)
    actions = set()
    for act in net_rule.actions:
      if (not isinstance(act, NetworkFunction)) and
          (not isinstance(act, forward)):
        actions.add(act)
    actions.add(forward('controller'))
    ctrl_rule = Rule(net_rule.match, net_rule.tag, actions)
    enforce_rule(ctrl_rule)
```

Fig. 3. Initialization phase for network functions

The controller rule is created by keeping the same match filter and flow tag as the network function rule. As for the set of actions, it is copied from that same rule except for the network function action that is replaced by a *forward* action to the controller.

2) Second phase: incoming packets

After the initialization phase, each time a packet arrives at a switch and matches a controller rule, it will be directly forwarded to that controller. The AirNet runtime system will then execute the general algorithm exposed in Fig. 4. It first looks up the appropriate bucket that is going to handle the packet (*i.e.*, the bucket's match covers the packet header) and uses it to call the network function included in the policy. Applying a network function will process the packet and return the results. In case of a data function, the result is a modified packet that will be re-injected into the network and transported to its final destination following the existing policies. In case of a dynamic control function, the result consists in first, a new policy that will be compiled and enforced on the physical infrastructure, and second, re-injecting the incoming packet into the network, thus transporting it according to the newly generated policies as well as the existing ones.

If the *limit* parameter is set on the network function then the bucket counts the number of incoming packets and calls the appropriate function when that limit is reached (cf. next section). Note that if the *split* parameter is also set, then it is necessary to have a packet counter for each micro-flow, which is obtained by applying the split arguments on the packet's header, and appending the result with the rule's match.

3) Third phase: limit reached

When a flow limit is reached, it means that the network function must not be applied anymore. In order to do so, we have implemented the `remove_net_function` that deletes the network function from the rule's action set, while leaving the other basic actions (*i.e.*, modify, tag and forward). The runtime then regenerates edge classifiers and compares them with the old classifiers using the `get_diff_lists` function. This step aims to identify the differences that exist between what is already installed onto the physical infrastructure and

```
def process_packet_in(switch_id, packet_match, packet):
  bucket = get_bucket_covering(packet_match)
  bucket.apply_network_function(switch_id, packet)
  if bucket.split is None:
    bucket.nb_packets += 1
    if bucket.nb_packets == bucket.limit:
      flow_limit_reached(bucket.match)
  else:
    micro_flow = get_micro_flow(packet)
    try:
      bucket.nb_packets[micro_flow] += 1
    except KeyError:
      bucket.nb_packets[micro_flow] = 1
    if bucket.nb_packets[micro_flow] == limit:
      micro_flow_limit_reached(micro_flow)
```

Fig. 4. Processing incoming packets for network functions

the new rules that were generated after removing the deprecated network function. Finally, these lists are enforced onto the physical infrastructure via adequate OpenFlow messages. These steps are exposed in Fig. 5.

```
def flow_limit_reached(flow):
  for rule in edge_policies.rules:
    if rule.match == flow:
      remove_net_function(rule)
  new_classifiers = to_physical_rules(edge_policies)
  updates = get_diff_lists(old_classifiers, new_classifiers)
  install_diff_lists(updates)
```

Fig. 5. Limit is reached for all packets of a network function

```
def micro_flow_limit_reached(micro_flow):
  for rule in edge_policies.rules:
    if rule.match.covers(micro_flow):
      new_rule = copy.deepcopy(rule)
      new_rule.match = micro_flow
      remove_net_function(new_rule)
      enforce_rule(new_rule)
```

Fig. 6. Limit is reached for a subset of packets of a network function

For micro-flows (cf. Fig. 6), the process is a bit different since the network function has reached its limit only for a micro-flow (depending on the *split* parameter) and not for all flows. Thus, we only have to install one new rule that handles this micro-flow, while maintaining the general controller rule. To this end, the runtime starts by fetching the network function rule that covers the deprecated micro-flow, it creates a copy of it, changes its match with the micro-flow match and removes the network function action (keeping the other actions). Finally, it enforces this new rule onto the physical infrastructure with a higher priority than the controller rule (thus guaranteeing that packets matching the micro-flow rule will be handled using this new rule while the other packets will still remain redirected to the controller).

V. USE CASE

In this section, we present a complete use case which consists in programming a dynamic load balancer. The goal is to show how a realistic AirNet program is built and executed from start to end. As a proof of concept, the use case has been implemented and tested on the Mininet network emulator.

Virtual network design: As with any AirNet program, the first step is to define a virtual topology that meets our high-level goals. Here, we reuse the virtual topology presented in *Fig.* 2.

Policy functions: The second part of the program deals with control policies definition. In this use case, we have grouped these policies into three functions, one per virtual device. The first function encapsulates policies that need to be installed on the *IO* edge. It configures the behavior of the edge as a simple input/output device, meaning that it will only match flows and redirect them either inside or outside the network.

```
def in_out_policy(self):
  i1 = match(edge="IO", src="Net.A", nw_dst=pub_WS,
        tp_dst=80) >> tag("in_web_flows") >> forward("Fab")
  i2 = match(edge="IO", src="Net.B", nw_dst=pub_WS,
        tp_dst=80) >> tag("in_web_flows") >> forward("Fab")
  i3 = match(edge="IO", dst="Net.A",
        nw_src=pub_WS, tp_src=80) >> forward("Net.A")
  i4 = match(edge="IO", dst="Net.B",
        nw_src=pub_WS, tp_src=80) >> forward("Net.B")
  return i1 + i2 + i3 + i4
```

Concerning the load balancing policy, it will be implemented on the *LB* edge. Its goal is to intercept web flows (*i.e.*, HTTP flows sent to the web server's public address), and pass them to a dynamic control function. This dynamic function will install at runtime rules that change the destination addresses (*i.e.*, IP and MAC) of these flows to one of the backend severs, while ensuring a workload distribution over the two servers using a simple Round-Robin algorithm (cf. network function *rrlb* in the `i1` policy below). The load balancer needs also to modify the responses in order to restore the public address instead of the private ones (policies `i2` and `i3`).

```
def load_balancing_policy(self):
  i1 = match(edge="LB", dst=pub_WS, tp_dst=80) >>
        self.rrlb()
  i2 = match(edge="LB", src="WS1") >>
        modify(src=pub_WS) >> tag("out_web_flow") >>
            forward("Fab")
  i3 = match(edge="LB", src= "WS2") >>
        modify(src=pub_WS) >> tag("out_web_flow") >>
            forward("Fab")
  return i1 + i2 + i3
```

The *rrlb* function is triggered for each first packet coming from a different IP source address, since the parameter *limit* is set to "1", and the parameter *split* is set to "nw_src". Below, we can see that the *rrlb* function extracts the match filter from the received packet, then uses it to install a rule for the other packets that belong to the same flow as the retrieved packet. The forwarding decision is based on the actual value of a token: if it is one, then the flow is sent to the first backend server, else it is sent to the second one.

```
@DynamicControlFct(data="packet", limit=1, split=["nw_src"])
def rrlb(self, packet):
  my_match = match.from_packet(packet)
  if self.rrlb_token == 1:
    self.rrlb_token = 2
    return my_match >> modify(dst="WS1") >> forward("WS1")
  else:
    self.rrlb_token = 1
    return my_match >> modify(dst="WS2") >> forward("WS2")
```

The last policy concerns the fabric. Here, we need a simple transport policy that carries in web flows to the egress edge, and out web flows to the ingress edge. The following extract shows the definition of this transport policy:

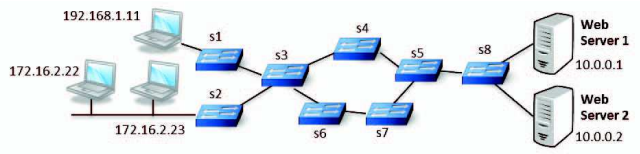

Fig. 7. Target physical infrastructure

```
def fabric_policy(self):
  t1 = catch(fabric="Fab", src="IO",
         flow="in_web_flows") >>  carry("LB")
  t2 = catch(fabric="Fab", src="LB",
         flow="out_web_flows") >> carry("IO")
  return t1 + t2
```

Experiments: The use case is tested by first launching Mininet, the POX controller and the AirNet hypervisor, and then by generating web requests from our web clients (three clients: one in `Net.A` and two in `Net.B`). Regarding the physical infrastructure, we relied on the topology depicted in Fig. 7 and we have chosen a virtual-to-physical mapping showed within Table I (lines 1 and 3).

In the first phase, our hypervisor compiles the high-level policies and generates OpenFlow rules that are pushed onto the switches through the controller (all this process is part of the proactive phase). The OpenFlow entries that are generated are *match*, *output* and *drop* actions. In addition, due to the *rrlb* dynamic control function, OpenFlow entries that send packets to the controller via *packet-in* messages are also generated.

Next, we executed several *wget* requests on the web server's public address from the hosts connected to `s1` and `s2`. All the requests and their responses were correctly routed through the network, allowing the web clients to retrieve the requested HTML files on the back-end web servers. The requests were balanced on the two web servers according to the round robin algorithm implemented in the *rrlb* dynamic control function (these packets arriving at runtime at the controller are handled in the reactive phase).

Discussion: Table I shows the number of rules that have been installed on each physical switch, during both the proactive and reactive phases.

The switches `s1` and `s2` have three OpenFlow rules each: two forwarding rules (web flows to and from networks 192.168.1.0/24 on `s1` and 172.16.0.0/16 on `s2`) and one drop rule for all other traffic. As for `s8`, it contains four rules: one that redirects the first packet of each web flow towards the controller, two to handle traffic coming from `WS1` and `WS2` and one drop-all rule. Regarding the fabric's switches, `s3`, `s4` and `s5` have each five rules installed: two to handle the incoming web flows towards the server's public IP address, two rules to handle the responses from the private web servers, and one drop-all rule for any other unidentified flow. Since `s6` and `s7` are not located on the shortest path, only one drop-all rule has been installed on these switches.

Overall, 27 physical rules have been automatically installed by the AirNet hypervisor (during the proactive phase), while only 9 policies have been specified on the virtual network. Note that this ratio would have been even greater if we were dealing with a more complex physical infrastructure including a greater number of hosts and switches.

TABLE I. NUMBER OF POLICIES AND RULES FOR THE USE CASE

Virtual device	**IO**		**Fabric**					**LB**	Total
Policies	4		2					3	9
Physical switch	s1	s2	s3	s4	s5	s6	s7	s8	Total
Rules (init. phase)	3	3	5	5	5	1	1	4	27
Rules (runtime, 3 clients)	3	3	5	5	5	1	1	7	30

Additionally, during the reactive phase, the number of rules changes dynamically on `s8` since the `LB` edge containing the *rrlb* function is mapped on that switch. Each time a host sends a request to the web server's public address, the first packet of the flow is redirected from `s8` to the controller that executes the *rrlb* function. As a result, a new rule is installed on `s8` to forward the flow to one of the two back-end web servers. Since we have used three clients in our tests, in the end, seven rules are installed on `s8`.

Although some issues are still under development, these first results are encouraging and demonstrate the feasibility of the approach. Notably, this use case shows that AirNet allows to significantly simplify network programmability by relying on high-level policies and network functions that can be easily composed and reused, without considering the large amount of underlying physical rules.

Another benefit is that the administrator no longer needs to deal manually with intersection issues between policies, and possible side-affects resulting from the application of a policy. For instance, in this use case, the fact of having applied a *modify* action on the servers responses to change their IP source address from `WS1` and `WS2` to `pub_WS` implies that rules installed on path `[s5, s4, s3]` must use the public IP address while rules installed on `s8` must use the private ones. This kind of issue is automatically solved by the AirNet hypervisor, avoiding a complicated and error-prone process.

VI. CONCLUSION

This paper described the design and the implementation of AirNet, a new high-level domain specific language for programming virtual networks. We used network virtualization as a main feature in order to both build portable control programs independent of the physical infrastructure and to spare administrators the trouble of dealing with low-level parameters, thus complying with the SDN promise to make network programming easier. The main novelty in AirNet is to integrate an abstraction model that offers a clear separation between functions that represent network basic packet transport capacities and functions that represent richer network services. In addition, AirNet provides, through the decorator design pattern, an easy and intuitive way for programming advanced network functions, allowing operators to extend the language according to their own requirements. Finally, we described our prototype of AirNet hypervisor, which has been successfully tested over several use cases, some of them presented in this article.

Currently, we are still testing the AirNet hypervisor and finishing some implementation issues such as handling dynamic events emanating from the physical topology. Also, we are working on the language's formal model in order to be able to guarantee the policies correctness.

REFERENCES

[1] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the Network Forwarding Plane," in *PRESTO*. ACM, 2010.

[2] E. Keller and J. Rexford, "The "Platform As a Service" Model for Networking," in *Proc. of the 2010 Internet Network Management Workshop (INM/WREN'10)*. USENIX Association, 2010.

[3] M. Aouadj, E. Lavinal, T. Desprats, and M. Sibilla, "Towards a virtualization-based control language for SDN platforms," in *Proc. of the 10th Int. Conf. on Network and Service Management (CNSM)*, 2014.

[4] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proc. of the 1st ACM Workshop on Research on Enterprise Networking*. ACM, 2009.

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Notices*, vol. 46, no. 9, 2011.

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

[7] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *SIGPLAN Notices*, vol. 47, no. 1, 2012.

[8] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proc. OSDI'10*. USENIX Association, 2010.

[9] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proc. HotSDN Workshop*. ACM, 2012.

[10] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software Defined Networks," in *USENIX Symposium, NSDI*, 2013.

[11] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A Retrospective on Evolving SDN," in *Proc. of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. ACM, 2012.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM*, 2008.