AirNet: the Edge-Fabric model as a virtual control plane

Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla

AirNet: the Edge-Fabric model as a virtual control plane

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*—While there are many motivations to virtualize networks, easing their management is probably one of the most important reasons. However, the choice of the network abstraction model that will be used to abstract the physical infrastructure represents a major challenge, given that this choice will have a significant impact on what an administrator can see and do. In this article, we present a new domain specific language, called AirNet, to design and control virtual networks. The central feature of this language is to rely on a new 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 developed a hypervisor that supports the AirNet language and handles, in particular, the virtual-to-physical mapping.**

I. Introduction

The NFV/SDN shift allowed the emergence of new opportunities for network virtualization solutions which, until recently, were very difficult to achieve. To be used to its best advantages, network virtualization requires, first and foremost, identifying practical abstractions that allow operators 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 [1]. Today, there are two main approaches that are used for abstracting the physical infrastructure: *i)* the overlay network model [1] which consists in overlaying a virtual network of multiple switches on top of a shared physical infrastructure and *ii)* the one big switch abstraction model [2] which consists in abstracting the whole network view in a single logical switch.

In continuation of our previous work on network control languages [3], we present in this article AirNet, a new high-level language for programming SDN platforms. In order to ensure better modularity and flexibility, we put network virtualization at the very heart of our language, and, unlike existing works, we rely on a new abstraction model that explicitly identifies three kinds of virtual units: *i) Fabrics* to abstract packet transport functions, *ii) Edges* to support, on top of host-network interfaces, control-plane functions and *iii) Data Machines* to provide complex data processing. This abstraction model allows AirNet to offer different types of network services (*i.e.*, transport, data, static and dynamic control services) that can be both composed and chained together to define the overall control policy, as well as reused over different physical infrastructures. 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, we present the abstraction model we rely on, and the main motivations behind its proposition. In section III, an overview of the language is presented through four use cases, while their experimental results are exposed in section IV. Finally, related work are presented in section V, followed by a conclusion and a brief description of our future work.

II. Choosing the right abstraction model

From our point of view, using the *one big switch* model presents mainly two drawbacks: the major one is that it forces network administrators to always use a single router to abstract their physical infrastructure, which can be a very restrictive approach especially when there are underlaying physical constraints that can not or should not be hidden from the control program operating on top of the virtual network (*e.g.*, specifying administrative boundaries, network function distribution according to physical topology characteristics). The second drawback is more a software engineering issue. Indeed, using the one big switch abstraction involves putting all in-network functions within the same router, thereby resulting in a monolithic application in which the logic of different in-network functions are inexorably intertwined, making them difficult to test and debug.

Regarding the *overlay network* model, it also presents a shortcoming, that is, unlike the *one big switch* model, there is no distinction or logical boundaries between in-network functions and packet transport functions, despite the fact that these two auxiliary policies solve two different problems. The result of this is that network operators will be forced to also consider packet transport issues when specifying their in-network functions, which will naturally lead to control program and network functions that are less modular and reusable.

Edge and Fabric: lifting up the modularity at the network control language level

To overcome the limitations of both models, we relied on a well-known idea within the network designer community, which is making an explicit distinction between edge and core network devices, as it is the case with MPLS networks.

Explicitly distinguishing between edge and core functions was also used by Casado *et al.* in a proposal for extending current SDN infrastructures [4]. We propose to integrate this concept in our network abstraction model, thereby *lifting it up* at the language level. Network operators will thus build their virtual networks using four types of abstractions:

- *Edges* which are general-purpose processing devices used to support the execution of network control functions.
- *Data machines* which are processing elements present at the data plane to support the execution of complex data functions.
- *Fabrics* which are more restricted processing devices used to deal with packet transport issues.
- *Hosts* and *Networks* which are abstractions that are 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 either directly to another border edge in order to be delivered to its final destination (host or network) or indirectly by first passing through one or more data machines in order to apply some complex data processing on the flow.

Considering the above discussion, we believe that decomposing network policies into transport, control and data functions will enable network operators to write control programs which are much easier to understand, reason about and maintain. More importantly, the possibility to interconnect multiple edges, data machines and fabrics provides a good level of abstraction, while also ensuring a sufficient flexibility in order to be able to consider both various contexts of use (*e.g.*, campus networks, data centers, operator networks) and physical constraints if necessary.

III. AirNet overview

In this section, we start by briefly presenting AirNet's programming pattern and its key instructions, then we illustrate the language's usage through four use cases (all of which have been successfully tested in Section IV).

A. AirNet's key instructions

Every AirNet program contains three main phases: the first phase deals with the design of the virtual network, the second one specifies the control policies that will be applied over this virtual network, and finally the third one defines the mappings existing between virtual units and switches present at the physical level. This last phase is separated from the first two allowing network operators to reuse their control program over different physical infrastructures without requiring any changes apart from the mapping instructions. Due to space constraints, we will not describe the mapping phase but we will present execution results of the use cases on different physical topologies in section IV.

The virtual network and control policies are specified thanks to several AirNet's instructions summarized in Fig. 1.

Designing the *virtual network* relies on a straightforward declarative approach: one primitive for each virtual unit that has to de added to the network (`addHost`, `addEdge`, etc.).

```
Virtual Network Design:
  addHost(name)
  addNetwork(name)
  addDataMachine(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
Fabric Primitives:
  catch(flow) | carry(dst, requirements=None) |
  via(dataMachine, dataFct)
Composition Operators:
  parallel composition: "+"
  sequential composition: ">>"
```

Fig. 1. AirNet's key primitives

Edge primitives are divided into three main groups: *Filters*, *Actions* and *Dynamic Control 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.

Static edge control policies are specified by composing a filter with actions (*e.g.*, `match()>>tag()>>forward()`). These policies can be enforced on the physical infrastructure at design time. In order to take into account more complex control policies, AirNet introduces *dynamic control functions* that are evaluated at runtime directly by the controller. Use cases III-C and III-D will show examples of such functions.

Fabrics provide three primitives: *catch*, *carry* and *via*. The *catch* primitive captures an incoming flow on one of the fabric's ports (based on a label inserted beforehand by an edge). The *carry* primitive transports a flow from one edge to the other (both connected to the fabric). It is also possible to specify forwarding requirements such as maximum delay to guarantee or minimum bandwidth to offer. Finally, a *via* primitive composed with a *carry* redirects a flow through a data function before reaching the final edge. Use case III-E will present such an example.

B. Static control function: simple forwarding example

The aim of this first use case is to illustrate AirNet's basic primitives. The high-level goal here is to enable *Net.A* hosts to communicate only with server *WS1*, and similarly *Net.B* hosts only with server *WS2*. Figure 2 depicts the virtual topology that we have chosen. We used two fabrics for demonstration purposes only. This choice remains at the discretion of the administrator. Although all edges in the virtual network can be connected to a unique fabric, in some cases, it can be useful to rely on multiple fabrics according to the network operator's high level goals or to the physical constraints in place. Note however that using multiple fabrics does not imply necessarily the usage of different physical resources: two fabrics can map to the same set of physical switches or not.

The second step is to define control policies that will allow the two networks to communicate with their respective servers.

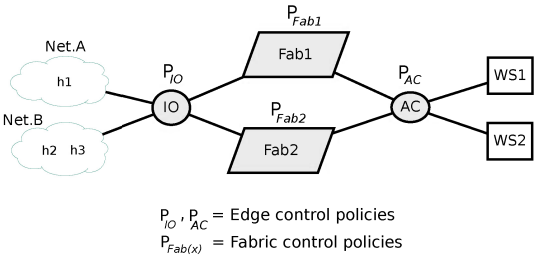

Fig. 2. A simple virtual network with two fabrics

The following extract shows policies that allow all network flows between *Net.A* and *Server1* (similar policies are used between *Net.B* and *Server2*):

```
def edges():
  e1 = match(edge="IO", src="Net.A", dst="WS1") >>
       tag("in_ws1") >> forward("Fab1")
  e2 = match(edge="IO", dst="Net.A") >> forward("Net.A")
  e3 = match(edge="AC", src="WS1", dst="Net.A") >>
       tag("out_ws1") >> forward("Fab1")
  e4 = match(edge="AC", dst="WS1") >> forward("WS1")
  return e1 + e2 + e3 + e4

def transport():
  t1 = catch(fabric="Fab1", src="IO", flow="in_ws1") >>
       carry(dst="AC")
  t2 = catch(fabric="Fab1", src="AC", flow="out_ws1") >>
       carry(dst="IO")
  return t1 + t2
```

The first function configures edges `IO` and `AC` as simple input/output devices, meaning that they will only match flows and redirect them either inside or outside the network. For instance, the policy `e1` uses the match instruction to capture all flows coming from `Net.A` and having `WS1` as destination, then it tags these flows as *in_ws1* by sequentially combining the *match* with a *tag* instruction. Finally, the result is passed to the *forward* action that transfers packets to the output port leading to the first fabric.

The transport function deals with fabric policies. In this simple example, labeled flows are carried from edge `IO` to edge `AC`, and vice versa.

Note that AirNet does not impose any constraint on the control program's structure (*i.e.*, how policies are grouped into more general functions). For example, here we have defined *edge()* and *transport()* functions but one could use a different decomposition such as one function for all policies allowing *Net.A-WS1* flows and another function for *Net.B-WS2*.

C. Dynamic control function: load balancer example

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 integrated the concept of dynamic control function that implements a decision making process capable of generating, at runtime, new policies that change the control program's behavior.

Dynamic control functions are defined in AirNet by using the decorator design pattern. Programmers will therefore be able to transform their own Python functions by simply applying this decorator, thereby being able to compose them with other AirNet primitives to build advanced policies.

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

As shown in the above syntax, the `DynamicControlFct` decorator has a *data* parameter that 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. The *limit* 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. If statistics are used instead of entire network packets, *limit* is replaced by a polling period specified thanks to the *every* parameter.

As a concrete example, we will consider a use case that implements a dynamic load balancer on the same previous virtual topology (Fig. 2). The load balancing function is installed on edge `AC`. It intercepts web flows (*i.e.*, HTTP flows sent to the web server's public address), and passes them to a dynamic control function that generates a new policy changing the destination addresses (*i.e.*, IP and MAC) of these flows to one of the backend severs (*i.e.*, `WS1` and `WS2`), while ensuring a workload distribution over the two servers. Moreover, the `AC` edge needs to modify the servers responses in order to restore the public address instead of the private ones (cf. `e2` and `e3`).

```
def AC_policy():
  e1 = match(edge="AC", nw_dst=pub_WS, tp_dst=80)
       >> Dynamic_LB()
  e2 = match(edge="AC", src="WS1")
       >> modify(src=pub_WS)
       >> tag("out_web_flows") >> forward("Fab1")
  e3 = match(edge="AC", src="WS2")
       >> modify(src=pub_WS)
       >> tag("out_web_flows") >> forward("Fab2")
  return e1 + e2 + e3
```

As shown below, the `Dynamic_LB` 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". The function extracts the match filter from the received packet, then uses it to generate a new policy for the other packets that belong to the same flow as the retrieved packet. Hence, one new forwarding rule will be installed at runtime on the physical infrastructure for each flow with a different IP source address. Regarding the forwarding decision, it is based on a simple Round-Robin algorithm: if the value of a token 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 Dynamic_LB(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")
```

D. Dynamic control function: data cap example

As explained in the previous section, dynamic control functions can also rely on network statistics. Here we present a data cap use case which monitors and possibly suspends traffic coming from a particular host if it exceeds a data threshold (again, we use the virtual topology shown in Fig. 2):

```
match(edge ="IO", src ="Net.A") >> tag("in_flows")
    >> (forward("Fab1") + check_data_cap())
```

The *check_data_cap* function is executed in parallel of the *forward* action (the + operator is used). Based on the decorator's arguments, the function will be called by the network hypervisor every hour, providing statistics sorted by network source address:

```
@DynamicControlFct(data="stat", every=3600, split=["nw_src"])
def check_data_cap(stat):
  if stat.byte_count > threshold:
    return (stat.match >> drop)
```

Note that policies returned by dynamic control functions are always installed with a higher priority than existing ones. In this example, it guarantees that the *drop* action on the subflow is installed with a higher priority than the global *forward* action executed in parallel, thus only blocking the flows that exceed the threshold.

E. Data function: codec conversion example

Until now, we have seen that AirNet allows to easily configure paths and other control functions between hosts within a virtual network, configuration which may be static or dynamic. However, it is quite common, especially today with the rise of Network Functions Virtualization (NFV), to have networks that include several network appliances or middleboxes that implement complex *data processing* on the packet's payload that cannot be performed by the switches' basic set of actions (*i.e.*, forward, modify, drop). Encryption, compression or transcoding are examples of such functions. For this purpose, AirNet provides a mechanism to easily redirect a flow through one or several data plane middleboxes, that we define as *DataMachines*. As for edges, a *DataMachine* must be connected to a fabric. However, they are not associated to any AirNet primitive since their processing depends only on the data function they host.

For this use case, we consider several clients streaming media from a VoD service. Fig. 3 shows the virtual topology we use. We assume that flows from the VoD service consumed by *Net.A* end-users must pass through the codec conversion function deployed on the CC data machine. This policy can be completely specified within the fabric that implements the service chaining logic between the virtual units:

```
def transport():
  t1 = catch(fabric="Fab", src="IO", flow="in_vod_flows")
       >> carry(dst=AC)
  t2 = catch(fabric="Fab", src="AC", flow="out_vod_flows")
       >> via(dataMachine="CC", dataFct="CodecConversion")
       >> carry(dst=IO)
  return t1 + t2
```

IV. Implementation and Evaluation

In this section, we briefly outline AirNet's implementation, then we present the experiments we have conducted on different use cases and physical topologies executed with the Mininet network emulator.

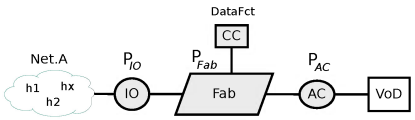

Fig. 3. Virtual network with one data machine

A. AirNet's general architecture

The AirNet language has been implemented as a domain-specific language embedded in Python, as well as a runtime system that we name "AirNet hypervisor". Fig. 4 gives an overview of our prototype's architecture. Our current implementation relies on the POX controller. The runtime core module is composed of two main parts: *i*) the proactive core that relies on other modules to resolve, in particular, policy and topology mapping issues and to generate the initial network's configuration, and *ii*) the reactive core that handles dynamic control functions and changes that may occur in the physical topology (*e.g.* link down).

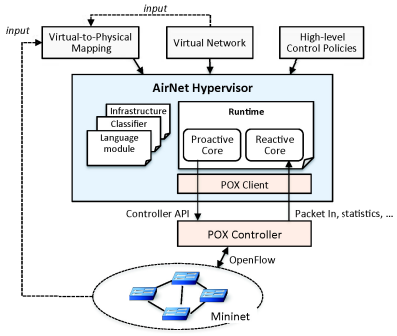

Fig. 4. AirNet's architecture overview

B. Experiments

In this section we present some experimental results we have obtained by conducting functional, performance and scalability tests. In the first set of tests (Table I), we executed four use cases addressing different scenarios with different virtual topologies (detailed in section III). These tests were performed on the same physical topology composed of 11 physical switches. In the second set of tests (Fig. 5), we did some measures with greater number of virtual policies as well as larger physical topologies. Finally, the last set of tests (Fig. 6) concentrated on end-to-end delay with respect to network functions utilization. All these tests were performed with Mininet in a virtual machine with 2 GB of RAM and a 2,7 GHz Intel Core i5 CPU.

Table I shows the execution results in terms of total physical rules generated and compilation time divided into two main steps: *i*) the virtual composition step which mainly includes retrieving infrastructure information to build a corresponding graph, and composing virtual policies to resolve all intersection issues, *ii*) the physical mapping step which includes transforming virtual policies into physical rules, finding appropriate

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

Use case	*Virtual policies*	*Physical rules generated*	*Composition time*	*Physical mapping time*
twoFab	12	42	101	20
dynLB	9	31	94	16
bwCap	6	27	89	11
dataFct	10	44	100	29

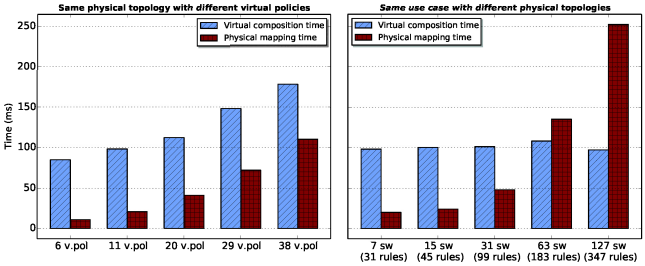

Fig. 5. AirNet's proactive phase: compilation time according to the number of virtual policies (left) and to the number of physical switches/rules (right)

paths for each flow, and finally distributing physical rules on switches according to the specified mapping module.

Number of generated physical rules. Table I underlines the difference between the number of specified virtual policies and the number of rules actually installed by our hypervisor on the physical switches, which is greater by a factor two or three. Thus, a first conclusion that we can draw is that the usage of AirNet effectively simplifies network programmability, and this not only by providing modern control instructions, but also by sparing administrators from the tedious task of writing a large number of low-level rules and handling at the same time their intersection and priority issues. Moreover, the cost of this simplification is highly acceptable since, for all the use cases, it remains in the order of a hundred milliseconds.

Virtual composition time. As illustrated in Fig. 5, the virtual composition time is highly correlated to the number and the complexity of the specified virtual policies, but totally independent from the size of the physical infrastructure. Indeed, we can see on the left graph that the more virtual policies, the greater the virtual composition time is. However, on the right graph, the same control program executed over different physical topologies (varying from 7 to 127 switches) shows the same virtual composition time.

Physical mapping time. Still looking at Fig. 5, we can see that the physical mapping time is correlated to both virtual policies and physical infrastructure. On the left graph, we have the same physical topology, but the more virtual policies are added, the more the hypervisor needs to solve policy composition issues, which ultimately gives a greater mapping time. The same goes for the right graph, the number of policies does not change but the more the topology is large and complex, the more the hypervisor needs to perform calculation to find paths and install a large number of physical rules, thereby resulting in a greater physical mapping time.

Dynamic and data functions delay. We finally made tests to measure the impact of dynamic control and data functions on the end-to-end delay. To conduct this experiment, we installed a path between two hosts within the network (composed of

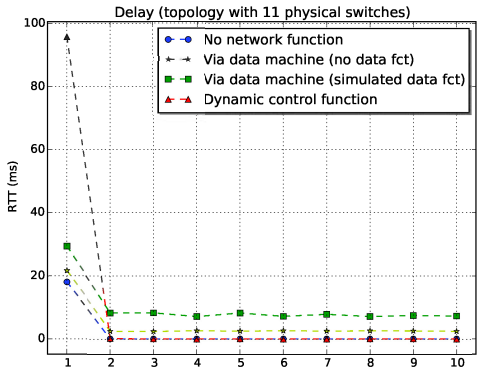

Fig. 6. Impact of dynamic control and data functions on the RTT

11 physical switches) and measured the round-trip time (RTT) between them using four different approaches: *i*) the path is installed proactively (*i.e.* at deployment time), *ii*) the path is installed by a dynamic control function at runtime, *iii*) the path is installed proactively but goes through a *DataMachine*, which does not perform any function (other than forwarding the flow) and *iv*) the path goes through a *DataMachine* in which we have simulated a data function. Fig. 6 shows the results.

With no network function, the first measure is higher than the other nine: this is due to the initial ARP query that is handled by a proxy ARP we have implemented within the controller. This also explains why the first RTT is higher in the data functions tests. In our experiments, passing through an "empty" *DataMachine* (implemented by a Mininet host with two interfaces) adds only a few milliseconds to the RTT. When we simulated a simple function (printing the payload's content), the RTT increased a little ($< 10ms$) but this is not relevant since the actual delay will depend entirely on the data function's complexity. Finally, regarding the dynamic control function, the high first delay ($\simeq 95ms$) is mainly due to the time required to redirect the first packet of the flow to AirNet's hypervisor, to evaluate the function and install the new path according to the returned policy. Once the new path is installed, the RTT is equivalent to the one of proactive static paths.

To conclude, these charts demonstrate the feasibility of our edge-fabric approach and, to a lesser extent, its scalability (we emulated tree topologies up to 127 physical switches). Moreover, the different use cases we have tested confirm that AirNet allows programmers to focus on the specification of their high-level policies and delegate the low-level complexity to the hypervisor (high number and intersection of physical rules). Also, we have seen that using these network abstractions leads to an extra time cost, nevertheless we think that this small cost is perfectly bearable considering the benefits in terms of modularity and flexibility brought by the use of network virtualization.

V. RELATED WORK

Proposing modern programming language for SDN platforms has been the subject of numerous research projects. In this section we will present those that we think are the most important and closest to our research problem.

The focus of early related works was mainly on the low-level nature of SDN programming interfaces and their inability to build control modules that compose. The *FML* language [5] 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* [6] 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 proposals introduced features enabling the construction of more realistic and sophisticated control programs. Indeed, languages such as *Procera* [7] and *NetCore* [8] offer the possibility to query traffic history, as well as the controller's state, thereby allowing network administrators to construct dynamic policies that can automatically react to conditions like authentication or bandwidth use.

AirNet's contribution differs from these previous work by the logical abstractions on which network operators specify their control programs. These logical abstractions mainly allow to: *i*) ease network programmability by providing only relevant information to administrators in order to specify their high-level goals and *ii*) reduce the correlation between the control programs and the physical infrastructure, thereby enabling a better reusability and greater flexibility.

Recently, Monsanto *et al.* proposed the *Pyretic* language [9], 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 applies a succession of functions on the same packet flow. Second, they allow network administrators to use their control policies over abstract topologies, thus constraining what a control module can see and do.

Merlin [10] is a recent programming language designed to address essential aspects of the one big switch abstraction. Its main contribution is to allow programmers to specify a set of authorized forwarding paths through the network using regular expressions that match sequences of network locations, including names of packet-processing functions. Merlin also allows to specify bandwidth constraints on network flows. However, we think that it presents two shortcomings: *i*) even if it is possible to process the entire network as a single big switch, there is no clear separation between the control program and the physical infrastructure (*i.e.*, physical constraints are tied to the control program) and *ii*) Merlin offers limited dynamic adaptation, meaning that Merlin allows programmers to only add new constraints on existing flows, but they can not change the overall control program behavior.

The novelty of the AirNet control language is to rely on an edge/fabric abstraction model, compared to existing work that make use either of the one big switch (*e.g.*, Merlin) or the overlay model (*e.g.*, Pyretic). We claim that using this model at the virtual network level offers greater modularity by enforcing logical boundaries between edge and fabric policies and thus easing their evolution, maintenance and reuse. Another difference with related work is that, thanks to a separate mapping module, we make a clear distinction between the virtual network and the physical infrastructure, while also allowing administrators to consider the possible underlying physical constraints (such as in-network functions placement, processing capabilities or administrative boundaries) that can not or should not be hidden from the control program.

VI. Conclusion

This paper described the design and the implementation of AirNet, a new high-level domain specific language for programming virtual networks. AirNet's programming pattern involves, first, defining a virtual network that copes with the operator's high-level goals and possible physical constraints, second, specifying the control policies that will apply to it. 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. 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 (*i.e.*, data and dynamic control functions). In addition, AirNet provides an easy and intuitive way for programming and composing these advanced network services, allowing operators to extend the language according to their own requirements. Finally, we described our AirNet hypervisor prototype, 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. Also, we are working on the possibility to send code (*i.e.*, data functions) to be executed on data machines, and on the proposal of a development environment for AirNet that would include a specific policy editor and debugger.

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] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A Language for Provisioning Network Resources," in *CoNEXT 14*, 2014.