Demo: AirNet in action

Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla

Demo: AirNet in action

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 demo paper presents the implementation and the use of AirNet, a new domain specific language 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. AirNet is supported by a runtime system handling, in particular, the virtual-to-physical mapping.**

I. INTRODUCTION

Over time, network management has become a tedious, complex and error-prone task for a human administrator. Network virtualization has therefore emerged as the most appropriate solution to address these limitations. However, to be used to its best advantages, network virtualization requires identifying practical abstractions that allow operators to ease the configuration and the control of the physical infrastructure. 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.

AirNet is a new high-level control language for SDN platforms, which main novelty is to integrate a new 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, complex network functions. Additionally, we have designed and implemented a hypervisor that supports this model and achieves its mapping on a physical infrastructure.

The demo presented here exemplifies the programming paradigm that we are advocating through AirNet and its Edge-Fabric abstraction model, knowing that a fuller presentation of AirNet and its features is available in [3]

II. AIRNET LANGUAGE

Every AirNet program contains three main phases: the first phase deals with the design of the virtual network, the second one specifies virtual control policies, and finally the third one defines the mappings existing between virtual units and switches present at the physical level. AirNet's primitives are divided into two main groups: *i) Edge primitives* which includes *Filters*, *Actions* and *Network Functions*. 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 packet flows. *Network functions* include *Dynamic control functions* which implement a decision making process capable of generating, at runtime, new policies that change the control program's behaviour, and *data function* which implement complex data processing on the packet's payload that cannot be performed by the switches basic set of actions. *ii) Fabrics primitives* which mainly include the *catch* and the *carry* instruction. The *catch* primitive captures an incoming flow 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). Ultimately, AirNet control programs are obtained by composing, sequentially and in parallel, edge and fabric policies, policies that are commonly obtained by sequentially composing a filter with actions (*e.g.*, `match()>>tag()>>forward()`, `catch()>>carry()`).

III. AIRNET 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. 1 gives an overview of our prototype's architecture. The runtime core module is composed of two main parts: the proactive and the reactive core.

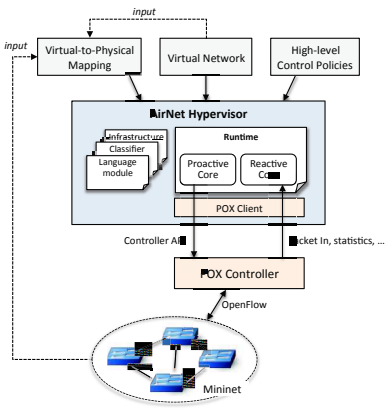

Fig. 1. AirNet's architecture overview

1) The proactive core: is a module that includes the implementation of two main processes: *i)* virtual composition process, the function of which is basicly retrieving infrastructure information to build a corresponding graph, and composing virtual policies to resolve all intersection issues, *ii)* physical mapping process which includes transforming virtual policies into physical rules, finding appropriate paths for each flow, and finally distributing physical rules on switches.

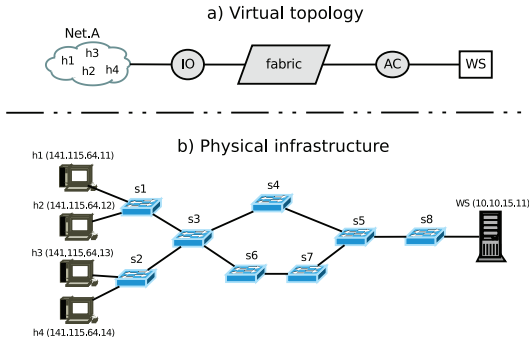

Fig. 2. a) Specified virtual topology. b) Targeted physical infrastructure

2) The reactive core: which handles network functions execution and changes that may occur in the physical topology (*e.g.* link down). Network functions life cycle is divided within the reactive core 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.

IV. Demonstration Scenario

In this section, we present a complete use case, which has been implemented and tested on the Mininet network emulator, using the physical topology depicted in figure 2. The use case consists in programming a dynamic access control list in order to enable communication between allowed users (*i.e.*, not blacklisted) and server `WS`.

As with any AirNet program, the first step is to define a virtual topology that meets our high-level goals. Here, we need one host to represent our web server, one network to encompass the users, two edges (`IO` and `AC`) that are connected to the users' network and the web server, respectively. Finally, we use a fabric to connect our edges. The overall virtual topology is depicted in figure 2. Moreover, 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.).

The second part of the program deals with control policies definition. AirNet does not impose any constraint on the program's structure. For example, here we have defined `access` and `transport` functions but one could use a different decomposition such as one function per virtual devices.

```
def access()
  e1 = match(edge="IO", src="Net.A", dst="WS") >> filter()
  e2 = match(edge="IO", dst="Net.A") >> forward("Net.A")
  e3 = match(edge="AC", src="WS") >>
       tag("out_flows") >> forward("fabric")
  e4 = match(edge="AC", dst="WS") >> forward("WS")
  return e1 + e2 + e3 + e4
def transport()
  t1 = catch(src="IO", flow="in_flows") >> cary("AC")
  t2 = catch(src="AC", flow="out_flows") >> cary("IO")
  return t1 + t2
```

The first function configures edges as simple input/output devices. For instance, the policy `e1` uses the match instruction to capture all flows coming from `Net.A` and having host `WS` as destination, then the result is sequentially passed to the *forward* action that transfers packets to the dynamic control function `filter`. On the other hand, the `transport` function deals with fabric policies. In this simple example, labeled flows are carried from edge `IO` to edge `AC`, and vice versa.

```
@DynamicControlFct(data="packet", limit=1, split=["nw_src"])
def filter(packet):
    if packet.srcip in blacklist:
      return (match(edge="IO", nw_src=ip.srcip) >> drop())
    else: return (match(edge="IO", nw_src=ip.srcip) >>
        tag("in_flows") >> forward("fabric"))
```

As shown above, the `filter` 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 *IP* address from the received packet, then it checks if the address is blacklisted. If so, a new dropping policy is generated for the other packets that belong to the same flow as the recieved packet, if not, a forwarding policy is generated.

Mapping rules: Before executing our program, the mapping rules have to be defined according to the underlying physical infrastructure. The following mappings have been defined: **`IO`** maps to `s1` and `s2`, **`AC`** maps to `s8`, **`fabric`** maps to [`s3-s7`], **`Net.A`** network map to `141.115.64.0/24` IP addresses, and finally, **`WS`** maps to `10.10.15.11` host.

These mappings are specified through a separate module thanks to the following instructions:

```
class MyMapping(Mapping):
    def __init__(self):
        Mapping.__init__(self)
        self.addEdgeMap("IO", ["s1", "s2"])
        self.addEdgeMap("AC", ["s8"])
        self.addFabricMap("fabric",
              ["s3", "s4", "s5", "s6", "s7"])
        self.addNetworkMap("141.115.64.*", "Net.A")
        self.addHostMap("10.10.15.11", "WS")
```

Experiments: In the first phase, the AirNet hypervisor compiles the high-level policies and generates OpenFlow rules that are pushed onto the switches through the POX controller (all this process is done by the hypervisor's proactive core). The OpenFlow entries that are generated are *match* and *output* actions to physical ports on `s1, s2` and on `s3, s4, s5, s8` which are located on the shortest path in the physical topology. In addition, due to the `filter` dynamic control function, OpenFlow entries that send packets to the controller are installed on `s1` and `s2` (these packets arriving at the controller are handled by the hypervisor's reactive core).

Next, at runtime, we executed several *wget* requests on the web server from hosts connected to `s1` and `s2`. For allowed hosts (*i.e.*, `h1` and `h3`), all the requests and their responses were correctly routed through the network, allowing the web clients to retrieve the requested HTML files. By contrast, for hosts which are blacklisted (*i.e.*, `h2` and `h4`), all their requests were dropped at the border of the network.

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, "Composing data and control functions to ease virtual networks programmability," in *Proc. of NOMS 2016*, 2016.