Towards a Virtualization-based Control Language for SDN Platforms

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

***Abstract*—Software defined networking (SDN) approaches rely on control languages to programmatically express the desired network behavior. Several SDN control languages use network virtualization to abstract the complex and dynamic nature of the physical infrastructure. However, almost all these languages use the same network abstraction model, which we believe is not the most appropriate one for expressing flexible and reusable network control policies. This paper presents work in progress towards a new high-level virtualization-based control language for SDN platforms. The main novelty of this language is to integrate a network abstraction model that explicitly identifies two kinds of virtual units: *i) Fabrics* to abstract packet transport functions and *ii) Edges* to abstract richer networking functions. We believe that this approach will allow network administrators to easily express modular and reusable network control policies independently of the underlying infrastructure.**

I. Introduction

Networks have become increasingly complex and hard to control due to major evolutions in computing environments, such as desktop and server virtualization, the wide adoption of cloud computing or the advent of "Big Data". Administrators are therefore looking for more flexible networks that can quickly adapt to the evolving needs of today's enterprises, carriers, and end users. Software Defined Networking (SDN) is the latest attempt in order to respond to this lack of flexibility of current network architectures. In order to do so, SDN decouples the control plane from the data plane, and centralizes it in a logical and programmable entity called the *controller* [1]. Thus, network administrators can quickly define and change control policies by simply (re)programming the controller using standard programming interfaces.

Unfortunately, current SDN controllers provide low-level programming interfaces that have several limits (*e.g.,* to compose existing control modules, it is necessary to *manually* combine their logic in a new program instead of just reusing them, as is the case in most high-level languages), thus making network control programs complicated, error-prone and difficult to maintain and reuse. Early works have addressed these deficiencies by proposing higher-level languages that include modern features, such as a declarative design [2], composition operators [3], embedded verification tools [4] and, more especially, the possibility to perform network virtualization [4, 5, 6], which is the main issue of this article.

There are many reasons why we need to virtualize networks (*e.g.*, isolation, customized network services), but probably the most important one is to ease their management [7]. Indeed, virtualization exposes logical abstractions (*i.e*, virtual networks) that are decoupled from the physical infrastructure. These abstractions provide just enough information to specify high-level goals, thus making control policies both easier to write, since only the desired behavior is expressed, and modular (subsequently reusable), since they are no more attached to a particular infrastructure. However, virtualization presents two major design challenges: the choice of the network abstraction model that will be used to abstract physical infrastructures (*i.e.*, the forwarding plane), and the technology needed to map the logical state onto the underlying physical infrastructure [8].

In this paper, we mainly address issues related to the first challenge by presenting work in progress towards the definition of a new high-level control language for SDN platforms. We put network virtualization at the very heart of our language, and, unlike existing works, we rely on a new abstraction model that we think is more appropriate for our language design requirements, which are: *i) expressiveness*: the language must be human-friendly and not attached to a specific platform *ii) modularity*: administrators must be able to write and compose independent control modules *iii) flexibility*: the language must cover most of current network functionalities (e.g., access control, routing, load balancing), and should not impose too strong restriction in order to be able to meet, as far as possible, future requirements as they raise.

The remainder of this paper is organized as follows: in section II, we discuss network abstraction models that are currently used by existing control languages, then we present our new approach. Section III gives an overview of our language's key elements. An illustration program is exposed through a toy example in section IV. Finally, we conclude and briefly present ongoing work.

II. Choosing the right abstraction model

A. Background

One of the major challenges of virtualizing software-defined networks is the choice of the network abstraction model that will be used to abstract the physical infrastructure. There are currently two main approaches : *i)* the overlay network model and *ii)* the single router abstraction model.

To the best of our knowledge, most network control languages use the *overlay network* model which consists in overlaying a virtual network of multiple switches on top of a shared physical infrastructure [8]. Virtual switches are very similar to standard switches in the physical infrastructure: they include lookup tables, ports and expose a set of basic forwarding actions. Virtual switches can also map to one or more physical switches, and are connected to each other within a logical topology via virtual links.

As an alternative to the *overlay network* model, Keller and Rexford proposed the *Platform as a Service* model [9]. This model abstracts the network view in a single logical router in order to enable network administrators to focus solely on their in-network functions (*i.e.*, any functionality that benefits from being inside the network) rather then worrying about managing the virtual network. The single router includes three main processing components : 1) a routing component that provides the ability to customize path selection 2) a data plane component that exposes some basic functionalities like forwarding and 3) a general-purpose processing component that exposes in-network functions like firewall, load balancing or access control.

The question, then, is which model to choose for our network control language, taking into account that the abstract model must ensure the expressiveness, the modularity and the evolutivity of the language.

B. Models discussion

As mentioned earlier, the biggest advantage of using the *Platform as a Service* model is that it allows network administrators to focus only on the expression of in-network functions that they plan to install on their network. However, we think that, from a network programming language point of view, this model suffers from a big disadvantage: it forces network administrators to put different in-network functions within the same router, thereby reducing significantly the reuse of this component. Moreover, the resulting application will be a monolithic program in which the logic of different in-network functions are inexorably intertwined, making it difficult to test, debug, maintain and reuse.

On the other hand, the *overlay network* is a more modular approach, since the model allows network administrators to define multiple logical switches, on which they can install in-network functions. These switches can be afterwards reused to, easily and quickly, construct sophisticated network control applications. However, we think that this model suffers from one major shortcoming, that is, unlike the *platform as a service* model, there is no distinction between in-network functions and packet transport functions, despite the fact that these two auxiliary policies solve two different problems. Indeed, this shortcoming makes the definition of in-network functions more difficult, since their specification must also consider issues related to packet transport across the virtual network (*e.g.*, selecting the appropriate virtual path among several available).

C. Edge and Fabric: lifting up the modularity at the language level

In order 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 the network edge and network core 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 [10]. We propose to integrate this concept in our network abstraction model, thereby *lifting it up* at the language level. Network administrators will thus build their virtual networks using two types of virtual devices:

- *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.

Considering the above discussion, using edges and fabrics will allow us to take advantage of both previous models. Indeed, using fabrics enables network administrators to abstract packet transport issues, thereby allowing them to focus solely on the definition of complex in-network functions. On the other hand, the possibility to use multiple edges allows to decouple and distinguish in-network functions, thus facilitating their test, debug and, more especially, their reuse. Moreover, we believe that our approach will enable network administrators to write control programs which are much easier to understand, reason about and maintain.

III. Language overview

Virtualization plays a central role in our language, and hence every control program will be composed of two main parts (and some initialization routines): the first part deals with the design of the virtual network, and the second part contains control policies that will be applied over the virtual network. In the following, we describe in more detail each of these parts. Figure 1 summarizes the key elements of the language.

A. Virtual network design

In order to allow network administrators to easily and clearly design their virtual networks, we chose a fully declarative approach. Thus, building a virtual network would only imply describing virtual devices and the connections (*i.e.*, virtual links) that exist between them.

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 | no_packets
Actions : forward(destination) | modify(h=v) | tag(label) | drop
Queries : packet(limit) | byte_count(every) | packet_count(every)
Fabric Primitives:
catch(flow)
carry(destination, requirements=None)

Fig. 1. Summary of the language's key elements

We distinguish three types of components, depending on their role in the virtual network.

The first type of components are *hosts* and *networks* that are used to represent sources and destinations of data flows. A *host* can represent a single end system (*e.g.*, end host, application-level gateway or proprietary hardware appliance), while a *network* can represent a range of end systems. The use of these two components both discharges administrators from relying on specific addresses and makes control policies easier to read and write.

The second type of components are *edges* which are general processing devices placed at the border of the virtual network in order to support in-network functions installation. Thus, edges play the role of host-network interfaces by the fact that ingress edges will receive incoming data flows, inspect packet's headers to identify which in-network function is to be considered, and redirect flows to an egress edge for delivery to the destination, or to an intermediate edge for potential further treatment. In addition, it is important to stress that edges are purely logical entities that can map to one or more switches in the physical infrastructure.

The third and last type of components are *fabrics* which represent the network's raw forwarding capacities. The fabric's primary purpose is packet transport. It exposes only a minimal set of forwarding primitives and uses a specific addressing mechanism that is much simpler than the one used by edges (*i.e.*, using a unique label instead of several header fields). In normal cases, all edges in the virtual network will be connected to a unique fabric. However, in some specific cases, virtual networks can include more than one fabric according to the network administrator's high level goals. Indeed, it is important to note that two fabrics within the same model will map to two *separate* collections of physical switches. This design choice allows us to capture specific network policies such as expressing an explicit physical backup path for critical data flows.

Once network administrators have finished with the description of virtual devices, they will then just need to set-up the different virtual links in order to connect hosts or networks to edges, and edges to fabrics.

B. High-level policy functions

Using two types of virtual devices, namely edge and fabric, implies having two distinct instruction sets. Indeed, this will allow the two components to *evolve separately*, focusing on their specific problems.

Fabrics expose two main primitives that are *catch* and *carry*. The first primitive captures a specific flow, identified by a label, entering any port of the fabric. The label identifying the data flow has been inserted beforehand by an edge. The second instruction *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.

Edges are more complex devices than fabrics, and hence expose a richer set of instructions. Edge primitives are divided into three main groups : *Filters*, *Actions* and *Queries*.

Filters are primitives that do not change the packet's contents. The language's main filter is the *match(h=v)* primitive, which, when installed on an edge, returns a set of packets that have a field h in their header matching the value v.

Contrary to filters, *actions* are primitives that can change packets value or location. They are applied on sets of packets that are returned by installed filters. The simplest action is *drop* which discards a packet received on one of the edge's input port. The *forward* action allows to move, within the same edge, a packet from an input port to an output port. The *modify* action is used to update one or more of the packet's header fields. Lastly, the *tag* action allows to attach a label onto incoming packets, considering that labels are the unique information that a fabric will use to identify a packet.

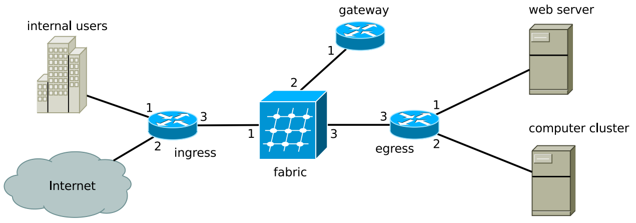

Fig. 2. Virtual network topology use case

The third and last group of edge primitives are *queries*. Like actions, queries are applied on filters. We distinguish two main kinds of queries depending on the type of information they return. The first kind is composed of *packet_count* and *byte_count* which, as their name suggests, allow to periodically poll packet and byte counters that are associated to filters. The second kind of query is *packet* which allows to poll entire raw packets. In addition to providing the ability to conduct network monitoring, queries enable network administrators to construct dynamic policies by allowing them to associate queries to callback functions that are executed each time a raw data is collected or a timer has elapsed.

IV. Toy Example

This section presents a simple use case in which we illustrate a preliminary version of our high-level network control language. The overall management goal of this use case is to configure an enterprise network in order to prevent external access to sensitive resources. The policy is that any user who is part of the enterprise's internal network can have access to all available resources (*i.e.*, web server and computer cluster). On the contrary, external users can only have access to web resources and are not allowed to access the enterprise's cluster.

As described previously, the first step consists in describing a virtual network that matches our high-level goals, thus abstracting all irrelevant information that are related to the physical infrastructure. The following program is used to describe the virtual network showed in figure 2 (notice that not all links are represented in this extract):

```
# Virtual network topology
topo.addEdge(name="ingress", ports=(1,2,3))
topo.addEdge(name="egress", ports=(1,2,3))
topo.addEdge(name="gateway", ports=(1))
topo.addFabric(name="fabric", ports(1,2,3))
topo.addNetwork(name="internal_users")
topo.addNetwork(name="Internet")
topo.addHost(name="web_server")
topo.addHost(name="computer_cluster")
topo.addLink(("ingress",3), ("fabric",1))
topo.addLink(("gateway",1), ("fabric",2))
topo.addLink(("egress",3),  ("fabric",3))
# ...
```

Having described the virtual network, the next step is the specification of the control policy. The following piece of code represents the in-network function that will be installed on the ingress edge. This function configures the edge so that it classifies incoming internal flows as "trusted" and the external ones as "unreliable". Once the classification has been done, the ingress edge will simply forward flows to the fabric in order to be transported to their right destination.

```
# ingress function
  match(edge="ingress",source="internal_users") >>
    tag("trusted_flow") >>
      forward("fabric")
  match(edge="ingress",source="Internet") >>
    tag("unreliable_flow") >>
      forward("fabric")
```

In the context of this example, the gateway is only designed to analyze unreliable flows. We therefore use the following transport function to configure the fabric so that unreliable flows are transported to the gateway, while trusted ones are directly transported to the egress edge.

```
 # fabric function
  catch(fabric="fabric",flow="trusted_flow") >>
    carry("egress")
  catch(fabric="fabric",flow="unreliable_flow") >>
    carry("gateway")
```

For the gateway's configuration, we define the below in-network function that performs two actions. The first one is to discard all flows that want to reach the enterprise's cluster, since only unreliable flows are redirected to the gateway. The second one is to reclassify all web flows as "trusted" flows, since they are allowed to access the enterprise's web server.

```
# gateway function (for unreliable flows)
  match(edge="gateway",destination="web_server") >>
    tag("trusted_flow") >>
      forward("fabric")
  match(edge="gateway",destination="computer_cluster") >>
    drop
```

Finally, the last in-network function simply configures the egress edge in a manner that it forwards web requests to the web server and forwards computation requests to the computer cluster.

```
# egress function
  match(edge="egress",destination="web_server") >>
    forward("web_server")
  match(edge="egress",destination="computer_cluster") >>
    forward("computer_cluster")
```

Due to space constraints, we did not detail the control policy responsible for handling server and cluster responses.

It is important to stress that none of the previous in-network functions consider packet transport issues. Indeed, all focus only on their high-level goal (*i.e.*, classifying in ingress, analyzing in gateway and delivering in egress), and at the end, functions just send data flows to the fabric which ensures the transportation to the right destination.

V. Conclusion and Current Work

This paper described the design of a new high-level language for "programming" software-defined networks. We used network virtualization as a main feature in order to spare administrators the trouble of dealing with the myriad of irrelevant information that are related to the physical infrastructure, thus complying with the SDN promise to make network programming easier. The novelty of this language lies in the use of a new virtual model that we think is more appropriate for both language design requirements (*i.e.*, expressiveness, modularity and flexibility) and network abstraction requirements (*i.e.*, providing just enough information in order to express the desired behavior).

Currently, we are working on the design and the technical development of a network hypervisor that will support the control language we presented. In addition to the main control module, which contains the virtual network and control policies, the hypervisor will rely on a mapping module, which mainly consists in associative arrays binding each virtual unit to its respective physical counterpart of the underlying infrastructure. This mapping information will be reused afterwards by the network hypervisor's runtime to generate a policy for the physical infrastructure that is semantically equivalent to the one applied over the virtual network.

We have implemented our network control language as a domain-specific language embedded in Python. To map the logical state of the virtual network onto the physical infrastructure, the prototype relies on the POX controller, an open source development platform for Python-based SDN control applications. At present, we are working on the hypervisor's proactive part that compiles the language's policies into OpenFlow instructions. The immediate next steps include, first, the development of the reactive part that allows to handle network events coming from the infrastructure (*e.g.*, link failures); second, testing the prototype on more complex use cases.

References

[1] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 3, 2008.

[2] 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.

[3] 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.

[4] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proc. of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. ACM, 2012.

[5] 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. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[6] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software Defined Networks," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, 2013.

[7] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, 2013.

[8] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the Network Forwarding Plane," in *Proc. of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO'10)*. ACM, 2010.

[9] 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.

[10] 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.