A Datapath-centric Virtualization Mechanism for OpenFlow Networks

R. Doriguzzi-Corin, E. Salvadori, M. Gerola
CREATE-NET, Trento, Italy
Email: {rdoriguzzi, esalvadori, mgerola}@create-net.org

M. Suñé, H. Woesner
BISDN GmbH, Berlin, Germany
Email: {marc.sune, hagen.woesner}@bisdn.de

***Abstract*—The adoption of a robust and scalable network virtualization framework is a key requirement in order to make the vision of a shareable network infrastructure a reality. To this aim, one of the most suitable approaches is the one which takes advantage of the emerging paradigm of Software-Defined Networking (SDN) and OpenFlow, its de-facto standard. Several virtualization frameworks have been proposed in the last few years; however, they are either based on proxy-based solutions that raises scalability and robustness issues (FlowVisor), or they rely on a simplified view of the datapath (generally based on Open vSwitch instances) that have little chances to be adopted in production network settings. This paper presents a novel OpenFlow-based network virtualization mechanism exploiting a recent open-source datapath project named eXtensible Datapath Daemon (xDPd); the proposed multi-platform datapath is based on a robust distributed virtualization architecture that is able to run on multi-version OpenFlow switch network scenarios, has a minimal overhead from a performance point of view and can be easily ported on several hardware platforms via xDPd libraries.**

Keywords*—*Network Virtualization, OpenFlow, FlowVisor, Software Defined Network

I. Introduction

One of the most relevant use case scenario for Software Defined Networks (SDN) is network virtualization [1]. In fact, the abstraction of node switches in an SDN architecture facilitates exposing different views of a physical network to different controllers. Compared to more traditional virtualization architectures based on existing techniques such as Virtual Local Area Network (VLAN), Virtual Routing Forwarding (VRF), Virtual Private Network (VPN) and Virtual eXtensible Local Area Network (VXLAN) techniques which provide only partial network virtualization features, SDN may in fact provide stronger mechanisms to improve the control of virtual network instances running on top of a physical topology. Among them, key features are: strict isolation between virtual networks in terms of performance and traffic leakage, the possibility to enhance existing network devices via a software upgrade that enable virtualization capabilities and lack of dedicated middleboxes that may introduce administrative overhead and act as potential single points of failure.

By considering the emerging adoption of an SDN paradigm at all network segment levels, the vision of a network infrastructure that can be safely shared among several administrators is finally becoming a reality (Infrastructure as a Service, IaaS). However, there is no common view on how an SDN network should be virtualized and, in fact, many virtualization frameworks based on SDN have been proposed recently, each one of them with their own advantages and disadvantages.

Thanks to its wide adoption and success, OpenFlow [2] is the protocol used in most of the SDN deployments and in which SDN virtualization techniques has been focused on. Leveraging on OpenFlow protocol, we may envision two major approaches to introduce network virtualization in an SDN network: (i) frameworks that leverage on an external proxy to intercept OpenFlow control messages and assign them to different controllers according to a specified "flowspace slicing" (e.g. FlowVisor [3] and VeRTIGO [4]); (ii) frameworks that assume the capability at switch level to instantiate several instances of OpenFlow virtual switches and then assign them to different controllers like [5], [6].

The former have been widely adopted in several Future Internet testbeds such as GENI [7] and OFELIA [8] thanks to their simplicity and ease of use. However they have several limitations: (i) the proxy controller constitutes a single point of failure for the control plane, effectively making it impossible in case of failure for the north-bound (slice) controllers to interact with the datapaths; (ii) there is an inherent overhead due to the fact that *PacketIn* events, as well as other control messages, have to be encapsulated/decapsulated twice and transmitted/received via a socket; (iii) their implementation is very much OpenFlow 1.0 centric, partially due to their internal architecture but also due to the known fact that slicing in multi-table datapaths supporting apply-actions is a complex problem to solve.

The latter have been recently proposed to overcome these limitations and to define a network virtualization mechanism that can effectively provide the key features identified before. However, almost all these alternative architectures have been proposing solutions based on Open vSwitch (OvS) [9], a virtual software switch that is being heavily used in data-center "server-centric" scenarios but has little applicability to carrier-grade switches.

In this paper a novel OpenFlow-based network virtualization framework has been proposed that overthrows FlowVisor limitations and leverages on a recent open-source datapath project named eXtensible Datapath Daemon (xDPd) [10] available for several hardware platforms and targeting carrier-grade SDN applications. The proposed framework is based on a robust distributed virtualization architecture that is able to run on a multi-version OpenFlow switch network scenarios via a minimal overhead, both from a performance and an operational point of view.

The rest of the paper is organized as follows. Section II describes the motivations behind this work and reviews the

related literature and existing tools. Section III discusses the architectural details of the proposed approach. In Section IV the results of some experimental sessions are given. Directions for future work and conclusions are drawn in Sections V and VI respectively.

II. Motivations and related work

Among the network virtualization frameworks alternative to FlowVisor, the following two are probably the most relevant for our work. In [6] Sonkoly et al. have proposed a framework that is capable of managing multiple instances of OpenFlow switches with different forwarding capabilities and OpenFlow versions, as well as to run and configure controllers designed for controlling a virtual network. The whole architecture assumes Open vSwitch (OvS) [9] as the basic pillar, an assumption that may be quite limiting in hardware switches scenarios. In the proposed architecture each virtual switch instance is associated to a single virtual network and to a single controller somehow limiting the flexibility of the network instantiation. Furthermore no performance measures are provided in terms of additional overhead introduced over the operation of the physical and virtual networks. In their proposal in [11], Skoldstrom et al. have focused their attention on the encapsulation techniques that may be adopted in a distributed virtualization framework to enforce isolation of virtual networks, by including a thoroughly analysis of overhead introduced by VLAN versus PWE (pseudo-wire emulation) mechanisms. However, the architecture is not OpenFlow version agnostic and, moreover, no performance measures are provided to measure the overall additional overhead of the proposed mechanism.

In order to tackle a robust and scalable SDN-based network virtualization mechanism, we believe there is still much room for improvements. Our decision was to investigate in more detail the potential improvements that may be introduced at the datapath level. As part of the activities performed within the ALIEN FP7 project [12], an evaluation of the different existing open-source OpenFlow datapaths to be extended was done. The main candidates evaluated were Open vSwitch (OvS) and the eXtensible Datapath daemon (xDPd) [10].

OvS is a production-quality virtual software switch that was originally built to replace the Linux kernel bridge with a more flexible software switch. OvS has support for OpenFlow 1.0, but the support for 1.1, 1.2 and 1.3.2 versions is still under development [13][1]. Although it was originally conceived to replace the Linux kernel bridge, it has also been used to port some hardware switches (ASICs), despite its internal architecture and code complexity have proven to make it a considerably difficult task.

The eXtensible Datapath Daemon, in its turn, is a relatively young open-source datapath project that targets the easy adoption of SDN/OpenFlow to a wide variety of different software[2] and hardware platforms, as well as an extensible design to support new OpenFlow versions and network protocols. It has to be said though, that the code-base is not yet as mature as the OvS's one. xDPd can be also considered a framework for building datapath elements, rather than a single datapath. Its architecture proposes a unified control (e.g. OpenFlow) and management, confining the platform specific code in the platform driver. The currently available open-source platform support includes a user-space software GNU/Linux implementation and support for NetFPGA 10G cards. There is also closed-source platform support for some Broadcom chipsets, OCTEON Network Processors or a DPDK accelerated software datapath among others.

In our network virtualization architecture the xDPd datapath platform was selected due to its architecture specially tailored to support multiple HW/SW platforms, but also due to the current stable support of OF1.0 and 1.2[3] as well as its more comprehensible and easy to extend software code-base.

As it will be described in more detail in Section III, when compared to [11] and [6], the proposed architecture shares the advantage of being distributed and thus inherently more robust to potential failures. It is also based on a software solution that minimizes the additional OPEX/CAPEX needed in order to be adopted in the network. However compared to those works, since it leverages on xDPd libraries, it has much better chances to be applied on several hardware switching platforms.

III. Datapath virtualization architecture

The distributed virtualization architecture presented in this paper is composed of two macro-blocks (see Fig. 1): (i) the Virtualization Agent (VA) which resides on OpenFlow-enabled switches and (ii) the so-called Virtualization Agent Orchestrator (VAO) which is an entity (stand-alone process or plugin for a Network Management System) in charge of configuring and monitoring the VA instances running on the network devices.

The VAO is a JSON-RPC client which is used to send configuration commands (*createSlice*, *deleteSlice*, *addFlowSpace* etc.) to the VA instances. On the northbound, the VAO exposes a JSON-RPC interface that allows slice configurations to be applied/updated either from a Network Management System (see Fig. 1) or via command-line. In addition, this interface is compatible with existing control software (in particular with FOAM [14]).

The VA has been designed with the following goals: (i) avoid Single Point of Failures (SPoF) through a distributed slicing architecture, (ii) provide an OpenFlow version agnostic slicing mechanism and (iii) minimize the latency overhead caused by the slicing operations.

Distributed slicing. As shown in Fig. 1, the proposed architecture is designed to avoid SPoFs. A failure of the VAO, the only centralized element in the architecture, can only prevent the instantiation of new slices without affecting the ones already in operation. Other approaches like [3] or [4] introduce an additional layer on the control channel to obtain the virtualization of the network resources. A failure of that layer would bring down all the running slices.

Protocol agnostic. The VA operates between the control communication module and the forwarding plane of the device.

[1] At the time of writing, the most recent release of OvS is the 2.1.2, while full support for OpenFlow 1.1, 1.2 and 1.3 is planned with the release of OvS version 2.3.0.

[2] In theory xDPd could control the Open vSwitch kernel module, however there is no known ongoing project targeting this support.

[3] Support of OF1.3.2 is under development

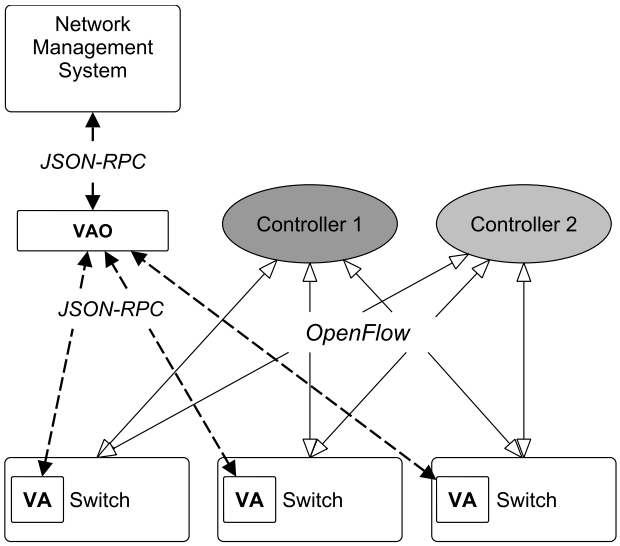

Fig. 1. The network architecture. The Virtualization Agent (VA) runs on OpenFlow-enabled nodes while the Virtualization Agent Orchestrator (VAO) is the process in charge of configuring/monitoring the agents.

For this reason, it does not need to inspect the control protocol to perform the slicing process therefore it can, in principle, support any control protocol (even different from OpenFlow).

Latency overhead. The operations needed to obtain the virtualization of the resources have a cost in terms of additional latency on actions that cross between the control and the forwarding planes. The overhead depends on how the virtualization mechanism is implemented but, as shown by the evaluation results reported in Section IV, other elements can contribute to the total latency. In particular, differently from FlowVisor, the Virtualization Agent neither inspects the OpenFlow protocol nor needs to establish additional TLS connections.

A. The slicing mechanism

In OpenFlow, for each incoming flow which does not have an entry in the switch flow table, a *PacketIn* event is generated and sent to the controller through the control channel. The controller, in turn, can answer with a *FlowMod* message to modify the flow tables of the switch. The workflow in Fig. 2 shows how the VA performs the slicing process (dashed box) for these messages. *PacketOut* and other messages are handled in a similar way. For each new flow, the fields of its header are matched against the flowspaces assigned to the configured slices. If the VA finds a match, the header is sent to the related OpenFlow endpoint which builds the *PacketIn* message by using the protocol version used for the communication with the controller. Vice-versa, if no correspondences are found, the VA tells the lower layers to drop the flow.
On the other side, the VA applies the slicing policies to the OpenFlow messages sent by the controller to the switch. In order to keep the VA internal processes protocol version agnostic, the VA intercepts the actions and the related flowmatch after they are decapsulated from the OpenFlow message[4] and before

[4]xDPd uses the ROFL-common C++ representation of the *FlowMod* which is version agnostic [15]

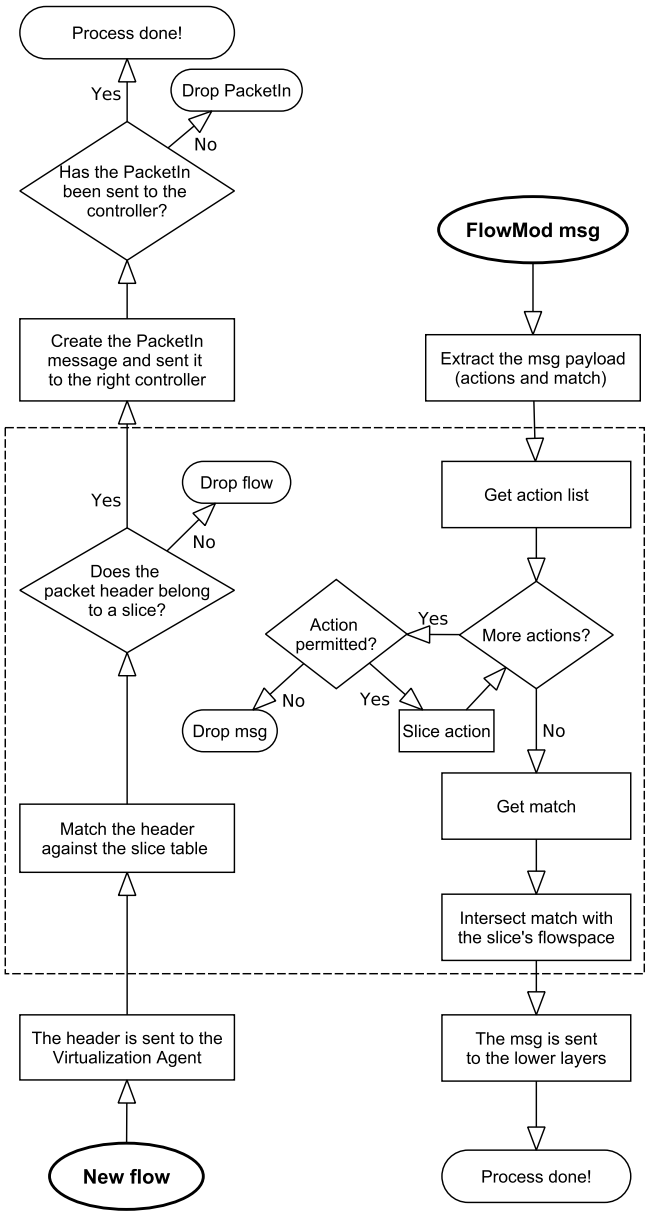

Fig. 2. The slicing process workflows for new flows and *FlowMod* messages (surrounded by the dashed box).

they are inserted into the switch's flow table. The actions are checked against the controller's flowspace (i.e. the VA checks if the controller is trying to control traffic outside its flowspace) and the match is intersected with the flowspace. The latter operation ensures that the actions are only applied to the flows matching the flowspace assigned to the controller, i.e. the VA prevents interference among different slices.

B. The datapath software architecture

A basic introduction of xDPd's software architecture is presented on this section in order to better understand the design principles followed for the VA component, however an in-depth explanation of xDPd's architecture is out of the scope of this paper. xDPd is a UNIX process that runs in the forwarding device, or as close as the forwarding device

as possible, like embedded systems next to ASICs. xDPd architecture is constituted by two pieces as depicted in Fig. 3; the Control and Management Module (CMM) and the platform driver or Forwarding Module (FM) which interface each other via an abstract API (AFA).

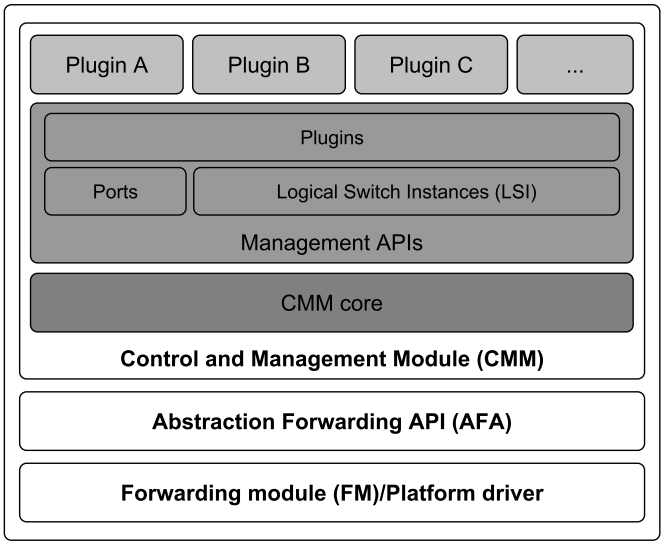

Fig. 3. The xDPd software architecture.

The CMM is shared between all the different platforms and encapsulates the platform independent code dealing with device or system management, monitoring and control plane (OpenFlow). On the management side, going top to bottom on Fig. 3, the CMM encompasses the plugin modules[5], a unified abstraction of the physical platform (fundamentally ports, either physical or virtual) and Logical Switch Instances (LSIs). The plugin modules can steer the configuration of LSIs (e.g. define the OpenFlow controller), including the attaching of platform ports to it. Examples of management plugins are NetConf/OFConfig agents or a file-based configuration reader. On the control plane side, the LSI abstraction encapsulates the OpenFlow endpoint and all the necessary control plane functionality, as well as having a handle to manage the forwarding state down in the platform driver. The control plane functionality can be further extended, like in the case of the VA, or even replaced, by control plane plugins.

The Platform driver or Forwarding Module contains the code that is platform specific and that is strongly tight to the particular forwarding device under control. The FM is in charge of presenting the device in an abstracted way to the CMM, including ports, device capabilities... and it does the basic management of the platform. For hardware forwarding devices, the Forwarding Module is in charge of updating and maintaining the forwarding state of the ASIC, whereas in a software forwarding device it may also include the packet I/O routines.

xDPd uses extensively a set of OpenFlow libraries called the Revised OpenFlow library (ROFL)[15]. ROFL includes libraries to build OF endpoints or agents, including OF protocol parsers and other utilities called ROFL-common, used extensively on the CMM. ROFL includes as well other libraries to help building datapaths, most notably ROFL-pipeline, which implements the data-model of the forwarding state of a multi-datapath (multi-LSI) forwarding device and the pipeline packet processing for software switches.

[5]Plugins are modules pluggable at compile time

C. Implementation

As mentioned in III-B, the Virtualization Agent has been implemented as a plugin for xDPd. Differently from other initiatives that are based on OvS (like the one proposed in [6]), our framework does not instantiate a virtual switch for each new slice. Instead, multiple LSIs (the virtual switches in the xDPd terminology) are only required when different versions of the protocol are used on the same physical switch. On the other hand, multiple controllers using the same version of the protocol are handled by the VA through the same OpenFlow endpoint encapsulated within a single LSI.

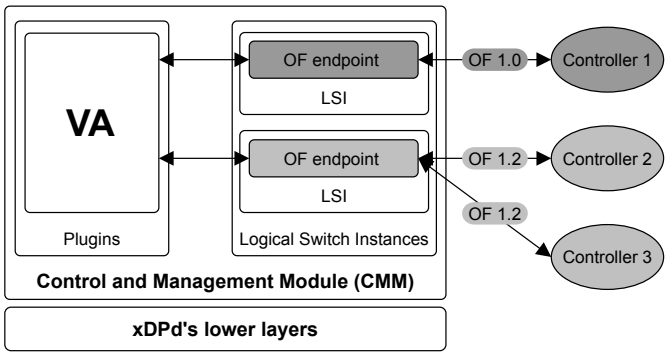

Fig. 4. The Virtualization Agent (VA) is the component in charge of the slicing mechanism and is implemented as a plugin for xDPd.

Referring to Fig. 4, two LSIs are instantiated to support protocol versions 1.0 and 1.2, while the VA interacts with the OpenFlow endpoints to perform the flowspace slicing operations. To maintain these operations OpenFlow version agnostic, the VA is queried by the endpoints before creating the OpenFlow messages directed to the controller and after the payload is decapsulated from the OpenFlow messages coming from the controller (as described in the workflow in Fig. 2).

IV. Evaluation and results

A prototype of the network virtualization architecture described in Section III has been tested in laboratory from a performance viewpoint. The tests have been performed focusing on the latency overhead introduced on the control channel by the VA operations for the most commonly used OpenFlow messages, i.e. *PacketIn*s. For these messages, we measured the "new flow time", namely the latency between sending a new flow to the switch and receiving the corresponding *PacketIn* message on the controller. Scalability considerations are also discussed in the last part of this Section.

The performance metric considered in our evaluation is the difference of latency overhead introduced by the VA and FlowVisor (currently the reference virtualization architecture for OpenFlow networks). With this purpose, we considered three different scenarios: (i) direct connection between a xDPd-enabled switch and the controller, (ii) same scenario with the

VA module enabled within xDPd and (iii) FlowVisor placed between xDPd (with the VA module switched off) and the controller (see also the logical paths A, B and C in Fig. 5 respectively).

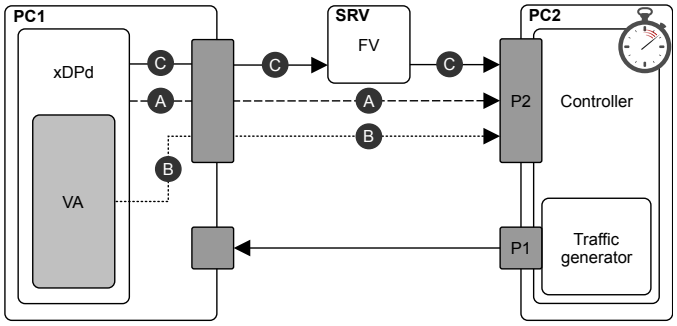

Fig. 5. Evaluation setup. A, B and C indicate the three test configurations considered in our evaluation. The arrows represent the direction of the traffic between two network interfaces.

Evaluation setup. The evaluation setup, represented in Fig. 5, was composed of two commodity desktop PCs (labeled as PC1, PC2) and one server-class machine (SRV) connected to each other via Ethernet cables. PC1 acted as OpenFlow-enabled switch by running an instance of xDPd plus the Virtualization Agent. PC2 hosted a modified version of the Ryu controller [16] that included a network traffic generator function and that was configured with version 1.0 of the OpenFlow protocol to meet the FlowVisor's requirements.
Finally, an instance of the latest release of FlowVisor[6] ran on SRV, a quad-core Intel Xeon 3.1GHz system equipped with 8GB of RAM memory (an equivalent system is currently used to host FlowVisor to slice the CREATE-NET's OFELIA island [8]).

Latency overhead comparison. To measure the latency overhead introduced on the control channel by the VA and FlowVisor, we performed three test sessions, one for the scenario A, one for B and one for C.
In each session, the interface labeled with *P1* in Fig. 5 was used to force the switch to generate *PacketIn* messages for the controller by sending as much packets as possible. More precisely, the controller's internal traffic generator was configured to: (i) send a single packet to the switch *PC1*, (ii) wait for a reply (the *PacketIn* message) on the interface *P2*, (iii) record the elapsed time between events (i) and (ii) and then (iv) repeat this process as quickly as possible.
Fig. 6 shows the CDF plots of the three test sessions.

Scalability. Optimally, the VA should be able to efficiently support as many virtual networks as xDPd and the physical substrate can manage. But, due to the overhead on the control channel caused by slicing operations described before in this paper, it might not be possible to accommodate so many of them in practice. To evaluate how the VA scales to large numbers of instantiated virtual networks, we repeated the scenario B of the *latency overhead* test with the VA configured with 100, 500, 1000, 2000, 5000 and 10000 flowspace rules. We forced the VA to match each *PacketIn* against all the entries in the flowspaces list (the worst case in a real-work deployment) before sending the message to the controller. The results are reported in Fig. 7.

[6] 1.4.0 at the time of writing

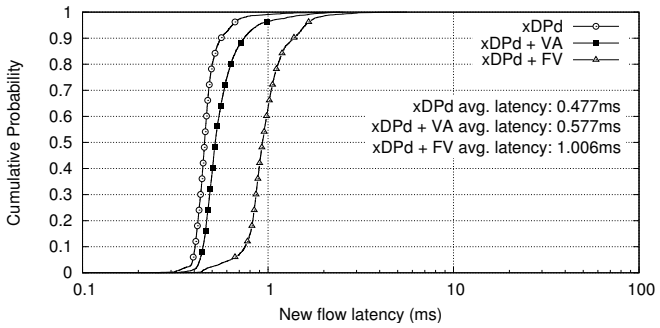

Fig. 6. Cumulative probability of the latency for *new flow* messages.

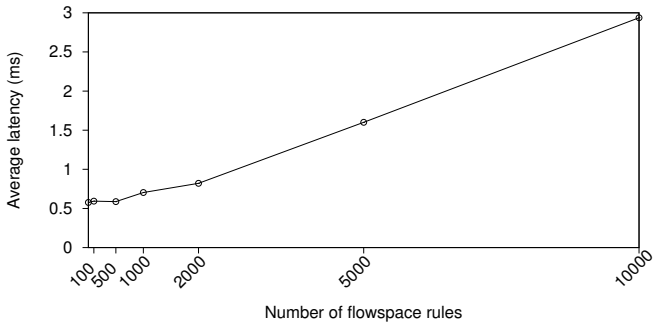

Fig. 7. Average latency for *new flow* messages on varying the number of flowspace rules.

In conclusion, the values reported in Fig. 6 show that even if the VA performed better that FlowVisor, the overhead introduced by the VA with *PacketIn* messages (about 18% on average) cannot be ignored, especially in deployments with high rates of new flows. In such conditions, we believe that controllers should proactively install flow entries on the switches as much as possible in order to avoid these additional latencies.

From the plot in Fig. 6 we can also notice a relevant difference in performance between the VA and FlowVisor (0.429ms on average). This gap is likely caused by specific processes that only FlowVisor introduces on the control channel: the OpenFlow protocol inspection and one additional TLS (Transport Layer Security) connection between the switch and the controller. In our specific setup, for each run of the *PacketIn* test, the control channel was involved only once to transmit the *PacketIn* message from the switch to the controller. However, in realistic deployments a new flow is forwarded towards the destination only after the *PacketIn* has been replied by the controller either with a *FlowMod* or with a *PacketOut* message. Therefore, in realistic conditions the gap between the VA and FlowVisor could be even more significant for the performance of the network.

Results of the scalability test (summarized in Fig. 7) show that the latency overhead on the control channel is fairly constant up to 500 flowspace rules and that it scales linearly from 500 to 10000 rules. Although these results were obtained by forcing the VA to scan the whole flowspace list (in practice,

the lookup process would stop when the first match is found), there is still room for optimization, including the replacement of the flowtable list with hashmap-like data structures.

V. Future work

The internal architecture of the VA and the way it interacts with the rest of the pieces of xDPd is still under constant evolution, for both performance and functional reasons. In particular, from the functional viewpoint, the architecture proposed in this work lacks some critical components to enforce bandwidth and flowspace isolation between slices.

Flowspace isolation permits two or more slices to share part of the flowspace and simultaneously prevent them from interfering with each other's traffic. We plan to achieve flowspace isolation at the Virtualization Agent Orchestrator level, where specific policies could prevent the definition/instantiation of a virtual topology whose flowspaces overlap with ones already in operation.

Bandwidth isolation allows the reservation of a portion of the nominal capacity of a link to a certain slice. I.e., a slice configured for X% of the bandwidth will always receive at least X% and possibly more if the link is under-utilized. In this case, we plan to operate at Virtualization Agent level by leveraging the xDPd mechanism for QoS enforcing.

VI. Conclusions

The increasing adoption of a Software-Defined Network (SDN) paradigm in all network segments is finally making real the vision of a network infrastructure that can be safely shared among several service providers via advanced network virtualization techniques. OpenFlow is the protocol used in most of the virtualization mechanisms proposed in the SDN literature. FlowVisor [3], the most popular network virtualization framework for OpenFlow networks, has several limitations: it may act as single point of failure; it is strictly based on OpenFlow v1.0; it introduces a non-negligible overhead that may introduce severe scalability issues.

In this paper a novel OpenFlow-based network virtualization mechanism has been proposed that overthrows FlowVisor limitations and leverages on a recent open-source datapath project named eXtensible Datapath Daemon (xDPd) [10]. This datapath engine is available for several hardware platforms and is designed to target carrier-grade SDN applications.

The proposed framework is based on a robust distributed virtualization architecture that is able to run on multi-version OpenFlow switch network scenarios. It also introduces the possibility to have a virtual switch being controlled by more than one controller thus increasing the flexibility when instantiating virtual networks to different controllers. A set of experiments in a realistic setting demonstrates that the proposed architecture introduces a marginal overhead from a performance point of view, which suggests it could be smoothly applied in a production environment.

Acknowledgements

The authors wish to thank Daniel Depaoli for his valuable support in the implementation and testing phase. This work is partially supported by the EU FP7 funded project ALIEN [12].

References

[1] M. Mendonca, B. N. Astuto, X. N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," in *Submitted to IEEE Communications Surveys and Tutorials*, June 2013.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 2, pp. 69–74, April 2008.

[3] R. Sherwood et al., "Can the production network be the testbed?" in *Proc. of USENIX OSDI*, Canada, 4-6 Oct. 2010.

[4] R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "Vertigo: Network virtualization and beyond," in *Software Defined Networking (EWSDN), 2012 European Workshop on*, 2012, pp. 24–29.

[5] P. Skoldstrom and K. Yedavalli, "Network virtualization and resource allocation in openflow-based wide area networks," in *Communications (ICC), 2012 IEEE International Conference on*, 2012, pp. 6622–6626.

[6] B. Sonkoly, A. Gulyas, F. Nemeth, J. Czentye, K. Kurucz, B. Novak, and G. Vaszkun, "Openflow virtualization framework with advanced capabilities," in *Software Defined Networking (EWSDN), 2012 European Workshop on*, 2012, pp. 18–23.

[7] "Global Environment for Network Innovations (GENI)," http://www.geni.net/.

[8] "OFELIA FP7 project," http://www. fp7-ofelia.eu.

[9] "Open vSwitch website," http://openvswitch.org/.

[10] "The eXtensible Datapath daemon (xDPd)," http://www.xdpd.org/.

[11] P. Skoldstrom and W. John, "Implementation and evaluation of a carrier-grade openflow virtualization scheme," in *Software Defined Networking (EWSDN), 2013 European Workshop on*, 2013, pp. 75–80.

[12] "ALIEN FP7 project," http://www.fp7-alien.eu.

[13] "OpenFlow 1.1+ support in Open vSwitch," http://openvswitch.org/development/openflow-1-x-plan/.

[14] "FOAM," https://openflow.stanford.edu/display/FOAM/Home.

[15] "ROFL," https://www.codebasin.net/redmine/projects/rofl-core/.

[16] "Ryu SDN framework," http://osrg.github.com/ryu/.