Customizing Data-plane Processing in Edge Routers

Customizing Data-plane Processing in Edge Routers

Fulvio Risso and Ivano Cerrato
Department of Computer and Control Engineering
Politecnico di Torino
Torino, 10129, Italy
Email: {fulvio.risso, ivano.cerrato}@polito.it

***Abstract*—While OpenFlow enables the customization of the *control* plane of a router, currently no solutions are available for the customization of the *data* plane. This paper presents a prototype that offers to third parties (even end-users) the possibility to install their own applications on the data plane of a router, particularly the ones operating at the edge of the network. This paper presents the motivation of the idea, the reason why we use OpenFlow even if it does not seem appropriate for the data plane, the architecture and the implementation of our prototype, and a first characterization of the system running in our lab.**

***Keywords*-Openflow; Software Defined Networking; Data plane processing.**

I. Introduction

OpenFlow [1] offers the possibility to customize the behavior of the *control plane* of the network. In fact, its original idea addressed the necessity to use real networks (instead of "toy" networks or simulations) to make experiments, enabling the deployment of many "virtual" networks on the same physical infrastructure. So far, less work has been done on the customization of the *data plane* of the network, which includes the applications that operate on (all) the traffic flowing through a network device, such as forwarding and bridging, network address translation, firewall, intrusion prevention systems, parental controls, etc.

The necessity to customize data plane processing is a very known problem for many entities, starting from end users that may be willing to inspect their traffic to/from the Internet, Network Service Providers (NSPs) that may need to optimize the network traffic, or content providers that would like to offer personalized services to end users. The possibility to install customized data plane applications on *edge routers* may bring more intelligence in the network and enable new features at lower costs, while core routers are expected to stay unmodified and keep forwarding data as fast as they can. For instance, we foresee that future routers will greatly improve their capabilities in terms of general purpose processing, and that they will include both traditional *network* linecards (e.g., expansion blades with network interfaces) and new *processing* linecards with different computing components (e.g., CPUs, GPUs, possibly specialized hardware component for network processing such as TCAMs, lookup tables, security accelerators, etc.), which can be used by our data plane applications. Obviously, computing components must be coupled with a huge amount of memory and an high speed interconnect used to transfer packets from one component to another. This way, our current data plane applications (which are often packaged as dedicated appliances) may evolve into a set of software images installed on the new router, which is being asked to provide a set of open primitives for supporting third-party software, instead of being locked with the software provided by the router manufacturer. In the end, this will result in lower capital and operating costs for the hardware, higher flexibility and scalability, and (potentially) more available services as new actors will be enabled to install their software on the routers.

This work focuses on the customization of the data plane of a network device and is based on two pillars. First, we offer the possibility to install *data plane applications* that can inspect and (potentially) modify the traffic in transit. Second, our network applications are *under the direct control of multiple actors* (e.g., end users, Network Service Providers, content providers). Those actors can install and manage *their* data plane applications operating on *their* network traffic (i.e., their *slice* of the network), without impacting on the services requested by other actors.

While the necessity of data plane customization seems clear, the idea to enable multiple actors to operate on data plane of the network usually raises some objections, as the most natural scenario consists in limiting this possibility to the router manufacturers or, at most, to NSPs. However, the authors are deeply convinced that only the active participation of new actors may bring a breath of fresh air in the networking world. For instance, end users, with their imagination, are the ones that drove the innovation in the PC and smartphone markets with the creation of many unexpected applications, and we expect them to be the ones that will contribute most to network evolution. In this respect, we envision for NSPs the possibility to evolve in *infrastructure providers* (a sort of *Network IaaS*), offering to multiple actors a pipe that transports bits (the network) and a programmable platform where those bits can be processed and even modified in transit.

This paper focuses on this new vision of a network edge node that supports custom data plane applications under the

control of multiple actors, presenting our long-term vision and the first steps toward the objective. Particularly, our current implementation is based on OpenFlow. Although this protocol was not proposed to handle data plane applications, OpenFlow represents the best option we have right now to demonstrate our idea, as it can redirect all the traffic of the data plane to an external controller. In our proof-of concept, the controller (which, according to the OpenFlow specifications, should handle the control plane) becomes a sort of data plane extension of the router in which we implement our customized processing. According to our speculations, in the future this external controller (in fact, a data plane extender) will be integrated in the router itself, which would be able to support any data plane applications through the new linecards oriented to generic processing presented above. Furthermore, although OpenFlow currently does not allow us to plan a production-ready deployment of our solution, we expect that the future OpenFlow 2.0 will offer a better support for data plane applications, as it should provide a more sophisticated model of the router and should be able to support also data plane features. Therefore, while our current prototype uses OpenFlow mainly because it represents the best technology in our hands (although far from perfect) and it offers us the possibility to create a testbed using real network devices, future OpenFlow version may look much more appropriate to our objectives.

This paper is structured as follows. Section II presents the related work, Section III describes our prototype, while Section IV offers the first numbers coming out from our preliminary implementation. Finally, Section V presents some conclusive remarks and gives some insights on our future work.

II. Related work

The idea of customizing the data plane of a network router was introduced several years ago by the Click modular router [2]. Other proposals, such as Shangri-La [3] and NetVM [4], enabled the transparent deployment of data plane software on multiple network processors (NPUs) through a powerful abstraction of the hardware. However, they targeted professional developers of network applications, i.e., the few employees of the major network manufacturers that develop those applications, and focused most on performance issues, e.g., demonstrating that high-level languages and NPU abstractions do not introduce notable performance limitations. Given the failure of those proposals in getting adopted in real products, we feel that the focus on the above two points was a mistake. Particularly, we feel that we should target also occasional developers, i.e., the ones that were never involved in network programming but that can make the difference in bringing new ideas in our world. Performance issues are important but we can address them later, when the advantages of our vision will be (hopefully) confirmed.

While previous proposals were never able to go beyond the research domain, recently network manufacturers started pursuing the same objectives by opening their platforms to third party applications. For instance, Juniper [5] and Cisco [6] released proprietary frameworks that enable the creation of network applications running on a limited set of routers, equipped by special service linecards. However, *(i)* the hardware has limited capabilities in terms of processing and memory, hence *(ii)* those solutions may not seem so appropriate for complex data-plane applications, and *(iii)* the control of those applications is firmly in the hands of the entity that operates the router. Furthermore, the exported API is *(iv)* proprietary and *(v)* oriented to very skilled programmers.

A solution based on OpenFlow can overcome most of these limitations. In fact: *(i)* it can be deployed on real networks thanks to the many manufacturers that support this standard; *(ii)* it is more oriented to "occasional" developers thanks to the simple API exported by the most advanced OpenFlow controllers; *(iii)* controllers are executed on a different box of the router, facilitating hardware upgrades when more processing power is needed. Other options beside OpenFlow are cited only for the sake of completeness. For instance, ForCES [7] never gained attraction in the industry, with few implementations limited to the research domain. Instead, the Web Cache Communication Protocol [8] is available on many network devices and it can be used to redirect a set of generic network flows to a controller that can inspect the traffic (in fact, version 2.0 does not limit its operations to web data). While this protocol can be used to perform custom operations on the data plane traffic, it does not allow to configure shortcuts in the switching plane of the network device, forcing all the selected traffic to go to the upstream controller. Consequently, although OpenFlow was not engineered for data plane processing and hence it does not guarantee optimal performance, it represents the best option we have right now to create data plane applications operating on real networks.

Within the OpenFlow domain, several works target the operations of a network edge node. However, as the best of the authors' knowledge, they focus on creating independent slices on the network, without going too much toward "traditional" data plane applications, and without giving end users the possibility to control those applications. For instance, [9] creates a network virtualization layer allowing multiple controllers, operated by different organizations, to coexist on the same network infrastructure. [10] proposes to partition the home gateway into independent slices that can be used by different entities (e.g., service providers) to provide different services. [11] focuses on security aspects, engineering a network edge node able to recognize users and apply the associated security policies. [12] proposes a language to program the network that guarantees the creation of independent slices.

Flow-based processing is another area of research. For instance, [13] suggests to use OpenFlow to create cluster of (virtual) machines to scale up the processing capabilities of the (virtual) network edge. However, although flow-based processing relates to data plane, we feel that data-plane applications cannot be reduced to a "simple" flow processing, as many aspects have to be considered, such as, who controls those applications, permissions, programming frameworks, etc.

III. The prototype

Our work focuses on the creation of **user-driven data plane applications** that operate on a **network slice** associated to a given **actor**, enabling the customization of the processing of the traffic inside the **network edge router**. We exploit **OpenFlow** (version 1.0) to build our first prototype, although in the future we may switch to other technologies (possibly, OpenFlow version 2.0) if those will be available.

A. Operating context

Our prototype operates in the context of a network edge node directly connected to the final users, as shown in Figure 1. Traffic coming from new hosts is redirected to a captive portal, which implements user's authentication. If the authentication is successful, a new OpenFlow controller is created that will receive all the traffic associated to the new network slice, defined by the $\{MAC_{user} \rightarrow *, * \rightarrow MAC_{user}\}$ tuple. This controller can host a set of (stacked) applications, usually installed by the user itself, which will be called sequentially on each user's packet. The calling order is (statically) chosen by the user; if an application modifies the content of a packet, all the following ones will receive the modified payload.

Finally, other controllers are created ahead of time by the entity that manages the network edge node (e.g., the NSP). Those controllers do no longer target end users; instead, they are dedicated to other actors (e.g., the NSP itself, a content provider, etc.) and can operate on the traffic of multiple users.

B. Network slices

Network slicing, i.e., the capability to create partitions over the network and to allow different actors to operate on their traffic, does not represent a novelty. However, those partitions were always oriented to guarantee network isolation, i.e., slices do not overlap.

As our objective is not to provide network isolation, we envision two types of slices that can overlap. *Vertical slices* guarantee network isolation between hosts and are created upon the successful authentication of the user. As a consequence, each packet will match the network tuple (based on the host MAC address) associated with only one slice, unless some special cases, such as the traffic between two hosts that are both attached to the same router, or the broadcast/multicast traffic[1]. Vice versa, *horizontal slices* are statically provisioned by the manager of the edge router and match the traffic specified by the proper network tuple, given as configuration parameter. Those slices are oriented to actors that setup services shared among multiple end hosts, such as a caching engine, a network optimizer, or the bridging process. Horizontal slices can be either associated with all the traffic of a given set of users (e.g., a network firewall, active over a subset of the connected hosts), or be limited to a portion of the users' traffic (e.g., a web caching engine, that intercepts only web traffic).

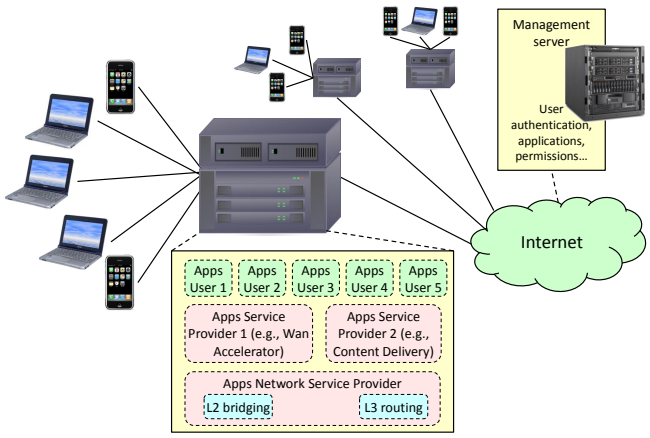

Figure 1. The operating context: a network edge node and its end users.

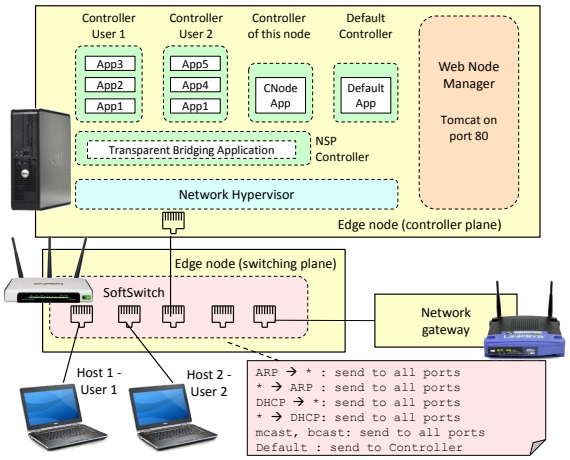

Figure 2. The architecture of the edge router in our earlier prototype.

C. The overall architecture

As depicted in Figure 2, the architecture of our edge router is composed by the following components.

The **softswitch** runs at the very bottom of the network stack, providing the basic switching capabilities to our sys-

[1]Currently, multicast and broadcast are not supported; those packets are flooded on all the interfaces, without delivering them to any controller.

tem. It matches each incoming packet against a list of flow tuples and, based on the result, the packet is either sent to a specific output port or to the controller. Our prototype uses the vanilla Open vSwitch for the sake of simplicity, but it can be any router supporting OpenFlow 1.0. Our softswitch, as shown in Figure 2, is configured with a few rules: multicast, broadcast, DHCP and ARP packets are flooded to all ports, while the rest of the traffic is redirected to the controller. Optionally, user applications can install new flow tuples that instruct the softswitch to send traffic directly on the output port, without involving the controller. However, this feature has not been implemented right now and it is left to the future work.

The **network hypervisor**, based on FlowVisor [9], enables the slicing we need in our prototype and it allows an OpenFlow node to connect to multiple controllers. The hypervisor receives packets from the softswitch and it sends them to the controllers of all the slices the packets belong to, according to the order defined by the priority associated with the flow rules identifying the slices themselves. Furthermore, it verifies that the tuples an application may want to instantiate in the softswitch are compatible with those assigned to the slice the application belongs to. Finally, the hypervisor isolates controllers from the physical hardware of the edge router, allowing each of them to use the resources of the router as it was the sole user. All the communications from the softswitch to FlowVisor, then from FlowVisor to the controllers, use the OpenFlow protocol.

Controllers (detailed in Section III-D) are special containers that execute our data plane applications. A wrapping framework receives traffic from the network hypervisor and dispatches it to each application. Traffic is then sent again to the hypervisor when the processing chain inside the controller has been completed.

An **embedded web server** is available on our edge node, to provide both global management and configuration services. For instance, it implements the captive portal and a set of web services that enable data exchange between the different components (hosts, edge router, management server). For example, one of those web services is used by the management server to check if a controller is still active and to keep up-to-date the status of the system.

The **network gateway** (a Linksys WRT54G) connects our components to the Internet. This box has been used in this first prototype because it implements two functions that are needed in our system, namely the DHCP server that assigns IP addresses to new hosts and the routing process that allows our edge "router" to implement only the (simpler) transparent bridging. However we plan to remove this component in the next steps of the project and to integrate those functions in our edge node.

Finally, an external **management server** coordinates the entire set of edge routers. It contains the user database, permissions, the list of applications to be installed, etc. Furthermore, it stores the applications associated with each user, which in fact are copied from this server to the edge router each time a new controller has to be activated, e.g. each time a user logs in.

D. User controllers

The technology used to implement the controllers is very important in our system and we made our choice based on the following four guidelines. First, the controller must be extremely **lightweight**, as the forecasts from some NSPs known by the authors indicate the necessity to handle about 100K hosts, hence 100K controllers, in each BNAS, i.e., the edge router in which ADSL users sessions are terminated. Second, the API exported by the controller must be compatible with **high-level programming languages**: authors are deeply convinced that the success of smartphones is also due to their programming languages, such as Java for Android, and Objective-C for iOS. Third, network applications should be created by **non-skilled programmers**, i.e., we do not require developers to be too much aware of how networks work, because this would impose a strong limitation on the potential number of developers for our platform. Fourth, as multiple controllers coexist on the same edge router, the system must be able to provide **computing and memory isolation** in order to allow multiple actors to operate safely on the same hardware.

Given the proof-of-concept nature of our system, we decided to base our prototype on the available software components whenever possible, leaving more optimizations to a future refinement phase. Starting from the existing OpenFlow controllers, we initially selected Beacon [14], based on Java, and NodeFlow [15], based on Node.js. Both solutions support a single language for the development of applications, but that language is very powerful, well known, and features many third party libraries. Furthermore, both are based on lightweight virtual machines that can provide acceptable computing isolation among controllers. FloodLight, also based on Java, was discarded because it does not allow to load/unload dynamically new applications without killing the entire controller, which represents a mandatory feature in our prototype.

Memory requirements, for both Java and Javascript VMs (i.e., an empty VM with a simple "hello world" program) are shown in the first part of Table I. The Javascript VM appears lighter, and this was confirmed when launching the OpenFlow controller (second part of Table I). Initially the memory occupancy of Beacon was rather high, but we were able to reduce it to a more reasonable value by removing some components that were not needed in our prototype. Although at the first sight NodeFlow looks a better choice because of its reduced memory requirements, we selected Beacon as the foundation of our controllers because it appeared more mature and stable. For instance, NodeFlow showed an unexpected increase in the memory occupancy

Table I
MEMORY OCCUPANCY FOR THE SELECTED OPENFLOW CONTROLLERS

Intel i5-3450S, 4GB RAM, OS Debian 7, 32 bits	
Description	Memory
Java (Oracle JRE 1.7, 32 bits) with "Hello World' app	24 MB
Node.js (v. 0.6.12) with "Hello World" app	5 MB
Beacon (standard controller)	102 MB
Beacon (reduced controller)	43 MB
NodeFlow (standard controller)	8 MB

when the controller was flooded with traffic, at least in our system.

Computing isolation among controllers is guaranteed by running each controller in a different Java Virtual Machine (JVM), which translates into different processes (with disjoint memory spaces) in the edge router. However, applications within the same controller share the same address space, although this was considered reasonable as those applications belong to the same user. Unfortunately, common JVMs such as Oracle JVM or OpenJDK cannot control the CPU and memory used by each process, which could trigger denial-of-service phenomenons when a controller consumes too many resources. It is worth noting that this limitation comes from the JVM we use and it can be addressed in our future releases.

E. Users, Groups and Permissions

Our current prototype supports users and groups associated with vertical slices; vice versa, horizontal slices cannot be assigned to groups.

The owner of a slice can upload new applications in the system; a new application will become active only when explicitly installed and upon the definition of its calling order within the controller. When uploading an application, the owner can specify who can install it (only the owner himself, all the members of its group, or everybody) using the *availability* parameter. Furthermore, group administrators can install applications in the slices of their group members and choose, through the *visibility* parameter, if that application is visible from the user or if it operates in "hidden" mode. An hidden applications, for instance, could be a parental control installed by a parent to its kids; this way, kids can manage their slice (adding/removing additional applications) without noticing that the parental control is active. An additional parameter, available only if the *visibility* is turned on, determines if the user can remove that application from his slice. Users do not have any control (nor visibility) on the applications installed in other slices, even if their traffic crosses them as well (e.g., the transparent bridging application in Figure 2).

Finally, some *network privileges* are associated with controllers. For instance, normal users are usually enabled to do whatever they want on *their* traffic, including generating and/or modifying packets within the controller. A controller of an entity in charge of network monitoring may have instead a "read mode" privilege. Further, other controllers could have also access to network parameters and influence the forwarding process of the node, such as determining the output interface for a given packet. Usually, the last privilege is allowed only in the controller owned by the NSP.

F. Configuring and monitoring the applications

Each application running in the controller is requested to implement two set of functions. The former includes a set of callbacks operating on network traffic that are called upon each packet arrival. The latter enables an (optional) set of web services for monitoring and configuring the application itself, which can be reached through a special URL composed by the default locator for the user controller (i.e., `http://config.ctrl`) followed by a string including the application name. All the URLs received in that format are checked for permissions (e.g., a final user cannot manage an "hidden" application, although this can be done by the group administrator), then are redirected to the web services exported by the application. Note that both the semantic and the syntax of the data exchange is completely application-dependent.

G. Implemented applications

Given the early stage of our work, four simple applications were developed. **NetMon** is a network monitor that extracts some statistics on the traffic on the selected slice (traffic sent/received per each IP address, the latest TCP connections, etc). **DNSFilter** is a DNS-based filtering for parental control that checks DNS requests and drops all those that are directed to sites belonging to a deny list. **GSafe** modifies all the requests to the Google search engine, preventing it from returning disturbing results. Finally, **TransBridge** implements the transparent bridging process.

While those applications allowed us to setup some nice demonstrations in our lab, we experienced some severe limitations when developing richer applications. For instance, traffic is received *packet by packet*, resulting in a huge amount of work when the payload has to be modified (e.g., adding some bytes to an HTTP request). This is because low level tasks such as TCP reassembly and session tracking had to be implemented from scratch, as currently no high-level libraries are available for those (annoying, but needed) tasks.

IV. EXPERIMENTAL RESULTS

A first prototype was built using an Intel Core 2 Duo P8400 PC as controller and a TP-LINK TL-WR1043ND access router with the OpenWRT Pantou release for the fast path. However, this platform was later discarded for two reasons. First, the throughput was not satisfying, e.g., the traffic between a user attached to the edge router and a remote server was about 55Mbps, with a single user controller running the DNSFilter application. Second, the

limited hardware of the TP-LINK (32MB RAM and 8MB Flash) prevented us from installing all the components (switching and control planes) on the same node, which would lead to a more integrated solution and may open the path to more aggressive optimizations.

As a next step, we set up an integrated box based on an Intel i5-3450S processor with four physical cores, 4GB RAM and 64GB SSD. We added also a WiFi adapter and an Intel PCI-E card with 4-GbE ports in order to mimic the hardware of a typical home access gateway. On that box we integrated both the data path (the vanilla Open vSwitch) and the "control" components (i.e., FlowVisor, controllers, web node manager). The machine was preloaded with the Linux Debian 7 operating system running at 32 bits.

The test setup included two hosts connected to the same edge router, which has five controllers: two user controllers, one associated to the edge router itself, one owned by the NSP that hosts the TransBridge application, and the default controller. Table II shows the latency of the system, split in three components. The first measures the time required by a packet from the NIC to the first controller and it includes also the time spent in Open vSwitch and FlowVisor. The second takes into account the time spent in the controllers (e.g., user controllers, NSP controller), including the time consumed by FlowVisor to redirect the packet to the second controller. The third measures the exiting time, i.e., the last hit in Flowvisor, Open vSwitch, and the output NIC. Latency has been measured collecting the timestamp of the selected packets with `tcpdump` and averaged over 1000 samples.

The first three tests have been done on an unloaded network, exchanging ICMP traffic between the two hosts and changing the number of applications installed in each controller (from zero to three). Unexpectedly, the most part of the time is spent in the low level components, such as the NIC, Open vSwitch, FlowVisor. With respect to the time spent in the controllers, this seems to be rather independent from the number of applications installed, which suggests that the framework that implements the controller (i.e., Beacon) represents an important bottleneck, impacting even more than the application itself. The fourth test is still done on an unloaded network, but in this case the traffic is represented by an HTTP request to the Google search engine, which has to be modified by the GSafe application. Finally, we repeated the last test by adding a cross traffic made by two large FTP transfers (from Host 1 to Host 2 and vice versa), accounting for about 350 Mbps each. Also in this case we cannot see any noticeable difference in the latency of the traffic crossing our edge node.

Table III shows the memory occupancy of the different components of the system, in the conditions of the fifth test mentioned above. The system reported 1072MB allocated memory, while each user controller uses in average 165MB, although this value largely depends on the installed applications (e.g., DNSFilter includes a set of forbidden sites that, by itself, accounts for 77MB). The other three controllers host simpler applications and weigh in the range of 40-60 MB. The memory consumption of Open vSwitch is negligible, about 8 MB; FlowVisor instead uses about 150MB.

Table II
LATENCY IN THE CONTROLLER

Test description	NIC - FlowVisor	Controllers	FlowVisor - NIC
1) No apps (ICMP)	215 μs	120 μs	233 μs
2) NetMon (ICMP)	226 μs	146 μs	237 μs
3) NetMon, GSafe, DNSFilter (ICMP)	210 μs	131 μs	229 μs
4) NetMon, GSafe, DNSFilter (HTTP GET)	150 μs	143 μs	190 μs
5) NetMon, GSafe, DNSFilter (HTTP GET, loaded network)	221 μs	154 μs	223 μs

Table III
MEMORY OCCUPANCY WITH LOADED NETWORK (TEST N.5)

Component	Memory occupancy
Controller user 1 / user 2	165 MB / 165 MB
Controller edge router	60 MB
Default controller	46 MB
NSP controller	48 MB
FlowVisor	150 MB
Open vSwitch	8 MB
Tomcat web server	156 MB
Operating system	274 MB
Total	1072 MB

Table IV
CPU CONSUMPTION WITH LOADED NETWORK (TEST N. 5)

Component	CPU consumption (over a single core)
Controller user 1 / user 2	35% / 35%
Controller edge router	0%
Default controller	0%
NSP controller	11%
FlowVisor	81%
Open vSwitch	100%
Operating system	2%
Total (over four cores)	66%

Table IV reports the CPU consumption of the different components; the bottleneck is represented by OpenvSwitch that saturates its CPU core; the CPU occupancy of FlowVisor is not negligible at all, reaching 81% of its CPU core. Controllers appears lighter, reaching no more than 35% of their CPU core. The total CPU consumption in this test appears to be 66%, which results by computing the average load over the four cores present in our CPU.

V. CONCLUSIONS

This paper presents the first implementation of an edge router that enables the customization of its data plane processing. Currently, data-plane applications are very common at the edge of the network, but are under the control of manufacturers (which build boxes and software) and NSPs,

which takes care of their deployment. In our proposal, multiple actors (content providers, NSPs, end users) can install and manage their own data plane applications that operate on the portion of the network traffic associated to the entity itself, on the same edge router. Isolation is guaranteed in terms of network traffic (each application operates only on its network slice) and (reasonably) in terms of computing and memory (a malfunctioning application does not affect the processing on other slices). User permissions and groups are supported; administrators can manage the data plane processing of their users. Finally, network slices can overlap, offering the possibility to have multiple actors operating on the same data for different purposes (e.g., end-users handling their traffic, content providers, etc.).

The numbers that come out of this first incarnation of our prototype are satisfying, particularly considering that the bottleneck does not appear to be our software; in fact, other components (Open vSwitch, FlowVisor) seems to be the most critical blocks. This provides an insight to our claim that OpenFlow, at least in its current incarnation, is not the technology that can guarantee an efficient implementation of our ideas, and that a more accurate implementation could obtain acceptable performance also on much less powerful hardware. However, our choice of OpenFlow allowed us to create the prototype in a very short time by reusing many existing components (with some modifications) and to demonstrate our solution also on real hardware.

The prototype running in our lab is still a proof-of-concept and many items have to be investigated in order to transform this idea into a mainstream technology. In our opinion, the biggest problem is the necessity to evolve the OpenFlow specifications in order to support data plane functions as well, as the current Openflow model targets the control plane of the router. This would enable the creation of data plane applications that will be portable across different devices (and vendors) and that will be able to exploit efficiently the hardware resources present in each router.

ACKNOWLEDGMENT

The authors would like to thank M. Cita and M. Pramotton, who take care of the implementation of the prototype, M. Ullio, V. Vercellone, F. Invernizzi, R. Milito, M. Nemirovsky, P. Monclus, M. De Benedetto, G. Borgione, M. Leogrande and the many friends who contributed to the definition of the idea.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ser. SOSP '99, 1999, pp. 217–231.

[3] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-la: achieving high performance from compiled network applications while enabling ease of programming," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, 2005, pp. 224–236.

[4] O. Morandi, F. Risso, P. Rolando, S. Valenti, and P. Veglia, "Creating portable and efficient packet processing applications," *Design Automation for Embedded Systems*, vol. 15, no. 1, pp. 51–85, Mar. 2011.

[5] J. Networks. (2012) Junos software development kit. [Online]. Available: https://developer.juniper.net/content/jdn/en/develop-overview/junos-sdk/getting-started.html

[6] Cisco. (2012) Application extension platform. [Online]. Available: http://www.cisco.com/en/US/products/ps9701/index.html

[7] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, "Rfc 5810: Forwarding and control element separation (forces) protocol specification," Internet Engineering Task Force, Mar 2010, status: Standards track.

[8] M. Cieslak, D. Forster, G. Tiwana, and R. Wilson, "Internet draft: Web cache communication protocol v2.0," Internet Engineering Task Force, Apr 2001, status: Internet Draft.

[9] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010, pp. 1–6.

[10] Y. Yiakoumis, K.-K. Yap, S. Katti, G. Parulkar, and N. McKeown, "Slicing home networks," in *Proceedings of the 2nd ACM SIGCOMM workshop on Home networks*, ser. HomeNets '11, 2011, pp. 1–6.

[11] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: dynamic access control for enterprise networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ser. WREN '09, 2009, pp. 11–18.

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

[13] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy, "Flow processing and the rise of commodity network hardware," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 2, pp. 20–26, Mar. 2009.

[14] D. Erickson. (2012, Apr) The beacon openflow controller. [Online]. Available: http://www.beaconcontroller.net

[15] G. Berger. (2012, Jan) The nodeflow openflow controller. [Online]. Available: http://garyberger.net/?p=537