Flow-based QoS forwarding strategy: a practical implementation and evaluation

Santiago Bentancur, Martín Fernández Bon, Gabriel Gómez Sena, Claudina Rattaro, Ignacio Brugnoli
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
{sbentancur, mfbon, ggomez, crattaro, ibrugnoli}@fing.edu.uy

***Abstract*—During the last decade we have seen an explosive growth in the deployment of cloud applications and services. In this context, one of the challenges is Quality of Service (QoS) management, which is the problem of allocating resources to the applications to guarantee a service level along dimensions such as performance, availability and reliability. Compared with the traditional best-effort service model based on BGP and the classical tunnelling alternatives (like MPLS), software-defined-networking (SDN) has the potential to provide a better QoS guarantee for cloud applications and services due to its centralized control, network-wide monitoring and flow-level scheduling. With this in mind, we propose a QoS-aware overlay routing based on the SDN architecture, which consists on the rewriting of IP address and TCP/UDP ports in order to be able to force the packets to follow the desired path on the overlay. We evaluate the proposal through both simulations and practical implementation analysis. In particular, we test our solution using two of the most popular SDN controllers: ONOS and OpenDayLight.**

***Index Terms*—Software Defined Network, Forwarding strategy, ONOS, OpenDayLight, Quality of Service**

I. Introduction

Providing end-to-end Quality of Service (QoS) at the global Internet is known to be a difficult task because traffic follows BGP routing policies which do not take QoS parameters into account. As solutions involving changes to BGP protocol or proposing an external protocol replacement are not likely to be adopted, the main issue to provide end-to-end QoS is to avoid BGP chosen paths if we can find alternative routes with better performance metrics. Avoiding BGP paths can be achieved with overlay networks which provide a way of controlling Internet flows without involving changes to the Internet Service Provider's (ISP) infrastructure. By dynamically optimizing routes at the overlay network, it is possible to overcome connectivity disruptions due to BGP outages or lack of quality of service up to a moderate number of nodes [1] [2] [3].

Current cloud applications and services rely many times on virtual servers at several datacenters distributed around the world and often require high resilience and application-dependent QoS. In this sense, the main use case for this paper is to provide flow based QoS for several virtual servers running at distant locations avoiding the collaboration of the ISPs. Our approach for scalable, QoS-aware overlay routing is based on the Software-defined networking (SDN) architecture, which provides a centralized control of the forwarding devices, adding flexibility, vendor independence and allowing to perform routing decisions in an optimised way. Our solution [4] allows to build a pure IP overlay network without the need of classical tunnelling techniques like IP-in-IP or MPLS, therefore providing a flexible, scalable and ISP independent architecture. Once we decide which path a flow must follow on the overlay to achieve the required QoS metrics, a forwarding strategy must be pushed to the SDN switches. The proposed strategy [5], as briefly described in Section III, is based on the rewriting of IP address and TCP/UDP ports in order to be able to force the packets to follow the desired path on the overlay.

Some preliminary results were published in our previous articles [4] [6]. In this paper we present a more general solution being our main contribution the implementation and deployment in a real environment, integrating commercial OpenFlow switches and considering two of the most popular SDN controllers: Open Network Operating System (ONOS) and OpenDayLight.

The rest of the paper is structured as follows. In Section II we analyze some related works and in Section III we briefly describe our flow-based QoS forwarding strategy. In Section IV we introduce the SDN controllers and we present the main characteristics of the simulated and real network implementations (for instance, we describe hardware, topology, etc). In Section V we present our main results and additionally we include some practical detected issues and its possible solutions. Finally, we conclude in Section VI.

II. Related work

Performing traffic engineering by appling custom routing policies on an overlay network has been proposed by various authors. In [2] the overlay network uses IP-in-IP encapsulation and the overlay propossed in [1] is based on overlay servers which use a custom overlay header. The overlay propossed by [7] also uses IP-in-IP encapsulation between dedicated overlay servers. Any kind of tunneling technology on an overlay network can provide the ability to perform traffic engineering. On the other hand, we propose a pure IP overlay network architecture avoiding any extra headers or encapsulation technique. The approach is feasible by making use of the SDN technology, which enables a centralized control of the network traffic. Moreover, the solution enables a fine grain flow based QoS management.

III. Forwarding strategy overview

Fig. 1 represents a simplified overlay network with four points of presence (blue circles, $PoPi$) interconnected by the ISP's routers (orange, Ri). As stated, it is possible that the green path may have better QoS metrics than the straightforward BGP path. The proposed forwarding strategy allows any selected TCP or UDP flow to be forwarded through the desired path just by modifying the packet headers, without affecting the MTU nor managing any tunnels. Based on the SDN paradigm, which provides a centralized view of the network, we can take global decisions at the SDN Controller level and instruct the OpenFlow switches at the overlay network to perform the desired actions on the traffic.

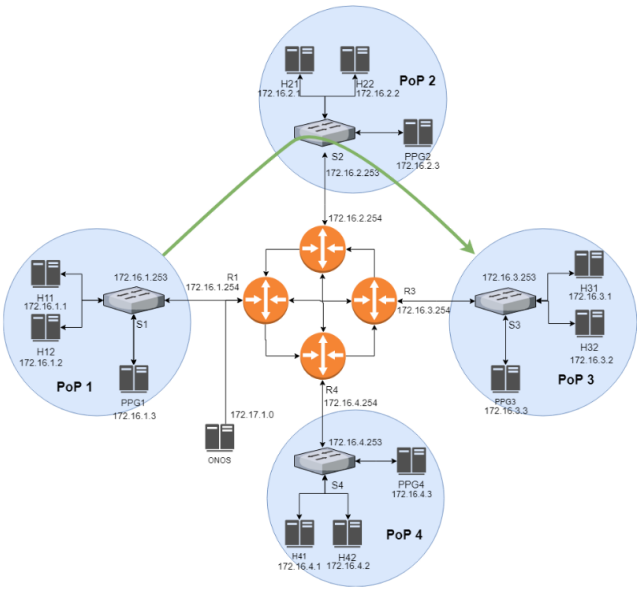

Fig. 1. Overlay topology with four points of presence and alternative path.

Let us now expose the proposed algorithm. If we want to forward the traffic from host $H11$ to host $H31$ through OpenFlow switch $S2$ (Fig. 1), the main idea is to change the packet destination IP address at switch $S1$ to an IP address belonging to switch $S2$. When the packet arrives at switch $S2$, the destination IP address is changed again to the final destination IP address $H31$. To fully implement this initial idea, a more general and complex forwarding strategy is needed. Considering the reverse path filtering [8] configuration surely enabled at the provider routers Ri and also to be able to provide the ability of managing individual TCP or UDP flows, the proposed strategy includes the modification or both source and destination IP address, and also both source and destination L4 ports on the way. The change of IP addresses on the way will allow the packets to follow the desired path, the source L4 port is used to carry the local identification of the specific routing policy applied, and the destination L4 port enables the retrieval of the original headers at the last OpenFlow switch (in the example $S3$). The detailed explanation and justification of the flexible and scalable proposed solution to provide fine grain flow QoS management are primary presented in [5] and will be also addressed in other articles.

To illustrate the idea more clearly, suppose a particular UDP flow from $H11$ (ephemeral port 1024) to $H31$ port 1200, wants to be routed through the green path shown in Fig. 1. Using the proposed forwarding strategy a packet belonging to this flow will appear at the network links as is shown in Table I.

TABLE I
UDP flow from H11:1024 to H31:1200

	H11 Host	**S1 to S2 link**	**S2 to S3 link**	**H31 Host**
Src IP	172.16.1.1	172.16.1.253	172.16.2.253	172.16.1.1
Dst IP	172.16.3.1	172.16.2.253	172.16.3.253	172.16.3.1
Protocol	UDP	UDP	UDP	UDP
Src port	1024	1500	1501	1024
Dst port	1200	4000	4000	1200

As shown, the source and destination IP address are being changed on the way in order to make the packets follow the desired path. Source ports on the way work like virtual circuit identifiers (in the example 1500 for the S1-S2 link and 1501 for the S2-S3 link) and destination port (4000 in the example) indexes a global table at the controller level needed to retrieve the original headers of each flow at the final switch. It is essential to be noted that the arriving packet at destination host $H31$ will have the initial headers originated at host $H11$, therefore the forwarding strategy is transparent to the end-points.

IV. Implementation and Demonstration

The solution is implemented over two Controllers, Open Network Operating System (ONOS) [9] and OpenDayLight (ODL) [10], chosen by its market adoption and features. Before introducing the demonstration results, let us now describe those controllers and some details of our technical implementation.

A. SDN Controllers

There is a variety of controllers and platforms to consider when selecting an SDN strategy, some of the well known projects are NOX, POX, Floodlight, ODL, ONOS and RYU. Because of their popularity [11], great level of documentation, the vast number of features and the ease for the development of new applications, we choose ODL and ONOS for this deployment [12] [13].

1) Open Network Operating System: The proposed forwarding strategy has been developed as an application running over ONOS called Overlay Network Routing Application (*ONRApp*) to perform the automatic management of the overlay topology and the route management processes, besides other functions (visit Github project link in [14]). The *ONRApp* application creates an abstraction layer between the SDN application plane and the complex process of implementing the forwarding policies as well as other network functionalities. The implemented services are exposed by *ONRApp* through a

REST API so that external entities can automatize and manage all the required functions through POST and GET methods using JSON format for sending and receiving parameters. The ONOS version used for this work is 2.1.0.

2) OpenDayLight: ODL project is another open-source platform that uses open protocols to provides a centralized control and monitoring of the network devices [10]. We have not developed an analogous *ONRApp* based on ODL yet, so in this case we have worked with static switch configurations. The ODL version used for this work is Beryllium-SR4.

As a first step, using Mininet emulator [15] a network composed of interconnected OpenFlow switches was created and its flow tables were statically configured using ODL. We conclude that to properly configure a network environment using ODL, we have to enable some features to expand its networking capabilities [10], as ODL has no pre-installed features by default. In particular, the features that need to be enabled are: odl-OpenFlowplugin-all, odl-restconf and odl-l2switch-all. Those features allow the use of OpenFlowplugin, an ODL related project [16], which enables a REST API running over HTTP, which allows access to data defined in YANG (RFC 7950) and provides classic L2 (Ethernet) forwarding across connected OpenFlow switches.

The ODL controller provides two RESTCONF interfaces (RFC 8040) in order to program the OpenFlow devices: the Configuration Datastore and the RPC Operations. As the last one is not persistent, we use the Configuration Datastore [16] which accepts requests in both JSON or XML format, using the former for the current implementation.

B. Emulated Overlay network

The topology represented in Fig. 1 is emulated with Mininet [15] and the test consist in being able to forward UDP and TCP packet streams through different paths on the overlay network. Before getting into the details of the traffic types evaluated, it is important to introduce the concept of Overlay Network Routing Policy (ONRP). This is the rule that defines which path will follow certain type of traffic, defined by the following expression:

$$\begin{aligned} ONRP = \{&ONRP\_id, Priority, \\ &Src\_Subnet, Dst\_Subnet, \\ &Src\_Port, Dst\_Port, \\ &L4\_Protocol, Path, ONAT\_Id\} \end{aligned} \tag{1}$$

It can be seen that each ONRP is identified by an identifier called ONRP_id and it is specified by the origin and destination sub-networks, the layer 4 ports, a sequence of points of presence that the matching traffic must follow from origin to destination and an associated priority. Assigning a level of priority to each ONRP improves the flexibility of the solution to implement complex routing scenarios. Table II shows 5 ONRPs created in a test environment to validate the flow identification algorithm. As it can be appreciated, it is possible not to specify some fields such as source port, destination port and layer 4 protocol. In these cases, the priority field is particularly important since, for example, if the host located at $PoP1$ whose IP address is 172.16.1.2 sends traffic to the host located at $PoP3$ whose IP address is 172.16.3.2, with source port 40005 and destination port 40006, it would be impossible to decide whether the matching ONRP is ONRP 1 or ONRP 5.

To validate the proper forwarding of the different flows, a sniffer is used to view the incoming traffic at any desired point of the network.

C. Real Overlay network

To validate the forwarding algorithm in a real scenario, a testbed with commercial switches is deployed. Fig. 2 and Fig. 3 show the main components of our real scenario.

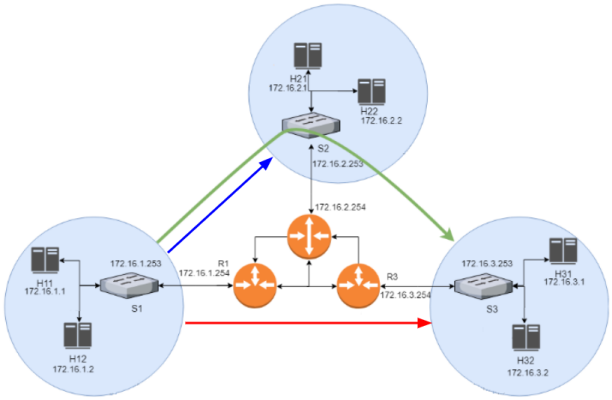

Fig. 2. Implemented Overlay topology.

1) Internet network is implemented by three Mikrotik RouterBOARD 433AH ($R1$, $R2$ and $R3$).
2) Overlay network is implemented with a Pica8 switch (model P3297, PicOS version 2.6.4 [17]) supporting OpenFlow 1.3. Three bridges are created, so as to provide three logic switches ($S1$, $S2$ and $S3$) one for each point of presence.
3) Controllers, one PC with ODL and another with ONOS.
4) Private network (clients $H11$, $H21$, $H31$, etc) are implemented with laptops.

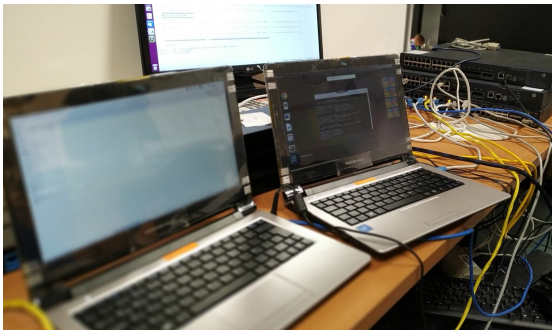

Fig. 3. Testbed scenario.

Two main representative scenarios for validating the forwarding strategy are deployed: (1) Basic Algorithm Validation

TABLE II
ONRPs IMPLEMENTED TO VALIDATE THE FORWARDING ALGORITHM IN THE SIMULATED NETWORK.

ONRP_id	1	2	3	4	5
Src_Subnet	172.16.1.2/32	172.16.1.2/32	172.16.1.2/32	172.16.1.0/24	172.16.1.0/24
Dst_Subnet	172.16.3.2/32	172.16.3.2/32	172.16.3.2/32	172.16.3.2/32	172.16.3.0/24
Protocol	UDP	UDP	UDP	TCP	*
Src_port	40005	40003	40002	4300	*
Dst_port	40006	40004	40003	80	*
Path	S1,S3	S1,S2,S4,S3	S1,S2,S3	S1,S2,S4,S3	S1,S2,S3
Priority	8224	8224	8224	4120	792

and (2) Flow Identification. Both are tested using ONOS and ODL Controllers. In (1) we generate UDP traffic according to Table III and the forwarding policy according to the green path illustrated in Fig. 2. On the other hand, in (2) we generate different traffic flows according to Table V.

V. EXPERIMENTS AND EVALUATION

The results obtained in the real overlay network are exposed below. The results over the simulated scenario are totally analogous but using a more complex topology and therefore considering a greater variety of paths (see for instance the ONOS simulated results in [5]).

A. Basic Algorithm Validation

In this basic proof, one ONRP is implemented (see Table III).

TABLE III
ONRP FOR BASIC ALGORITHM VALIDATION.

Src IP	Dest. IP	Src. port	Dest. port	Path
172.16.1.1	172.16.3.1	43000	8080	S1-S2-S3

TABLE IV
BASIC ALGORITHM VALIDATION: UDP FLOW FROM H11:43000 TO H31:8080.

	H11 Host	S1 to S2 link	S2 to S3 link	H31 Host
Src IP	172.16.1.1	172.16.1.253	172.16.2.253	172.16.1.1
Dst IP	172.16.3.1	172.16.2.253	172.16.3.253	172.16.3.1
Protocol	UDP	UDP	UDP	UDP
Src port	43000	1	1	43000
Dst port	8080	1	1	8080

First of all, in Figs. 4, 5 and 6 we show the flow table entries at each OpenFlow switch, determined by the desired forwarding policy. On the other hand, in Figs. 7, 8, 9, 10 and 11 we show traffic captures at $H11$, $R1$, $R2$, $R3$ and $H31$ respectively, where we can observe how packets are modified by the OpenFlow switches.

Priority	Cookie	Match Fields	Actions
200	0x2b00000000000038	ipv4_dst=172.16.3.1,hard_timeout=15000,ipv4_src=172.16.1.1,idle_timeout=2800,eth_type=0x0800,tp_dst=8080,ip_proto=17,tp_src=43000,	set_field:1->udp_dst,set_field:1->udp_src,set_field:172.16.2.253->ip_dst,set_field:172.16.1.253->ip_src,output:3

Fig. 4. Flow table entry at switch S1.

Priority	Cookie	Match Fields	Actions
200	0x8	ipv4_dst=172.16.2.253,hard_timeout=12000,ipv4_src=172.16.1.253,idle_timeout=2800,eth_type=0x0800,tp_dst=1,ip_proto=17,tp_src=1,	set_field:1->udp_dst,set_field:1->udp_src,set_field:172.16.3.253->ip_dst,set_field:172.16.2.253->ip_src,set_field:80:fa:5b:2d:52:3d->eth_src,set_field:00:0c:42:4a:b7:b5->eth_dst,ALL

Fig. 5. Flow table entry at switch S2.

Priority	Cookie	Match Fields	Actions
200	0x2b00000000000038	ipv4_dst=172.16.3.253,hard_timeout=15000,ipv4_src=172.16.2.253,idle_timeout=2800,eth_type=0x0800,tp_dst=1,ip_proto=17,tp_src=1,	set_field:8080->udp_dst,set_field:43000->udp_src,set_field:172.16.3.1->ip_dst,set_field:172.16.1.1->ip_src,output:19

Fig. 6. Flow table entry at switch S3.

In this test, the packet match fields identifying the flow are source-destination IP addresses and ports; then the flow rules may be interpreted as follows. If a packet matches the flow table entry of $S1$ (in other words source IP is 172.16.1.1, destination IP is 172.16.3.1, source port is 43000 and destination port is 8080), then IP addresses and ports will be changed according to the corresponding action. In this sense, we can observe Figs. 7 and 8. The first one shows the original traffic generated at $H11$ and the second one reflects the traffic forwarder by $R1$. Packets to $R1$ arrive with 172.16.1.253:1 and 172.16.2.253:1 as Source IP:port and Destination IP: port respectively. Due to the destination address, $R1$ forwards that traffic to $R2$.

```
18:56:41.986421 IP 172.16.1.1.43000 > 172.16.3.1.8080: UDP, length 1000
18:56:41.995113 IP 172.16.1.1.43000 > 172.16.3.1.8080: UDP, length 1000
18:56:42.003885 IP 172.16.1.1.43000 > 172.16.3.1.8080: UDP, length 1000
18:56:42.012619 IP 172.16.1.1.43000 > 172.16.3.1.8080: UDP, length 1000
18:56:42.021363 IP 172.16.1.1.43000 > 172.16.3.1.8080: UDP, length 1000
18:56:42.030110 IP 172.16.1.1.43000 > 172.16.3.1.8080: UDP, length 1000
```

Fig. 7. Traffic capture in H11. This traffic arrives to S1.

```
SRC-ADDRESS                    DST-ADDRESS
172.16.1.253:1 (tcpmux)        172.16.2.253:1 (tcpmux)
172.16.1.253:1 (tcpmux)        172.16.2.253:1 (tcpmux)
172.16.1.253:1 (tcpmux)        172.16.2.253:1 (tcpmux)
172.16.1.253:1 (tcpmux)        172.16.2.253:1 (tcpmux)
172.16.1.253:1 (tcpmux)        172.16.2.253:1 (tcpmux)
172.16.1.253:1 (tcpmux)        172.16.2.253:1 (tcpmux)
```

Fig. 8. Traffic capture in R1. Traffic modified by S1.

In Fig. 9 we show a traffic capture at $R2$ in the interface connected to $S2$ (traffic from $R2$ to $S2$ and vice-verse). Traffic from $R1$ (with the tuple 172.16.1.253:1 and 172.16.2.253:1 as

source and destination parameters) is forwarded to $S2$ and then this switch applies the action that is defined in its flow table (see Fig. 5): IP address are changed and the traffic is sending through the same interface that was received.

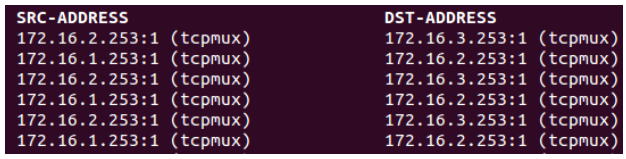

Fig. 9. Traffic capture in R2. Traffic from R2 to S2 and vice-verse.

Finally, in Fig. 10 and Fig. 11 we show the traffic outbound $R3$ and traffic that effectively arrives to $H31$. As expected, captures in $H11$ and $H31$ show the same traffic characteristics.

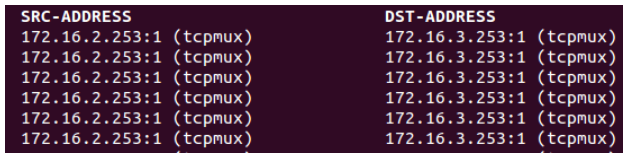

Fig. 10. Traffic capture in R3. Traffic from R3 to S3, before S3 applies the correspondent actions.

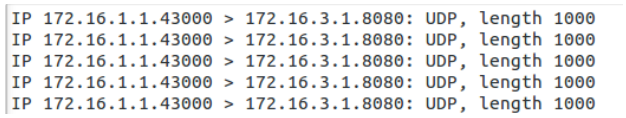

Fig. 11. Traffic capture in H31.

B. Flow Identification test

For the second proof we use the same topology described in Fig. 2 and we implement three different ONRPs:

- ONRP 1: UDP traffic from subnet 172.16.1.1/32 to subnet 172.16.3.1/32 with source port 43000 and destination port 8080 is routed through S1-S2-S3 path.
- ONRP 2: UDP traffic from subnet 172.16.1.0/24 to subnet 172.16.2.0/24 with any source port and destination port 900 is routed through S1-S2 path. This policy shares a "link" with ONRP 1.
- ONRP 3: UDP traffic from subnet 172.16.1.0/24 to subnet 172.16.3.0/24 with any source port and any destination port is routed through S1-S3 path.

To test this scenario we generate four flows according to Table V. Observe that flow 1 belongs to ONRP 1, flow 2 to ONRP 2 and flow 3 and flow 4 belong to ONRP 3. It should be noted that no port is specified in the last policy, so different flows can be generated that match ONRP 3.

As an example, following we explain $S1$ rules and its local identifiers (see summary in Table VI). Also in Fig. 12 we present a traffic capture in $R1$. For flow 1, src port = 0 and dst port = 1 are used as local identifiers in [$PoP1$,$PoP2$] path, on the other hand for flow 2 (as it belongs to another policy),

TABLE V
FLOW IDENTIFICATION: UDP FLOWS FROM H11.

	flow 1	flow 2	flow 3	flow 4
Src IP	172.16.1.1	172.16.1.1	172.16.1.1	172.16.1.1
Dst IP	172.16.3.1	172.16.2.1	172.16.3.1	172.16.3.1
Protocol	UDP	UDP	UDP	UDP
Src port	43000	100	1000	2000
Dst port	8080	900	8080	8080
Path	S1,S2,S3	S1,S2	S1,S3	S1,S3

src port = 1 and dst port = 1 are used. Note that both flows have to go through the link [$PoP1$,$PoP2$] and according to the local identifier (src port) it is possible to differentiate both flows. For flow 3, since it belongs to another policy and it has to travel on the link [$PoP1$,$PoP3$], src port = 2 and the dst port = 1 are used. So far all flows belong to different policies and have been successfully identified. In the case that they belong to the same policy, as is the case of flow 3 and flow 4, it is necessary to be able to differentiate them. To achieve this, the src port = 2 is configured for flow 4 indicating that they belong to the same policy but now the dst port = 2 is configured in order to differentiate both flows.

TABLE VI
SUMMARY OF FLOW LOCAL IDENTIFIERS IN SWITCH S1.

flow	Src. port	Dest. port	ONRP
1	0	1	1
2	1	1	2
3	2	1	3
4	2	2	3

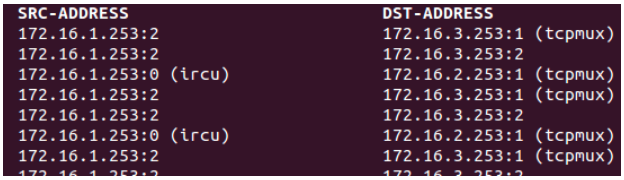

Fig. 12. Traffic modified by S1 captured at R1.

C. Practical issues and its solutions

During the validation and evaluation process, we have found some misbehaviour of the commercial OpenFlow switches (besides the Pica8, we have also tried to use two HP A5500-24G-4SFP HI, Software Version 5.20.99 but without success).

1) INPORT feature: Besides changing the source and destination IP address and source port for bounced packets, as explained in Section III, it is necessary to force the intermediate switches (for instance $S2$ in Fig. 3) to forward the packets through the same interface they arrived. To solve this issue, OpenFlow provides the "INPORT" and the "ALL" forwarding schemes [18], but the second option implies a flooding in all the interfaces thus representing an obvious performance penalty. In the simulated-environment, "INPORT' works completely successful. However, in the real one we need to use the "ALL" forwarding scheme.

2) Cookies unique identification: In section B.11.11 of the OpenFlow specification version 1.5 it is stated that: "Having the cookie in the packet-in enables the controller to more efficiently classify packet-in, rather than having to match the packet against the full flow table". The ONOS controller uses this idea by setting an application identifier in the 12 most significant bits of the cookie so that it can easily identify which application installed the flow entry that provoked a certain PACKET_IN.

Using the same concept, we configured the ONRP_id in the remaining 48 bits of the cookie in order to simplify the matching process at the controller level [5]. In particular, this meant the existence of multiple flow entries with the same exact cookie value. Although this might seam wrong, we were not able to find any theoretical mistake based on the OpenFlow specification. Moreover, the testing done with the Mininet emulator, which uses switches OpenVSwitch version 2.9.2, where completely successful. However, during the execution in the real-environment testing bench, we encountered that the HP A5500 Switches implemented a flow entry duplication filter based on the cookie value. This made our implementation completely useless, therefore in order to solve this problem, we reduced the ONRP_id size from 48 bits to 16 bits and we used the 32 remaining bits to uniquely differentiate every flow entry installed.

3) ODL and HP A5500: During the real network experiments we detected a communication problem between ODL and switches HP A5500. We believe that is strongly related with the switch firmware version. For this reason, the real overlay network only includes a Pica8 OpenFlow switch.

4) ARP for switches: As the OpenFlow switches need to have an IP address assigned in order to receive and process packets matching the ONRPs, it is necessary to solve the ARP requests and replies. For instance, when $R2$ receives a packet destined to $S2$ switch IP address, it will send an ARP request to obtain the required MAC address. In the case of the complete application developed for the ONOS controller, we implement a module which implements the required ARP message handling [5]. For the tests performed with the ODL controller, we use additional laptops configured with the same IP address associated to the switches just to provide the needed responses to ARP requests. Subsequent messages sent by the routers to the MAC address of those additional laptops are handled by the OpenFlow switch, avoiding forwarding the traffic to the laptop and enabling to bounce the packets back to the router.

VI. Conclusions

We demonstrate that the proposed forwarding strategy for flow-based QoS forwarding can be properly implemented over a SDN architecture considering the benefits of a software implementation at a centralized point of the network.

The main challenges we faced were related to behaviour of the commercial switches available to implement the testbed as reported in Section V-C, probably caused by outdated hardware or firmware.

We hope to be able to do further testing with other commercial OpenFlow switches and it will be also useful to analyze the implementation of a complete application to run over ODL controller.

Acknowledgment

This work was partially supported by ANII-FMV project "Routing and metrology in Overlay Networks using the Software Defined Network paradigm".

The authors would like to thank Diego Mazzuco for his time to solve different issues of ONOS development.

References

[1] B. D. Vleeschauwer, F. D. Turck, B. Dhoedt, P. Demeester, M. Wijnants, and W. Lamotte, "End-to-end QoE Optimization Through Overlay Network Deployment," in *Information Networking, 2008. ICOIN 2008.*, Jan 2008.

[2] D. G. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient Overlay Networks," in *18th ACM SOSP*, 2001.

[3] H. Zhang, L. Tang, and J. Li, "Impact of Overlay Routing on End-to-End Delay," in *Computer Communications and Networks, 2006. ICCCN 2006.*, 2006.

[4] P. Belzarena, G. G. Sena, I. Amigo, and S. Vaton, "Sdn-based overlay networks for qos-aware routing," in *Proceedings of the 2016 Workshop on Fostering Latin-American Research in Data Communication Networks*, ser. LANCOMM '16. New York, NY, USA: ACM, 2016, pp. 19–21. [Online]. Available: http://doi.acm.org/10.1145/2940116.2940121

[5] I. Brugnoli, M. Fernández, and D. Mazzuco, "Overlay network routing application (onrapp) (undergraduate tesis) universidad de la república (uruguay). facultad de ingeniería. iie," jun 2019. [Online]. Available: https://iie.fing.edu.uy/publicaciones/2019/BFM19

[6] I. Amigo, G. G. Sena, M. Chami, and P. Belzarena, "An sdn-based approach for qos and reliability in overlay networks," in *Network Traffic Measurement and Analysis Conference, TMA 2018, Vienna, Austria, June 26-29, 2018*, 2018, pp. 1–2. [Online]. Available: https://doi.org/10.23919/TMA.2018.8506581

[7] O. Brun, L. Wang, and E. Gelenbe, "Big Data for Autonomic Intercontinental Overlays," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. pp.575 – 583, 2016. [Online]. Available: https://hal.laas.fr/hal-01461990

[8] T. U. PENGUIN, "rp_filter and lpic-3 linux security." [Online]. Available: https://www.theurbanpenguin.com/rp_filter-and-lpic-3-linux-security/

[9] ONF, "Onos: github repository." [Online]. Available: https://github.com/opennetworkinglab/onos

[10] OpenDaylight, "Documentation release beryllium." [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/opendaylight/stable-beryllium/opendaylight.pdf

[11] F. Pakzad, "Comparison of software defined networking (sdn) controllers. part 7: Comparison and product rating." [Online]. Available: https://aptira.com/comparison-of-software-defined-networking-sdn-controllers-part-7-comparison-and-product-rating/

[12] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "Sdn controllers: A comparative study," in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, April 2016, pp. 1–6.

[13] M. Darianian, C. Williamson, and I. Haque, "Experimental evaluation of two openflow controllers," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct 2017, pp. 1–6.

[14] I. Brugnoli, M. Fernández, and D. Mazzuco, "Onrapp: Overlay network routing application (git repository)," 2019. [Online]. Available: https://gitlab.fing.edu.uy/tesis/onra/

[15] Mininet, "Mininet." [Online]. Available: http://mininet.org/download/

[16] OpenDaylight, "Odl openflowplugin release master." [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/odl-openflowplugin/stable-oxygen/odl-openflowplugin.pdf

[17] Pica8, "Picos support for openflow 1.3." [Online]. Available: https://docs.pica8.com/display/PicOS21116cg/

[18] ONF, "Openflow switch specification version 1.3.0 (wire protocol 0x04)." [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf