Host Discovery Solution: An Enhancement of Topology Discovery in OpenFlow based SDN Networks

Pilar Manzanares-Lopez[1], Juan Pedro Muñoz-Gea[1], Francisco Manuel Delicado-Martinez[2], Josemaria Malgosa-Sanahuja[1] and Adrian Flores de la Cruz[1]

[1]*Department of Information Technologies and Communications, Universidad Politecnica de Cartagena, Antiguo Cuartel de Antigones, Cartagena, Spain*

[2]*Department of Computing System, Universidad de Castilla-La Mancha, Albacete, Spain*

Keywords: Host Discovery, Monitoring, Software Defined Networking.

Abstract: Software Defined Networking (SDN) is an emerging paradigm based on the separation between the control plane and the data plane. The knowledge of the network topology by the controller is essential to allow the implementation of efficient solutions of network management and network resource utilization. Most of the OpenFlow SDN controllers include a mechanism to discover the network nodes (router and switches) and the links between them. However, they do not consider other important elements of the networks: the hosts. In this paper we propose a host discovery mechanism to improve the topology discovery solutions in SDN networks. The proposed mechanism, that has been coded as a software module in Ryu SDN controller, allows the detection and tracking of hosts even when they don't generate traffic. The implemented software module has been tested in emulated SDN networks and in real scenarios using ONetSwitch, a real programmable SDN platform.

1 INTRODUCTION

Software Defined Networking (SDN) is a new networking paradigm which is based on the separation between the control plane and the data plane. In traditional IP networking, the network elements (routers and switches) implement routing protocols which are used to determine network paths and consequently make routing decisions. Accordingly to these decisions, the data packets will be forwarded. In contrast, in SDN paradigm, the network elements only perform packet forwarding (it is called data plane). The other tasks, all of them included in the called control plane, are located in an entity called the SDN controller. The control plane is responsible for making decisions on how packets should be forwarded by one or more network devices and pushing such decisions down to the network devices for execution (Haleplidis et al., 2016).

In order to be able to make these decisions, it is essential that the control plane has updated information about the network topology. That is the reason why network discovery is a key aspect of SDN. The knowledge of the network topology is necessary to be able to control and modify the data paths. Network monitoring and traffic engineering are also important tasks in network management that require an accurate knowledge of the network topology (Akyildiz et al., 2014).

Most of the SDN controllers, e.g. POX (POX, 2016), Ryu (Ryu, 2016), OpenDayLight (ODL, 2016) Floodlight (Floodlight, 2016), implement network topology discovery mechanisms that are limited to the discovery of network elements (switches and routers) and links between them. However, in our opinion, an important component of the network is not considered: the hosts. Identify the existence of hosts, even before they generate traffic, can offer to the controller a very useful information to be used in the network management tasks.

In this paper we propose a host discovery mechanism designed for SDN networks. This functionality complements the non-standardized topology discovery mechanism, based on the use of LLDP (Link Layer Discovery Protocol) packets (IEEE, 2009), included in most of the OpenFlow-based SDN controllers.

The proposed mechanism has been coded and included in the Ryu SDN controller and it has been tested in two scenarios. On the one hand, the so-

Manzanares-Lopez, P., Muñoz-Gea, J., Delicado-Martinez, F., Malgosa-Sanahuja, J. and Cruz, A.
Host Discovery Solution: An Enhancement of Topology Discovery in OpenFlow based SDN Networks.
DOI: 10.5220/0005967000800088
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 1: DCNET*, pages 80-88
ISBN: 978-989-758-196-0

lution has been tested in an emulated OpenFlow-SDN network using Mininet (Mininet, 2016), the famous emulator in SDN. On the other hand, it has been tested using ONetSwitches (Hu et al., 2014), an all programmable SDN platform. The main chip of ONetSwitch is the Xilinx Zynq-7045 SoC, which integrates ARM Cortex-A9 dual core processor-based processing system (PS) and Kintex-7 FPGA-based programmable logic (PL) into a single chip. The PS part makes ONetSwitch software programmable and the PL part enables the reconstruction of the hardware logic. The use of the ONetSwitches allowed us to evaluate the implemented host discovery module in a wider set of experiments than using just the Mininet emulator. Moreover, it led us to do a detailed analysis of a part of the OpenFlow standard: the meaning of the port status fields, which allow the controller to track the status of each switch port.

The rest of the paper is structured as follows. Section 2 describes the basic concepts of OpenFlow standard. Section 3 describes the non standardized topology discovery solution implemented in most of the OpenFlow-based controllers. Next, section 4 proposes a mechanism to implement the host discovery functionality, improving the basic topology discovery of OpenFlow-based controllers. Section 5 describes the usefulness of the host discovery utility. Finally, section 6 concludes the paper.

2 SDN OPENFLOW BACKGROUND

The separation between the infrastructure (forwarding plane) and the SDN controller (control plane) is done via standardized southbound interfaces. OpenFlow, that is promoted by Open Network Foundations, is the dominant standard providing the SDN southbound interface between the SDN controller and the SDN switches. In this paper we consider the OpenFlow version 1.3 (ONF, 2012), the latest version of OpenFlow that has support from switch vendors.

After the start-up, the switches and the controller perform an initial handshake to establish the OpenFlow connection. Each switch contacts its controller on the corresponding IP address and TCP port number and establishes a TCP, which could be encripted using TLS protocol. The switch and the controller send a Hello message with the higher OpenFlow version supported. Then, if the switch has support for the OpenFlow version sent by the controller, the controller sends a OFPT_FEATURES_REQUEST message to the switch. The switch responds with an OFPT_FEATURES_REPLY message that includes a unique switch's identifier called datapath_id, its active ports and the corresponding MAC addresses. After that, the OpenFlow connection is established.

This initial handshake informs the controller about the existence of switches, but not about the interconnection of the network elements. To obtain a picture of the network topology, a topology discovery solution must be implemented. Most of the well-known SDN controllers implement a similar, but non-standardized, topology discovery mechanism that allows the discovery of the links interconnecting switches. This mechanism is described in the following section.

As part of the control plane, the controller could use the network topology information to solve different network management and control tasks. Among them, a fundamental task is to find out the most appropriate paths and configure the switches consequently to do the forwarding tasks.

An OpenFlow switch consists of one or more flow tables and a group table. Using OFPT_FLOW_MOD messages, the controller can add, update and delete flow entries in flow tables, reactively (in response to packets) or proactively. Each flow entry consists of a match structure (many fields to match flows such as input switch port, Ethernet source/destination address, VLAN ID, VLAN priority, IP source/destination address,...), a set of counters and a set of actions. On packet arrival, the packet is matched with the match fields in a table and, if any entry matches, the counters in that entry are updated and the associated actions are performed. If no entry matches, then a table-miss event has occurred. In OpenFlow 1.2 and earlier, every flow table has a default table-miss flow entry. However, in OpenFlow 1.3, the default table-miss flow entry must be inserted manually by the controller. The actions associated to a table-miss flow entry may include dropping the packet, sending the packet to another table, or sending the packet to the controller. In the last case, a OFPT_PACKET_IN message is sent from switch to controller, which carries the packet. When controller receives the OFPT_PACKET_IN message, it evaluates its contents and decides what actions will be done over the packet included in the OFPT_PACKET_IN message. Then, the packet is returned to the switch in a OFPT_PACKET_OUT, which carries the packet and a list of actions to be applied over the packet. Also, the controller could send an OFPT_FLOW_MOD message to include a flow entry in switch flow table, in order to process similar packets in future.

3 TOPOLOGY DISCOVERY IN OPENFLOW-BASED NETWORKS

As described before, a SDN controller knows the existence of network elements thanks to the initial handshake process. In figure 1, the initial handshake between switch s2 and the controller in represented by steps 1 and 2. For simplicity, the handshake corresponding to the rest of switches is not shown.

However, the mechanism to discover how the switches are interconnected is not standarized in OpenFlow-based networks. Nevertheless, most of the OpenFlow-based controllers implement a topology discovery based on LLDP (Link Layer Discovery Protocol).

LLDP (IEEE, 2009) is vendor-neutral link layer protocol in the Internet Protocol Suite used by network devices for advertising their identity, capabilities, and neighbors on IEEE 802 local area networks, principally wired Ethernet. LLDP packets are sent by devices from each of their ports at a fixed interval, encapsulated in Ethernet frames with *ethertype* field set to $0x88cc$. The LLDP frames are sent to the bridge-filtered multicast MAC address $01:80:C2:00:00:0E$. Therefore, the LLDP protocol is a one-hop protocol where the LLDP packets are only received by directly connected network devices.

OpenFlow switches do not initiate the sending of LLDP packets by themselves. To initiate the topology discovery mechanism, the controller sends an OFPT_PACKET_OUT message for each port of each switch. Each OFPT_PACKET_OUT contains as payload a LLDP frame and contains an instruction to send this frame for the corresponding port (step 4 in figure 1). The forwarding of the LLDP frame allows the neighbors to know about themselves, but doesn't allow the controller to obtain this information.

In order to let the controller know the relation between switches, they must have a table flow entry which orders to send to the controller, by an OFPT_PACKET_IN message, any LLDP frame received from any port except the Controller port. This rule is sent by a OFPT_FLOW_MOD message after the connection phase (step 3 in the figure), before the sending of the first LLDP packet.

Finally, after the reception of the OFPT_PACKET_OUT messages, switches forward the encapsulated LLDP frames for each port except the incoming (step 5 in the figure). When switches receive the LLDP packets, they send them to the controller using OPFT_PACKET_IN messages (step 6 in the figure).

The topology discovery mechanism includes the discovery of switches and links between them but it does not include the discovery of hosts. Some SDN controllers such as Ryu and OpenDayLight implement a basic host discovery solution. This basic solution is based on the fact that the table-miss flow entry of switches forces the sending of the packet to the controller. Thus, when a switch receives ARP or IP traffic from a host, since there are no installed flow rules for the incoming flow, the switch forwards the first received packet of the flow to the controller. Based on this packet, the controller discovers the host identity.

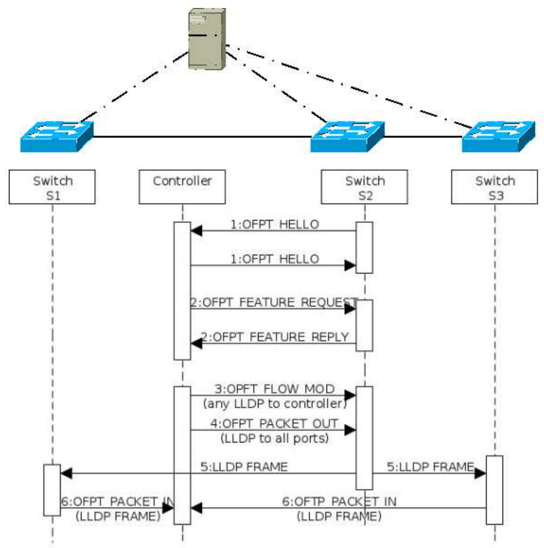
Switch S1
Controller
Switch S2
Switch S3
1:OFPT HELLO
1:OFPT HELLO
2:OFPT FEATURE REQUEST
2:OFPT FEATURE REPLY
3:OFPT FLOW MOD (any LLDP to controller)
4:OFPT PACKET OUT (LLDP to all ports)
5:LLDP FRAME
5:LLDP FRAME
6:OFPT PACKET IN (LLDP FRAME)
6:OFTP PACKET IN (LLDP FRAME)

Figure 1: Topology discovery process in OpenFlow-based SDN networks.

Ryu (Ryu, 2016) is a component-based software defined networking framework that provides software components to create network management and control applications. It supports OpenFlow and is programmed in Python. In particular, the topology discovery functionality is implemented in a module called */ryu/topology/switches.py*.

Another mechanism to discover hosts could be the use of the LLDP protocol in hosts. However, this solution requires that a LLDP daemon has to be running in each host. And depending of the case, this could be very difficult to assert, i.e. in topologies where hosts and network devices are not administrated by the same entity.

In the next section we present a host discovery module that enhances the typical topology discovery. Although the proposed mechanism has been implemented in Ryu, it could be adapted to other SDN controllers.

4 PROPOSED HOST DISCOVERY IN OPENFLOW-BASED NETWORKS

4.1 Technical Aspects to be Considered

Our proposed mechanism to implement initial host discovery, that is, without having to wait for the host to generate traffic, uses three kinds of information that the SDN controller knows: the knowledge of a switch ports, the changes in the status of the ports, and the knowledge of links between switches. Next, we are going to discuss each one and how it is useful during the host discovery procedure.

After the session establishment, the controller sends a port description request message to each switch querying a description of all the OpenFlow ports. The OpenFlow message type is OFPT_MULTIPART_REQUEST, concretely a OFPMP_PORT_DESC. Each switch responds with a port description reply message. In this case, the message type is a OFPT_MULTIPART_REPLY, concretely a OFPMP_PORT_DESC.

The port description reply enables the controller to get a description of all the OpenFlow ports of that switch. Among other fields, the description of a port includes a unique port number that identifies the port on the switch, the MAC address for the port and two fields called "*config*" and "*state*" which are composed of several flags. The structure of both fields is described below:

```
 enum ofp_port_config {
  OFPPC_PORT_DOWN=1<<0;
/*Port is administratively down*/
  OFPPC_NO_RECV=1<<2;
/*Drop all the pkts received  by port*/
  OFPPC_NO_FWD=1<<5;
/*Drop packets forwarded by port*/
  OFPPC_NO_PACKET_IN=1<<6;
/*Do not send packet-in msgs for port*/
};

enum ofp_port_state {
  OFPPS_LINK_DOWN=1<<0;
/*No physical link present*/
  OFPPS_BLOCKED=1<<1;
/*Port is blocked*/
  OFPPS_LIVE=1<<2;
/*Live for Fast Failover Group*/
};
```

The "*config*" value is set by the controller and it is not changed by the switch. It is composed of four flags. According to the OpenFlow version 1.3 specification, the OFPPC_PORT_DOWN flag indicates that the port has been administratively brought down and should not be used by OpenFlow.

The "*state*" value indicates the state of the physical link. It is composed of three flags. According to the OpenFlow version 1.3 specification, the OFPPS_LINK_DOWN flag indicates that the physical link is not present. The port state bits are read-only and cannot be changed by the controller. When any state flags change, the switch sends a OFPT_PORT_STATUS message to notify the controller of the change.

Thanks to the LLDP-based topology discovery mechanism implemented in most of the controllers, the controller can distinguish between edge ports and non-edge ports of the switches. The non-edge ports are the ports which are used within the links between switches. Consequently, they send and receive LLDP messages. The edge ports are the ports which are not used within the links between switches. They send LLDP messages but do not receive anyone. Thus, the controller can identify the ports that potentially could be connected to hosts.

The definition of the "*link down*" flag seems clear: no physical link present. However, after analyzing some experiments we made with the ONetSwitch platform, the obtained results introduced some doubts about it. According to the OpenFlow specification, the "*link down*" flag in the port description reply should be 0 if there is a connected link and 1 if not. Nevertheless, from the experiments, we could conclude that the "*state*" value of a switch port is always 4, regardless whether there is a physical link connected or not when the port description request is received. That is, the "*live*" flag is set to 1 and the "*link down*" flag is set to 0. Even more, if a switch port is free, and then a host that is on is connected to it, the "*link down*" flag does not change and any OFPT_PORT_STATUS message is generated.

From our observations we conclude that the "*link down*" flag does not inform if a link is present or not. Actually, the "*link down*" flag is used to indicate that a change has happened, in particular, that the link connected to that port is not available anymore.

For all that, in order to implement the host discovery module, it is not enough knowing the edge ports and check the initial "*state*" value indicated by the port description reply messages. It is necessary to implement a mechanism to certainly identify which ports are connected to hosts. As it will be detailed later, this mechanism is inspired by the host discovery option of the open source utility *nmap* (Lyon, 2008).

On the other hand, the proposed advanced host discovery mechanism also improves the basic host discovery solution by offering reaction to the dy-

Table 1: Pseudocode of the proactive_host_discovery function.

```
01:  proactive_host_discovery():
02:      for switch in network.switches:
03:       for port in switch.ports:
04:        if edge_port(port):
05:         send_ping_arp_flowmod(switch,port)
06:         send_ping_arp(switch,port,netaddress)

07:  send_ping_arp_flowmod(switch,port):
08:      match = (ether_types==ARP & eth_dst==port.hw_addr)
09:      action = output_to_controller
10:      msg=create_flow_mod_message(match,action)
11:      send(msg, switch)

12:  send_ping_arp(switch,port,netaddress):
13:    for ip in range netaddress:
14:     dst=FF:FF:FF:FF:FF:FF
15:     src=port.hw_addr
16:     dst_ip=ip
17:     src_ip=broadcast_netaddress
18:     ethfr=ether_frame(ARP_request,dst,src,dst_ip,src_ip)
19:     send_packet_out(ethfr,switch,port)
```

namism of the network. Hosts can leave the network at any moment and links can fail. To identify and respond to the network changes, the proposed module will listen and process the OFPT_PORT_STATUS messages that are generated by the switches when a port is added, modified or removed.

4.2 Proactive Search of Hosts

The proposed host discovery module must be able to locate hosts which are initially connected to the network, and also new hosts which join the network during the period of time that the network is being monitored.

As described before, using only the control messages exchanged between the switches and the controller, it is not possible to differentiate non-connected ports from ports connected to hosts. In order to solve this first task, the proposed mechanism implements a solution inspired by the host detection option of the open source *nmap* utility.

After the session establishment, and once the port description messages have been exchanged, the controller starts a proactive search of hosts. The pseudocode is shown in table 1.

The controller sends a set of OFPT_PACKET_OUT messages for each edge port to each edge switch (line 6). Each OFPT_PACKET_OUT message contains an ARP request asking for a particular IP address belonging to the IP network address range. Although the ARP request is created by the controller, the source hardware address is set to the hardware address of the edge port. Thus, when an ARP reply is generated to respond to a proactively generated ARP request, the switch will be able to differentiate the packet from ‘normal’ ARP replies (which are generated due to traffic between hosts) and send it to the controller.

After receiving each of the OFPT_PACKET_OUT messages destined to a particular edge port, the switch sends the ARP requests out of the specified port. If the edge port is a non-connected port, no ARP replies will be received. However, if a host is connected to the edge port, an ARP reply will be sent to the switch.

However, in order to perform the host discovery task, the ARP reply must be received by the controller. For that reason, as it can be observed in the pseudocode (line 5), before sending the OFPT_PACKET_OUT messages to a switch, the controller sends a OFPT_FLOW_MOD message for each edge port to add a new flow entry in the OpenFlow table. The match conditions for each new flow entry are: the input packet must be an ARP packet whose hardware destination address is the edge port hardware address. If the input packet matches, the packet will be sent to the controller. Then, the controller will process the ARP traffic as usual (defined in */topology/switches.py*), and a host will be discovered and added to the list of hosts.

Based on both topology discovery and host discovery modules, the controller will know the complete network topology, that is, the switches, the links between them, the IP and MAC addresses of the hosts and the switch ports they are connected to.

4.3 Tracking of Host Connections

Dynamism of networks must be taken into account when implementing a topology discovery solution: ports can be added or deleted from switches, new links can be established or existing links can fail. Thus, the proposed host discovery mechanism must also react to network changes that affect host connections.

As said before in section 4.1, if a change happens on a port, the switch notifies the controller by sending a OFPT_PORT_STATUS message. The reasons to generate this type of message are: a port has been added (OFPPR_ADD), a port has been deleted (OFPPR_DELETE) or the state of a port has changed (OFPPR_MODIFY).

The first two reasons only affect to the number of ports that must be taken into account: a port has been added or a port has been deleted. However, the last event is more complex.

If a link is suddenly disconnected, the switch detects that a physical link is not present anymore, and then a OFPT_PORT_STATUS message is sent to the controller indicating the change in the port state from 4 (the "*live*" flag set to 1) to 5 (both "*live*" flag and "*link down*" flags are set to 1). This event is associated to the physical disconnection of a link.

In a first experiment, a ONetSwitch initially has 4 non-connected ports. By means of an OF-PMP_PORT_DESC message, the switch informs the controller of the port state values: 4 (only the "live" flag set to 1). Then, a host (that is on) is connected to the switch. In this situation, no OpenFlow message is generated. A new physical link has been created between the host and the switch, but no notification of port state change is generated. As it was tested before, the standard just defines a "*link down*" flag to indicate that the existing link connected to a port is not available anymore.

Consequently, if the host (that is on) is physically disconnected, the switch generates a OF-PPR_MODIFY message indicating that the port state value has changed to 5 (both "live"and "link down" flags are set to 1). Finally, if the host is connected again, another OFPPR_MODIFY message with port state value 4 is sent to the controller. The conclusion of this first experiment is that the "*link down*" flag is certainly related to the fall (and later recovery) of physical links, but it is not related to the establishment of a new link.

In a second experiment the same switch has three non-connected ports and the last one is connected to a host that is off. As in the previous experiment, an OF-PMP_PORT_DESC message is sent to the controller to inform of the port state values: 4 (only the "live" flag set to 1). Continuing with the experiment, the host is turned on. Although in this case the physical link already existed, a series of OFPPR_MODIFY messages are sent to the controller.

Although there is a number of mechanisms which can have impact on the link state, after analyzing the OpenFlow traffic capture and the host and switch logs, we could conclude that the OFPPR_MODIFY messages were generated due to Ethernet auto-negotiation phase. The initial "*port state*" value that the switch sent to the controller was 4. Then, when the host is turned on, the switch indicates to the controller that the link changes to down (a OFPPR_MODIFY message is sent with "*port state*" value 5) until the 100 Mbps negotiation phase finishes. Then, a new OFPPR_MODIFY message with "*port state*" value 4 is sent. Next, again, the switch indicates to the controller that the link changes to down (a OF-PPR_MODIFY message is sent with "*port state*" value 5) until the 1000 Mbps negotiation phase finishes. Then, a new OFPPR_MODIFY message with "*port state*" value 4 is sent. Finally, when the host is turned off again, two OFPPR_MODIFY messages are sent to the controller: the first one indicating a "*port state*" value of 5 and the second one a "*port state*" value of 4.

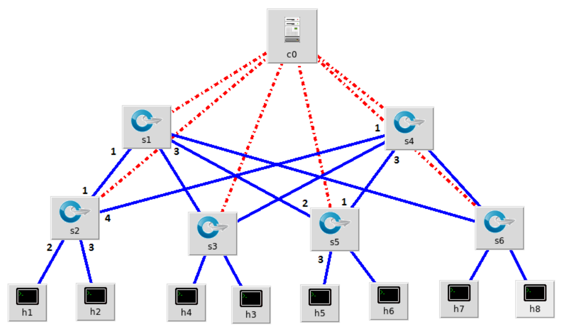

Figure 2: Fat-tree topology.

Taking into account the conclusions of the experiments, the host discovery module must react to the reception of OFPPR_MODIFY messages. When the received "*port state*" value is 5, it must be checked if there were a previously detected host connected to that port. If so, the host will be eliminated from the list of hosts. Otherwise, if the received "*port state*" value is 4, it must be checked if it is an edge port. If so, a proactive search of hosts on that port will be initiated, as described in the previous section.

However, due to the fact that the joining of host that is on does not generate a port status change and consequently it does not generate any message to be sent to the controller, it is necessary to periodically repeat the proactive mechanism on all the edge ports.

5 USEFULNESS AND EVALUATION OF HOST DISCOVERY

Using the mechanism based on the exchange of LLDP messages, SDN controllers can discover switches and connections between switches. This topology information can be used to determine the best path (shortest path, in terms of number of hops) from a source switch port to a destination switch port. However, existing SDN controllers only perform a basic host discovery.

The discovery of hosts can be useful in a SDN network topology discovery tool. For example, in a data center, it is tedious and error-prone to manually maintain the locations of virtual machines due to their frequent migration (Hong et al., 2015). The host discovery module together with an adequate monitoring tool provide an easy way to guarantee flexible network dynamics, optimizing the use of the network resources.

On the other hand, the host discovery before they generate traffic can also be useful in redundant networks to solve the problem of ARP flooding. With

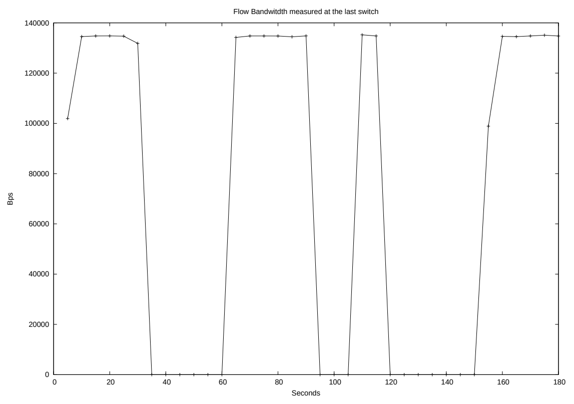

Figure 3: Flow bandwidth measurement obtained on the last switch s5.

the proposed host discovery mechanism, ARP Request messages are only sent to edge switches, and the ARP Responses are directly sent to the controller who will be able to maintain a database of the MAC and IP addresses of the activate hosts.

Figure 2 shows a fat-tree network, an example of common data center interconnection topology. We have used Mininet to emulate this topology. In this scenario, we are going to show an example of the utility of the tracking of host in the network management.

In the testbed, an UDP flow is created between host h2 (10.0.0.2) and host h5 (10.0.0.5) using the iperf tool. When switch s2 receives the first UDP packet, due to the fact that there is no any flow entry in the OpenFlow table that matches, a miss-table event occurs and, consequently, the UDP packet is sent to the controller using an OFPT_PACKET_IN message. The controller, who knows the location of h5 thanks to the proposed host discovery module, calculates the optimum path between switch s2 and switch s5, and installs an adequate flow entry on each intermediate switch. If at a given moment h2 fails or migrates to another point of the network, it is useful for the controller to detect this change immediately in order to act accordingly.

In this test, the UDP connection between host h2 and host h5 lasts 180 seconds. During this period, the host h5 leaves the network and joins again later three times. In the second 30, h5 leaves the network for 30 seconds; in the second 90, h5 leaves the network for 15 seconds and; in the second 120, h5 leaves the network for 30 seconds. Thanks to the proposed host discovery module, that includes host tracking, each time that the host h5 leaves the network, the controller realizes and intermediately deletes the flow entries corresponding to the route between h2 and h5 on all involved switches.

Figure 3 represents the transmission rate corresponding to the flow entry on switch s5 associated to the UDP flow between 10.0.0.2 (h2) and 10.0.0.5 (h5). The UDP flow has been monitorized using a monitoring tool developed by the authors of this work (Muñoz-Gea et al., 2016). It has been monitored on the last switch on the flow path since this is what is seen by the receiver. As can be observed, there is no flow entries corresponding to the UDP connection in the flow table of switch s5 during the interval in which the host h5 is disconnected. Similar results are obtained when the rest of switches are monitored.

OpenFlow also provides the possibility of getting detailed statistics on specific ports on selected switches. Making use of this functionality, results showed in figure 4 have been obtained. The first graph shows the transmission rate of the ports on switch s2 (that is, the first switch on the path between h2 and h5).

The second graph corresponds to switch s5, the last switch on the path. In this figure it can be observed the reception rate of the ports on switch s5, and even more descriptive, the transmission rate of the port to which h5 is connected. As can be seen, the transmission rate coincides with the expected result. The last two graphs represent the transmission and reception rates of the ports on switches s1 and s4.

It is interesting to point out that, as can be seen in the these graphs, at the beginning of the communication, the controller decides that the optimum path between h2 and h5 is s2-s4-s5. However, after the first time that h5 leaves the network and later joins again, the controller decides to select a new route between h2 and h5: s2-s1-s5. After the third and forth re-joining to the network, the controller chooses the route s2-s4-s5 again.

6 CONCLUSIONS

In this paper we have proposed a solution to implement a dynamic host discovery and tracking in SDN OpenFlow networks. It has been implemented as a new module of the well-known Ryu controller.

The module complements the non-standardized topology discovery mechanism implemented in most of the SDN OpenFlow controllers which, based on the exchange of LLDP packets, are able to identify switches and links between them. In order to define the host discovery mechanism, we had to analyze a particular aspect of the OpenFlow 1.3 standard: the "port state" flags and their relation with the generation of OFPT_PORT_STATUS messages. Both aspects are not widely described in the standard neither the bibliography.

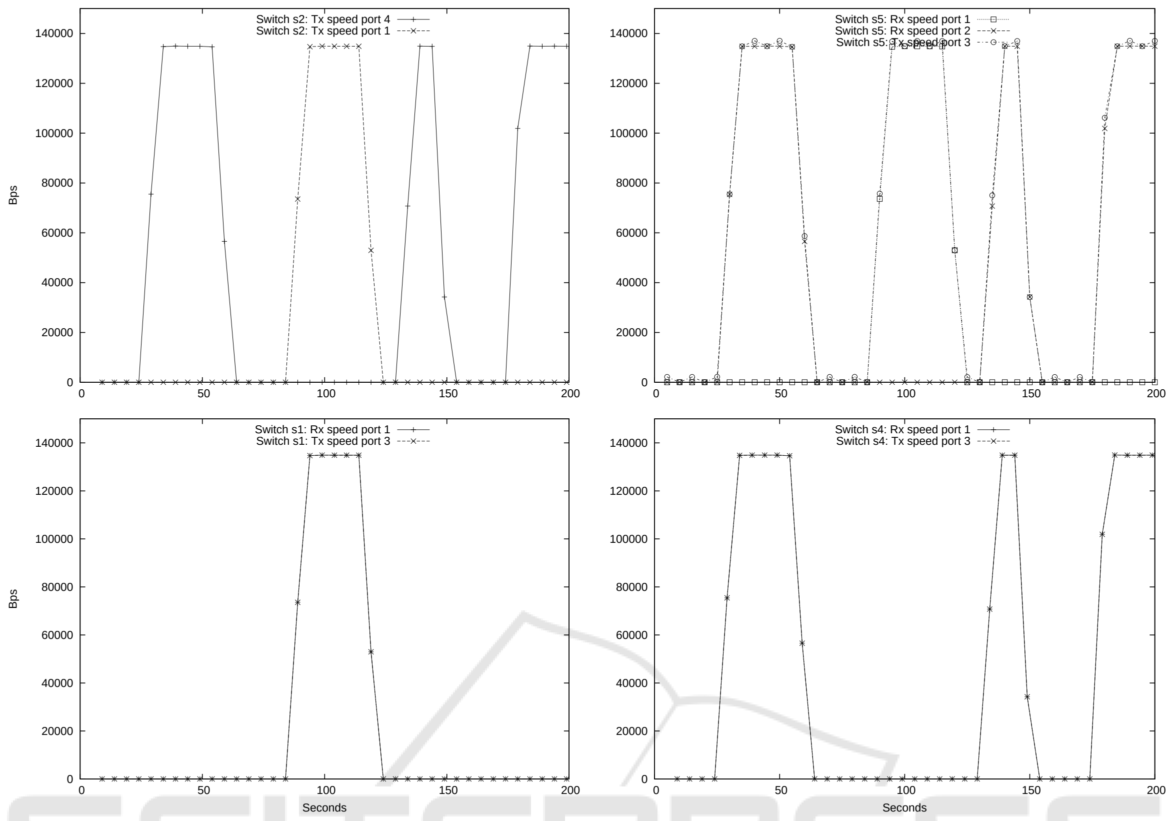

Figure 4: Transmission and reception rate of the ports on switches s2, s5, s1 and s4.

The proposed host discovery solution has been tested in emulated OpenFlow-SDN networks using Mininet, and also, in real scenarios using ONetSwitches, an all programmable SDN platform. Some of the benefits of the host discovery functionality have been described, and a use case has been described and analyzed considering a data center interconnection topology.

ACKNOWLEDGEMENTS

This work was supported by the MINECO/FEDER Project Grant TEC2013-47016-C2-2-R (COINS) and the MINECO/FEDER Project Grant TIN2015-66972-C5-2-R.

REFERENCES

Akyildiz, I., Lee, A., Wang, P., Luo, M., and Chou, W. (2014). A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks (71)*.

Floodlight (2016). Project floodlight: Open source software for building software-defined networks. http://www.projectfloodlight.org/floodlight.

Haleplidis, E., Denazis, S., Pentikousis, K., Salim, J., Meyer, D., and Koufopavlou, O. (2016). Sdn layers and architectures terminology. In *RFC 7426*.

Hong, S., Xu, L., Wang, H., and Gu, G. (2015). Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*.

Hu, C., Yang, J., Zhao, H., and J.Lu (2014). Design of all programmable innovation platform for software defined networking. In *Open Networking Summit*.

IEEE (2009). Ieee standard for local and metropolitan area networks - station and media access control connectivity discovery. In *IEEE Std 802.1AB*.

Lyon, G. F. (2008). *Nmap Network Scanning*. Insecure.com LLC.

Mininet (2016). Mininet. an instant virtual network on your laptop (or other pc). http://www.mininet.org.

Muñoz-Gea, J., Manzanares-Lopez, P., Malgosa-Sanahuja, J., and de la Cruz, A. F. (2016). Network failures support for traffic monitoring mechanisms in software-defined networks. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium, NOMS'16*.

ODL (2016). Opendaylight. http://www.opendaylight.org.

ONF (2012). Openflow switch specification. version 1.3.0. https://www.opennetworking.org/images/stories/

downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

POX (2016). Pox controller. http://github.com/noxrepo/pox.

Ryu (2016). Build sdn agilely. component-based software defined networking framework. https://osrg.github.io/ryu/.