B649 Project: Flow Management System using RYU controller

Siddharth Pathak

sidpath@iu.edu

May 4, 2018

1 Abstract:

The use of Software Defined Networking (SDN) architecture has seen considerable rise in recent times. Main motivation behind using SDN is to manage the ever increasing complex networks, and becoming as non vendor dependent as possible. SDN architecture decouples the control and data plane, through a protocol. In this project, I explore OpenFlow – a prominent protocol which follows the SDN architecture principles – and RYU controller, which provides Application Programmable Interface (API) to manage OpenFlow protocol supporting devices. I have implemented a Flow Management System using SDN and RYU controller on a Dell Z9100 OpenFlow switch. I analyze how scalable the OpenFlow switch is, by adding several different flows. To generate flow traffic, I programmed a flow generator using Scapy.

2 Introduction:

Explosion of devices connected to the Internet has made networks complex, and difficult to manage. This rise in number of devices has made network management an important task. For example, in an IT organization, if the organization wants to specially treat guest users connected to their network (limiting bandwidth to non-employees), then it has manually configure the network devices. These network devices can belong to different vendors, and have vendor specific configurations. This results in the making the network static in nature, as making constant changes in the above manner is quite a hassle.

The problems of the traditional network architecture can be solved by the use of SDN. With the use of SDN, existing networks can be made dynamic and easily configurable through a centralized access point. SDN enables the dynamic nature by the adding flows on the network devices in real-time. Because one can add flows in real-time, the configuration which would take days in the existing networks, SDN will make it possible in hours. Also, addition and removal of rules is software based, and they are not embedded into the devices.

The core idea behind SDN is the decoupling of Control Plane and Data Plane. SDN makes the data plane programmable, giving access to the programmer to manage the data plane. The other major contribution of SDN is that it aggregates all the data planes into a single entity. Programming one control plane, will have the control plane manage all the data planes in the network. Thus SDN allows the programmer to control several data planes using one control plane.

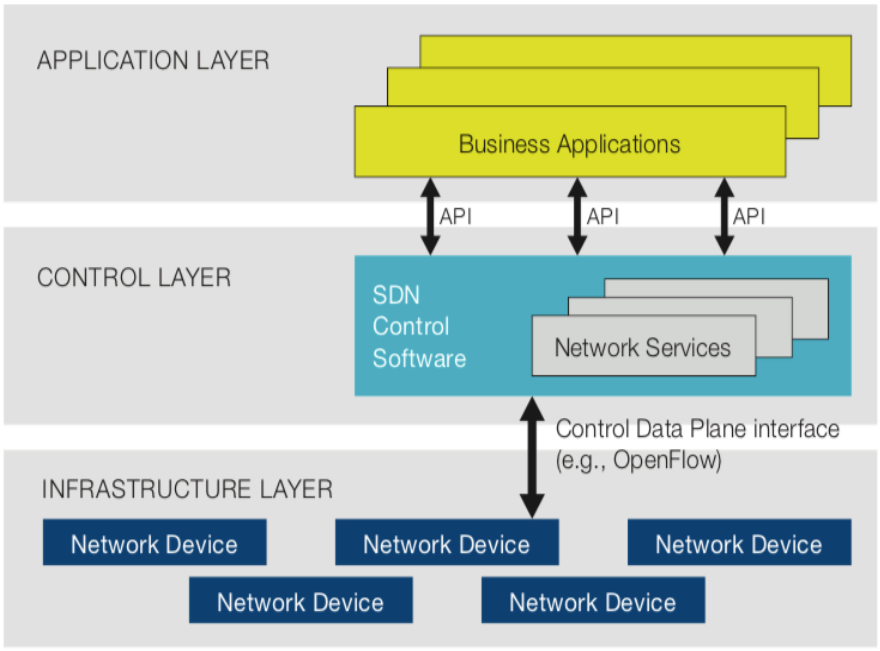

Figure 1: SDN Architecture

Figure 1 [1] shows the architecture of SDN. The infrastructure layer consists of the network devices and their data planes, which are controlled by a controller(Ryu, NOX, etc) via a Protocol (OpenFlow). Programmer programs the controller which uses the OpenFlow protocol to manage the data plane. The controller can also be accessed via its APIs, which in turn will manage the network devices.

2.1 Openflow Protocol

OpenFlow is a protocol which connects the data plane and the control plane. It specifies the rules and instructions which can be imposed on the OpenFlow switch by the control plane. OpenFlow defines different types of table and requires minimum of one table to be present on the switch. The tables contains Flows, which are nothing but a combination of Match fields and the corresponding Actions. Match fields contain the rules against which packet data will be compared. For example, match fields will have fields like Destination Port, Source Port, Protocol type, DSCP, etc. The actions are the instructions which are needed to be executed on the incoming packet. For example, the instructions can be drop, forward packet to a specific interface, modify the packet, etc. The Flows in the table have several different fields apart from match fields and actions like priority, timeout, number of packets matched, etc. If the incoming packet is not matched on any of the rules in the flow table, then it will be forwarded to the control plane, and then the controller can decide how to treat the packet.

There are several different controller platforms like NOX, POX, RYU, etc. In the current project, I have used RYU controller and the modules which come with it for packet parsing. I have implemented a Flow Management System on a test bed consisting of a Client, Router and a Server. For

testing the flow management system, I have used Scapy to generate different types of flows.

The following sections describe the test bed, the management system, results and future work respectively.

3 Implementation

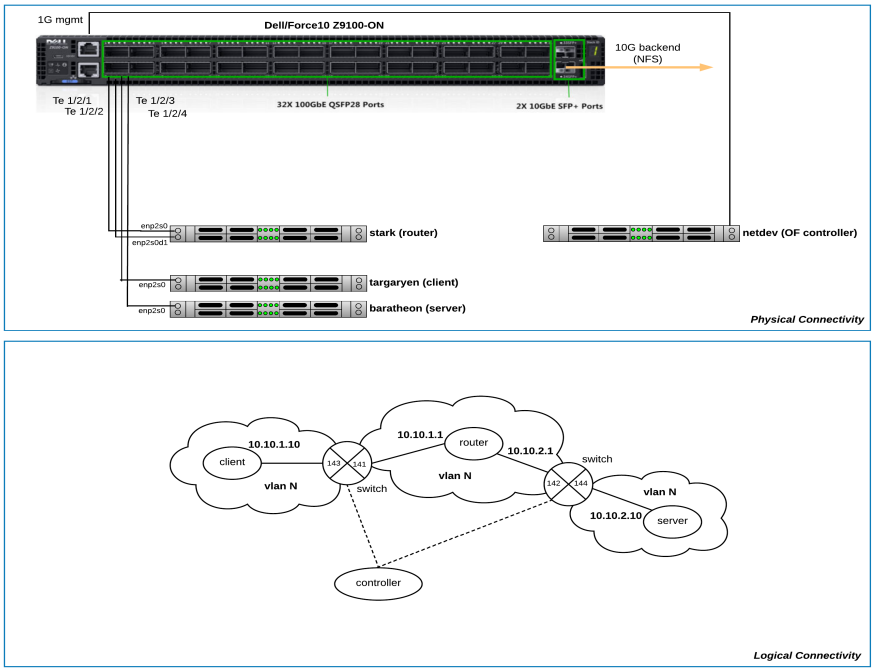

Figure 2: NetSys Test Bed

As per figure 2 [2], the test bed consists of a client on IP 10.10.1.10, a router having two IPs 10.10.1.1 and 10.10.2.1 in two different subnets respectively, and a server with IP 10.10.2.10. The client is connected to the router via a switch having two interfaces on ports 141 and 143. Similarly the server is connected to the router via a switch having two interfaces on ports 142 and 144. Both the switches are managed by Ryu Controller. This subnet is present in a VLAN, thus all the traffic flowing between the hosts consist of VLAN header. All this is hosted as a single OF-instance on DELL Z9100 switch. The controller has one table – the ACL table.

3.1 Adding flows

The flow management system is programmed using the RYU controller in Python. The RYU controller is configured in such a way that any incoming packet which doesn't match any entry in the Table is forwarded to the controller.

Once the controller receives the packet, it parses it to get the relevant fields. Initially, the controller parses the Ethernet Header to get the destination and source interface. In the test bed, all the

packets are VLAN tagged, because of which there is a VLAN header present in the packet containing the EtherType field.

Match	Action
ARP	Forward
ICMP	Forward
TCP, Dest Port:500-700	Drop
UDP, Dest Port: 1700-2000	Drop
TCP	Forward
UDP, Port: 25	Forward, Set DSCP = 26
UDP	Forward

Table 1: Different Flows

Depending on the fields stored after parsing the packet, the controller decides which flows to add. If the EtherType is of ARP then, the controller adds a flow for packets of type ARP. If the EtherType is of IP, then the controller further parses the IP header to get the IP Protocol field value. If the IP protocol is of type ICMP, then the controller adds flow to forward the ICMP packet. If the IP protocol is of type TCP and UDP, then the flows added depend on the contents of the TCP and UDP header fields. If the TCP packet's destination port is between 500 and 700, then it adds the flow to drop the packet, instead of forwarding. Similarly, if the UDP packet's destination port is between 1700 and 2000 then it adds flow to not forward the packet, but drop it. If the packet is of destined for Mail Server i.e the destination port is 25, then it marks the packet with DSCP field of 26 (High Priority Traffic), and forwards it. For all the other destination port of the TCP and UDP the controller adds flows to forward the packet.

3.2 Generating Traffic

For adding flows to the switch using controller, necessary flow traffic should be generated which matches the corresponding fields in the controller.

Initially, I tried generating traffic using iPerf3. iPerf3 was able to generate high data size traffic, but it only matched one type of flow. The requirement was to test the flow management systems response to huge number of different flows at the same time. Other way to generate different types of traffic which will match unique flows is using Bash script and combination of iperf3 and ping. I used Scapy which is a python module to generate flow traffic. Using Scapy along with Python, I generated huge number of unique flows and sent it to the controller. Unique flows consisted of TCP/UDP traffic having different destination ports, as the main objective was to test how many flows can the controller handle, and not what kinds of flow.

4 Results

After sending flow generator traffic to the controller, it was able to add flows almost instantly. Though there is a limit to the number of flows the controller can add to its ACL table. The controller is not able to add more than 733 flows to the table. Even if the controller has incoming flow traffic from the switch which matches the add flow criteria in the controller, it doesn't add new flow to the table. Thus, once the limit is reached no traffic which doesn't match the existing flows will be forwarded to the outbound interface. Furthermore, as the number of flows in table increases, accessing the OpenFlow switch though CLI becomes slower. This performance impact doesn't affect the existing flow traffic. The switch still forwards the matching flow traffic instantly.

5 Conclusion and Future Work

I have developed a flow management system on the Dell Z9100 OpenFlow switch using Ryu Controller. The flow management system consists of flows related to ARP, ICMP, TCP and UDP with specific destination ports, special treatment of SMTP traffic, and blocking traffic on certain TCP/UDP destination ports. To test how the controller responds to the incoming flow traffic, I used Scapy to generate traffic. After flow traffic generation, I found out that the Dell Z9100 OpenFlow switch controlled by a Ryu controller in the above described test bed, can't store more than 733 flows in its ACL table.

One of things which can be done to have the switch handle more flows is to have the switch delete the old flows which are not used much or rather the ones which are least recently used. The switch has statistics of the number of packets matched per flow, which can be pulled by the controller. As new packet reaches the controller, and the number of flows equal to 733, then the switch can decide the flow to delete and replace it with the new flow for the new packet.

References

[1] Software-Defined Networking: The New Norm for Networks. ONF White Paper.

[2] NetSys Test Bed image from SICE-netsys/NS-SP18 GitHub.

[3] The Road to SDN: An Intellectual History of Programmable Networks. Jenniger Rexford, Ellen Zegura, Nick Feamster

[4] Malware Detection for Mobile Devices Using Software-Defined Networking. Ruofan Jin, Bing Wang.