STUDY OF SDN USING MININET

Aashish Dugar, Dr. M Madiajagan
Dept. of Computer Science, Birla Institute of Technology and Science Pilani-Dubai Campus,
Dubai, United Arab Emirates
f2012303@dubai.bits-pilani.ac.in

Abstract :The traditional architecture of networks has long supported the needs of the data centers. However, with the today's heavy data usage by users, the need for a shift to virtualizing functions and applying a more modern approach to configure them has become inevitable. One such application that has emerged is Software Defined Networking(SDN). SDNs can be tested on an everyday PC with the help of Mininet, a platform used for simulation of the same.

Keywords: Software Defined Networks, Architecture, OpenFlow

1. Introduction

The usage of internet has far surpassed the regular mail and browsing purposes. Controlling these networks have become a lot more large scale than the current architectures can handle. The data requirements of individual users have called for the advent of network architectures that can support data centers with the dynamic applications and services deployed today. Thus came the SDNs and NFV (Network Function Virtualization).

In general, SDNs helps perform the function of routers and switches via and external controller. Normally the controlling plane would be present in the router/switch itself, however in these architectures the controlling plane rescinds from the forwarding plane, having "virtualized" the forwarding tables and decisions to a separate controller. Rather than each switch making path planning decisions that may result in loops or redundant data flow, the controller with a bird's eye view of the entire network can help efficiently and securely move data packets.

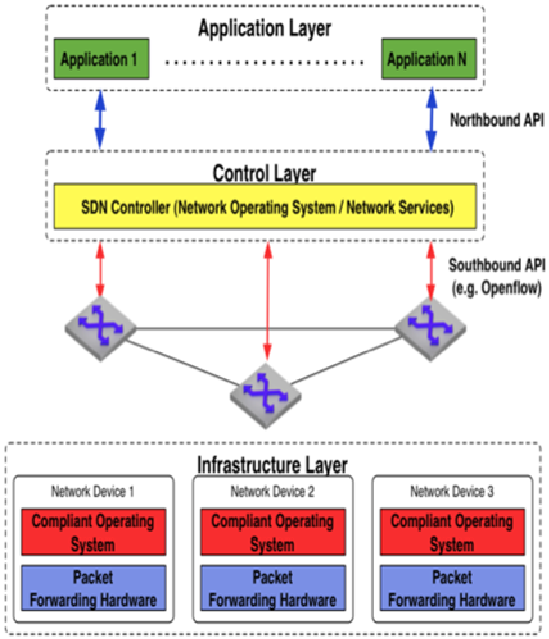

Fig. 1. SDN Architecture

This controller is connected to the breadth of the network and commands take place through the OpenFlow protocol.

SDN and OpenFlow are mistaken to be the same concept at times. OpenFlow is a communication protocol like TCP or UDP with set instructions and guidelines. SDN is the broader term where OpenFlow serves as a crucial part. However, this protocol isn't a hard and fast implementation. There are others that can take its place, but this is the more widely accepted protocol.

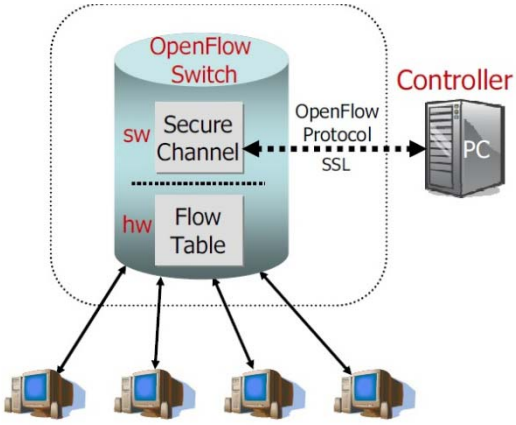

Fig. 2. OpenFlow Switch Design [McKeown et al., (2008)]

Some of the prominent slated benefits of the SDN are-

- Reduction in cost of hardware as the routers and switches will be reduced.
- Energy efficient
- Innovation for new products
- Upgrade and running cycles will be optimized
- Ease of Access

2. Traditional vs. New

The primary difference between the two generations of architecture is essentially the positioning and usage of the control plane. The traditional forms had a singular control-data plane where the control plane would hold and provide information for the forwarding tables. The data plane would consult the forwarding tables for information on where to send the packets or frame of data to.

However, these networks are quite inflexible and were soon rendered replaceable due to quite a few reasons-

(1) In terms of scalability, the networks become highly unstable and unsustainable after a point. IT, based on predictable traffic patterns, have tried to overflow their networks. However, traffic patterns have become so incredibly dynamic that it is not possible to predict their patterns.

(2)Almost all of it is hardware oriented and have to be manually programmed and set-up for usage, for e.g. switches, routers, ASICs etc. Due to the dedication and fixed role of each individual hardware, tasks such as overhauling or even making minor changes to the architecture become quite tedious.

(3)There are a number of things to be taken into regard for something as simple as adding or moving a device, such as firewalls, web authentication, protocols to be implemented, vendor model, type of network topology etc. This has to led to today's networks becoming static to prevent loss of service. This also makes the implementation of a policy across a network very tedious as each and every physical device has to be reconfigured to support that protocol.

(4)There also lies the competition amongst vendors. Due to this conflict, vendors also provide tools and hardware specific to their software, making it even more perplexing.

With the rise of virtualizing networks, all the problems of the hardware historic might not be solved, but it does manage to improve and manage todays networks with a lot more efficacy [Perrin and Hubbard, (2013)].

(1) Automation reduces complexity. This is one of the primary principles revolving around for SDNs. The automation will help reduce operation costs, decrease network instability due to operator error. It also helps to introduce cloud based management of apps to the business models of companies more easily.

(2) Improvement and changes to the existing network can be done constantly as the hardware does not have to be continuously altered. This allows the program and reprogramming of the network to satisfy the clients. This allows for much greater control over the creation of the network.

(3) Its considered to be Vendor-Agnostic, meaning the control can lie entirely with the enterprise. This allows for a wider reach with the vendor and clients.

(4) It gives nearly atomic level control to the administrator, which would allow changes and protocols to be implemented at the level of the session going on, the user currently online and even the device being used.

3. Theoretical Implementation

The SDN Architecture has been conducted using Mininet. Mininet helps to emulate virtual networks, hosts, switches and controllers on any home computer [Vanjari and Ingle, (2015)]. It runs on Linux and its switches support OpenFlow, thereby allowing the user to customize the routing and implement SDNs. The parts of the SDN implemented in our set-up are-

- SDN Controller - Situated on the control plane which determines the forwarding path for the packets at each node and updates the routing tables at data plane nodes.
- SDN Forwarding Element/Networking Devices - Data Plane node which forwards packets with reference to the forwarding tables.
- SDN Application - Helps with communication between the controller and APIs. Can also help in various other tasks such as build network view with information from the controller for further tasks such as network management, analysis of any breach etc.

We've selected 2 ways to form our SDN. One will be through the basic user functions entered on the terminal itself in the shell. The other would involve modding a python script to carry out more complex user functions.

The technical specifications involve running Ubuntu 16.04 on a VMWare Workstation 12, Mininet 2.2.1 and the latest version of WireShark that comes with the above deck.

3.1 Shell Commands

To initialize Mininet we need to run-

$ sudomn

This will create a simple SDN with 2 hosts, h1 and h2, a switch s2, and a controller c0. The topography for the command is given below

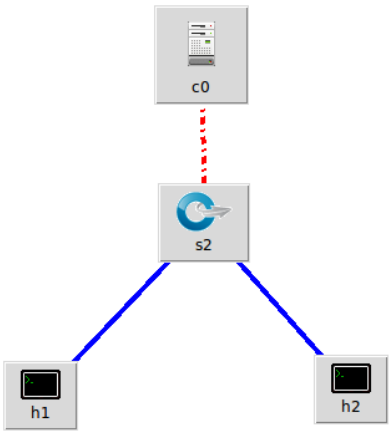

Fig. 3. Default Mininet Topology

To check if the above topography is established and connected, we can run the following command-

$ sudomn --test pingall

Some of the other test configurations that can be created in the same manner are-

$ sudomn --topo single,4 --switch ovsk --controller remote --mac

Where the commands mean the following-

- topo- Style of topology to implement, we can also specify the number of switches that we would like to use. The different kinds of topologies that we can implement are single, linear, tree etc.
- switch- Assign a specific kind of switch (kernel mode ovs in our case.)
- controller- Essentially the kind and the number of controllers you would like to implement.
- mac- Auto assign a MAC address.

We ran a ping test on nodes to check their RTT, or Round Trip Time on different topologies with similar features. There were quite some unique results. The set-ups we've used were-

(1) Single style topology with 3 switches,3 hosts and the default controller.

(2) Linear style topology with 3 switches,3 hosts and the default controller.

(3) Tree style topology with 7 switches,8 hosts and the default controller (the code during input will be similar, however it will produce more switches and hosts because the code involves depth of the tree being the output for given input, i.e. 3 levels for 7 switches).

The CLI commands needed for the above setup were-

- Mininet> h1 ping -c1 h2

This command searches for h2 and sends 1 packet of data from h1 to h2. H1 then receives a confirmation that the packet has been delivered.

- Mininet> net

This command displays all the information regarding the links used between switches and hosts. It helps to understand the form of topology upon implementing each type.

- Mininet>iperf

To test the TCP bandwidth between the specified hosts (default case is h1 and h2).

- Mininet>dptcl

This is a powerful function as it lets us modify the forwarding tables to our liking. Normally in traditional networks the switches do not have any foresight and rely on a previously set out path for packet forwarding, even though a shorter path may be available [DeCusatis et al., (2016)].

3.2 Python Script

This method involves creating a network topology through a python script. It helps in giving us more freedom to create various topologies and by adding more complex functions to the program and achieve varied results. The script that we will run is –

```
#!usr/bin/python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
#importing libraries to invoke below functions

class Network(Topo):
        def build(self,n=3):
#Topo is overridden by the build fn.Also self might be the default function
                switch=self.addSwitch('s1')
                for h in range(n):
                        host=self.addHost('h%s'%(h+1))
                        self.addLink(host,switch)

def Test():
        topo= Network(n=3)
        net= Mininet(topo)
        net.start()
        print "dumping all existing connections"
        dumpNodeConnections(net.host)
        print "Testing"
        net.pingAll()
        net.stop()

if __name__ == '__main__':
        setLogLevel('info')     #Extra information to be printed out. info,o/p,debug etc.
        Test()

topos={'topo1':Network}         #Dictionary function for compiling file via CLI
tests={'test':Test}
```

Fig.4.Script for running architecture via Mininet

The above is script is then run through the terminal with the following command. We need to enter 'sudo' as Mininet requires administrative privileges to run –

$ sudo -E python <name of file>.py

Mininet has the unique feature to display output via Wireshark, so we've used the program to test our results and see that our connections and links have been verified.

4. Results

4.1 Shell Commands

Upon running the ping test on the 3 topologies, the following results were obtained running the commands noted above –

For (1)

```
aashish@ubuntu:~$ sudo mn --topo single,3 --switch ovsk --mac --controller default
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0
c0
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.73 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.733/2.733/2.733/0.000 ms
mininet>
```

For (2)

```
aashish@ubuntu:~$ sudo mn --topo linear,3 --switch ovsk --mac --controller default
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=5.73 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 5.738/5.738/5.738/0.000 ms
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth2
s2 lo:  s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo:  s3-eth1:h3-eth0 s3-eth2:s2-eth3
c0
```

Fig. 6.Linear type architecture results

For (3)

```
aashish@ubuntu:~$ sudo mn --topo tree,3 --switch ovsk --mac --controller default
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
*** Adding links:
(s1, s2) (s1, s5) (s2, s3) (s2, s4) (s3, h1) (s3, h2) (s4, h3) (s4, h4) (s5, s6) (s5, s7) (s6, h5) (s6, h6) (s7, h7) (s7, h8)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
c0
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7 ...n
*** Starting CLI:
mininet> net
h1 h1-eth0:s3-eth1
h2 h2-eth0:s3-eth2
h3 h3-eth0:s4-eth1
h4 h4-eth0:s4-eth2
h5 h5-eth0:s6-eth1
h6 h6-eth0:s6-eth2
h7 h7-eth0:s7-eth1
h8 h8-eth0:s7-eth2
s1 lo:  s1-eth1:s2-eth3 s1-eth2:s5-eth3
s2 lo:  s2-eth1:s3-eth3 s2-eth2:s4-eth3 s2-eth3:s1-eth1
s3 lo:  s3-eth1:h1-eth0 s3-eth2:h2-eth0 s3-eth3:s2-eth1
s4 lo:  s4-eth1:h3-eth0 s4-eth2:h4-eth0 s4-eth3:s2-eth2
s5 lo:  s5-eth1:s6-eth3 s5-eth2:s7-eth3 s5-eth3:s1-eth2
s6 lo:  s6-eth1:h5-eth0 s6-eth2:h6-eth0 s6-eth3:s5-eth1
s7 lo:  s7-eth1:h7-eth0 s7-eth2:h8-eth0 s7-eth3:s5-eth2
c0
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=3.30 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.306/3.306/3.306/0.000 ms
mininet>
```

Fig.7.Tree type architecture results

The curious thing to be noted is the RTT 2, i.e., the links have already been discovered between the hosts. As we can see the initial links each had their own average time for a packet's RTT based on the topology. However, after the location has been established for a known data port, the RTT time averages nearly to the same for all topologies. This is of course assuming similar, if not ideal conditions, for all the topologies in terms of bandwidth access, topology establishment time etc. Given below are the 2nd RTT and the final results.

For (1)

```
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.395 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.395/0.395/0.395/0.000 ms
```

Fig. 8. RTT 2 results for Single Topology

For (2)

```
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.350 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.350/0.350/0.350/0.000 ms
mininet>
```

Fig. 9.RTT 2 results for Linear Topology

For (3)

```
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.344 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.344/0.344/0.344/0.000 ms
mininet>
```

Fig.10.RTT 2 results for Tree Topology

The following final results were noted-

Table 1.Topology Comparison Results

Topology	Hosts	Switches	Links	RTT 1	RTT 2(after route is configured and established)
Single	3	3	4	3.056	0.395
Linear	3	3	8	5.763	0.350
Tree	8	7	15	3.898	0.344

4.2 Python Script

Upon running an SDN using the script feature, the following output was obtained-

```
aashish@ubuntu:~/Documents$ sudo -E python mbsdn.py
*** Creating network
*** Adding controller
Traceback (most recent call last):
  File "mbsdn.py", line 28, in <module>
    Test()
  File "mbsdn.py", line 18, in Test
    net= Mininet(topo)
  File "build/bdist.linux-x86_64/egg/mininet/net.py", line 172, in __init__
  File "build/bdist.linux-x86_64/egg/mininet/net.py", line 500, in build
  File "build/bdist.linux-x86_64/egg/mininet/net.py", line 467, in buildFromTopo
  File "build/bdist.linux-x86_64/egg/mininet/net.py", line 283, in addController
  File "build/bdist.linux-x86_64/egg/mininet/node.py", line 1554, in DefaultController
  File "build/bdist.linux-x86_64/egg/mininet/node.py", line 1376, in __init__
  File "build/bdist.linux-x86_64/egg/mininet/node.py", line 1394, in checkListening
Exception: Please shut down the controller which is running on port 6653:
Active Internet connections (servers and established)
tcp        0      0 0.0.0.0:6653            0.0.0.0:*               LISTEN      6657/controller
tcp        0      0 127.0.0.1:6653          127.0.0.1:55444         ESTABLISHED 6657/controller
tcp        0      0 127.0.0.1:55444         127.0.0.1:6653          ESTABLISHED 4136/ovs-vswitchd
aashish@ubuntu:~/Documents$ sudo wireshark
```

Fig.11.Running Python script through Terminal

Using WireShark, we could see the ‘OpenFlow’ protocol being implemented between the hosts during the time of running the Address Resolution Protocol (ARP).

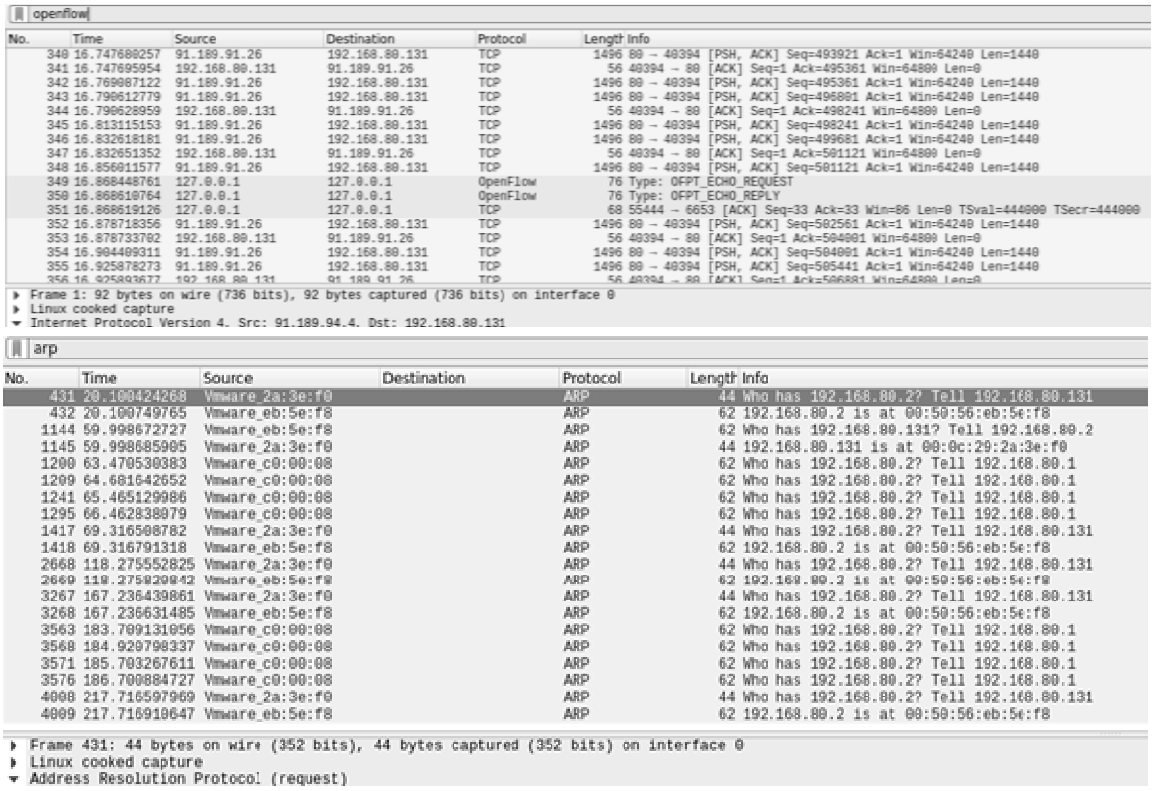

Fig.12.ARP and OpenFlow protocols detected in WireShark

The interaction between the hosts is clearly visible, with the addition of one more step, showing its communication with the controller. The continuous requests have to use this additional step to traverse through the OpenFlow protocol in order to access information and route addresses about the other hosts. This is the essentially the basis of SDN.

5. Conclusion

Traditional Networks, due to today's virtualization, security and mobility, are soon to be replaced with the next gen networking architectures. Factors like experimentation on network without the host divulging information, writing software pertaining to the vendor's need in the case of updates all add to the advantage of next gen network architectures. This trend of SDNs is soon to flourish within cloud computing, data centers and related applications.

Here we have discussed how to use Mininet to implement a controller, create a simple architecture and to use some commands to capture important information. This information ranges from checking client connection to the server, capture OpenFlow messages or online activity using Wireshark.

Future prospects include the rapid integration of SDNs to the current professional and scholastic environment, interconnecting networks and overlays, and majorly trying to implement decentralized multi-controller environments to prevent single point failure, handle complexity upon growth and in general, reliability and scalability.

References

[1] D. Nadeau T.; Gray K. (2013): Software Defined Networks, 1st edn., OReilly Media Inc.
[2] DeCusatis, C., et al. (2016): Modeling Software Defined Networks using Mininet, 133, pp. 1-6.
[3] Fernandez M.P., et al. (2013): Comparing OpenFlow Controller Paradigms Scalability: Reactive and Proactive, Advanced Information Networking and Applications (AINA), IEEE, pp. 1009–1016.
[4] Kaur K., et al. (2014): Mininet as Software Defined Networking Testing Platform, 29, pp. 139-142.
[5] Lara A., et al. (2014): Network innovation using Open-Flow: A survey, Communications Surveys Tutorials, IEEE, Vol. 16, No. 1, pp. 493-512.
[6] McKeown, et al. (2008): OpenFlow: Enabling Innovation in Campus Networks, ACM SIGCOMM, 38, No.2, pp. 69–74.
[7] Perrin, S.; Hubbard, S. (2013): Practical Implementation of the SDN and the NFV in the WAN, Heavy Reading, pp. 2-11.
[8] Pooja; Sood M. (2015): SDN and Mininet: Some Basic Concepts, IJANA Vol. 7, Issue 2 pp. 2690-2693.
[9] Vanjari, S. A.; Ingle, R.B. (2015): Network Traffic Control Using Distributed Control Plane of Software Defined Network, IJCST Vol. 3, Issue 5, pp. 177-181.