Journal of Network Communications and Emerging Technologies (JNCET)
Volume 11, Issue 1, January (2021)

# Implementing and Experimenting with OpenMUL Scalability in Software Defined Network

Shivalika Singh
Department of Computer Science, CHRIST, Bangalore, Karnataka, India
shivalika.singh@mca.christuniversity.in

Bhargavi Goswami
School of Computer Science, Queensland University of Technology, Brisbane, Queensland, Australia
bhargavi.goswami@qut.edu.au

Joy Paulose
Department of Computer Science, CHRIST, Bangalore, Karnataka, India
joy.paulose@christuniversity.in

Libin Thomas
Department of Computer Science, CHRIST, Bangalore, Karnataka, India
libin.thomas@christuniversity.in

Published online: 31 January 2021

**ABSTRACT**

**Traditional networks are plagued by many disadvantages. One of the major problems affecting networks is that software updates for networking devices such as routers or switches are always vendor controlled. This led to the birth of a new networking phenomenon known as software defined networking (SDN). Unlike traditional networking devices which have their individual control and data planes, SDN architecture offers a common, centralized control plane for all networking devices that are attached to a network. The brain of an SDN network is the controller which sits in the control plane and takes care of all requests within a network. Thus, it is crucial to choose a controller which can be scaled according to the dynamic changes in the network. One of the popular choices for a controller during the creation of an SDN environment is OpenMUL. In this paper, authors evaluate the performance of OpenMUL in terms of scalability by taking multiple scenarios into account which demonstrates incremental growth in the number of hosts configured to the SDN network. These scenarios are simulated using networking tools such as Mininet, OpenMUL and iPerf. The variations in performance metrics are observed and analyzed rigorously for each of the defined scenarios.**

**Index Terms – Software Defined Network (SDN), Mininet, Python, OpenFlow, OpenMUL, iPerf.**

## 1. INTRODUCTION

The expansion of networks has become unstoppable with the advent of newer technologies. As a result, traditional networking devices are not well-equipped to handle data processing in an efficient manner. Traditional networking devices have limited scope for programming. If a company wants specific software upgrades for the devices configured to its network, then it may have to wait for an unprecedented amount of time since all software updates are vendor controlled.

By the time an update is released, its need may have become obsolete since the company may have worked out an alternative solution for its earlier requirements. Any modifications to the configuration of the network requires an engineer to manually write scripts for updating the network. This manual entry of scripts often gives rise to several problems. For example, wrong configuration of connected devices unknowingly leads to hiring a larger taskforce to configure all networking devices. Additionally, these devices end up failing if they encounter any event for which they don't have a predetermined response. Thus, the community of researchers in the domain of networking came up with the model of SDN as a solution to all the limitations of traditional networking.

A networking device constructed based on the traditional network architecture consists of three planes, i.e. Management Plane, Data or Forwarding Plane [1] and Control plane. The combination of all these planes in the same device makes the network complex and hard to manage [2]. SDN makes the control plane centralized and common for all devices attached to a network. Hence,

switches and routers connected to an SDN network consist only of data and management planes. Some of the commonly used SDN controllers are OpenDaylight [3], Floodlight [4], ONOS [5], Ryu, Beacon [6], etc. Our focus in this paper stays at MUL Controller.

As mentioned, the control plane is the brain of the SDN architecture. It is said so because it is the job of the control plane to determine whether to accept or drop a new incoming packet depending on the guidelines enforced by the network administrator. Similarly, the control plane also decides how to push data to the route tables of switches.

Some of the SDN controllers with varying features such as POX [7], OpenDaylight, Floodlight, Beacon [8], Ryu [9], NOX, etc, were compared by the authors in [10]. The authors of this paper have tried to assess the performance MUL controller as well as provide an analysis of its features in comparison with other popular SDN controllers.

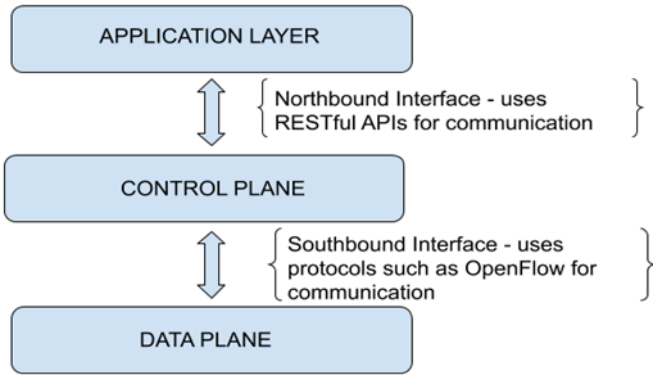


Figure 1 SDN Architecture

Figure 1 describes the SDN architecture which consists of two main components: (1) Northbound interface: It is connected to the application layer. In the case of OpenMUL, the northbound interface is connected to NB API (NorthBound Application Programming Interface) applications which use REST (Representational State Transfer) APIs for communication. (2) Southbound interface: It is responsible for providing abstractions for discovery. In the case of OpenMuL, the southbound interface enables communication between MuL multi-threaded Core and the data plane via protocols such as OpenFlow. The MuL multi-threaded core is the infrastructure that is used for handling devices and applications.

## 2. COMPARISON OF OPENMUL WITH OTHER CONTROLLERS

This section highlights the differences that exist between OpenMuL and other SDN controllers. It is mentioned for the convenience of the research community and network administrators so that they can make informed decisions before investing huge funds into constructing an SDN network. As mentioned earlier, the controller is the brain of an SDN network, if the correct controller is not chosen then the SDN network will not function at optimum efficiency.

ONOS: It is a popular choice for SDN controllers as it is open source. It was released by Open Networking Lab in collaboration with enterprises such as AT&T and NTT. The controller is written entirely in Java. It supports protocols such as SNMP, NETCONF and OpenFlow 1.0-1.3 for southbound communication and REST APIs for northbound communication. The architecture of ONOS is distributed in nature. Some of the advantages of such an architecture is high availability and easy scalability.

OpenDayLight: It is also an open-source controller handled by the Linux Foundation along with several enterprises in the domain of computing such as Dell, Cisco, VMWare, Intel, etc. The controller is written entirely in Java which implies that it's compatible with any platform capable of running Java. OpenDayLight can support OpenFlow 1.0-1.4, NETCONF, OVSDB, YANG, BGP/LS, SNMP for southbound communication and REST APIs as a northbound interface. The controller's architecture is distributed and highly modular in nature.

Floodlight: Floodlight is an OpenFlow controller which implies it supports only OpenFlow as a protocol for southbound interface whereas applications communicate with the controller using HTTP-REST commands. It is programmed using Java under the Apache license. It is endorsed by a community of developers including Big Switch Networks. The architecture of the controller is centralized in nature, that is, the controller takes care of routine tasks to keep a check on the network, whereas the applications are designed to address various requirements of the user over a network.

Beacon: It has become a popular choice for an SDN controller since its development over the last decade. One of its most desirable features is cross platform support. The controller is developed in Java and it supports OpenFlow for southbound communication along with REST APIs for northbound communication. The controller is designed with a centralized architecture consisting of

dedicated applications to perform several tasks such as processing of incoming and outgoing packets, setting up of topology, pushing route details of packets to flow tables of switches, etc.

RYU: It is an open-source controller available for constructing SDN frameworks. It is also widely known as a Network Operating System (NOS) since it supports several protocols for communication with devices that sit in the data plane of an SDN framework. Some of these protocols are NetFlow, VRRP (Virtual Router Redundancy Protocol), SNMP (Simple Network Management Protocol), NETCONF, OVSDB (Open vSwitch Database Management Protocol), sFlow OF-Config and OpenFlow versions 1.0-1.4. Ryu is a Python based SDN controller and offers a component-based architecture. Such an architecture offers the ability to scale the network with ease as the demand grows at the cost of drop in efficiency.

OpenMUL: It is also an open source SDN controller which is designed using C language by the OpenMUL Foundation. It offers a multi-threaded framework that has the capability to run applications that are modular in nature and guarantees a high level of performance. It can support protocols such as OF 1.0-1.4, OVSDB, OFCONFIG for southbound communication and uses REST APIs to communicate with the huge range of applications that are part of its northbound interface. The architecture of OpenMUL has a distributed and modular design which offers flexible functionalities for the construction of networks via a simple interface with numerous access points.

After performing a comparative study of some of the most commonly used controllers in the domain of SDN, it can be said that most of the controllers which sit in the control plane of any SDN network are developed in languages like Java and Python. However, the majority of implementation of the forwarding plane is done in C language (e.g. OpenvSwitch) which is used for communication using OpenFlow in SDN networks. A controller also written in C language can communicate more effectively with the hardware components in the forwarding plane. This is one major advantage OpenMUL has over other controllers as it is developed in C.

Another advantage OpenMUL has over other controllers is its distributed architecture which helps it keep separate address space for core applications and base networking services. It also supports standard protocols used in industry for communication with the data plane instead of being just an OpenFlow based SDN controller. A feature that gives OpenMUL edge over the other controllers are the various kinds of APIs it offers for running applications with different requirements. It supports REST APIs for running web applications, the Python APIs can be used for the purpose of faster application development whereas the bindings written in C are used for running performance intensive applications.

In conclusion it can be said that OpenMUL can be the fundamental unit of any SDN framework, allowing users to minimize the difference between the actual and expected performance of the framework. It also enables in extending the life of existing networks and leveraging new resources and functionalities only available in SDN.

## 3. OPENMUL CONTROLLER

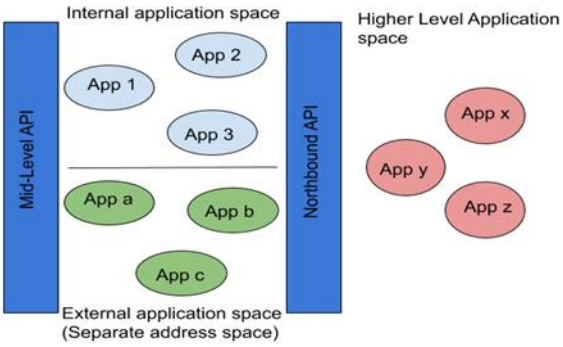


Figure 2 MUL Architecture

OpenMUL is an open source SDN controller written in C. The controller has a multi-threaded infrastructure which ensures stability, high performance and the ability to run modular applications. OpenMUL has the capacity to work with several southbound protocols such as NETCONF [11], OpenFlow [12], etc. Its northbound interface can support several kinds of applications and uses REST APIs such as ML-API and NB-API as an interface for accessing these applications [13]. Applications utilizing ML-API have the choice to be run in line with the corresponding process address space as MuL core, as shown in Figure 2 [14]. This improves the performance of the application. The SDKs provided by MuL enable the developer to run applications without understanding the underlying process of how they are executed.

MuL is designed for flexibility. It supports an architecture which is modular in nature and offers features that are adaptable for the setup and management of networks via an interface consisting of numerous access points. MuL also has an extremely modular architecture which allows dynamic insertion, deletion and updation of components without affecting the rest.

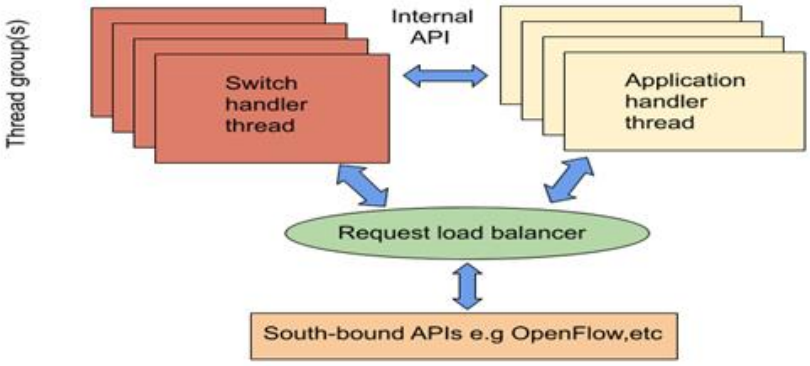


Figure 3 MUL Core Architecture

Following are the major components [15] of OpenMUL controller:

(1) Mul director/core: It is an essential unit of the controller which is responsible for handling processing related to Openflow and all switch connections as shown in Figure 3 [14]. One of its crucial jobs is to ensure that all switch centric details such as groups, flows are kept in sync across all switches. It maintains logs of failures encountered by the controller. It also provides an interface for programming applications in the form of mid-level APIs.

(2) Mul infrastructure services: On top of the MuL director/core sits the infrastructure services of the controller. The services provided by MuL are adaptive service chaining, network stats monitor, legacy network plugin etc.

(3) MuL system apps: A shared API between the core and the services is used to design the system applications. If the switches support basic dependencies of these applications, then they can be used across various versions of Openflow. CLI and L2switch are some of the applications provided by the controller.

## 4. SIMULATION ENVIRONMENT

To conduct this experiment, OpenMUL VM is used along with Mininet, iPerf and gnuplot. Mininet [16] is used as a simulator and OpenMuL controller is used as the SDN controller. The OpenMUL VM is available for download on the official site of OpenMUL [17].

The authors have used OpenFlow kernel switch in this experiment by activating OpenFlow protocol 1.3. Table 1 summarizes the dependencies of this experiment.

| **OS** | **Ubuntu 14.04.02 LTS** |
|---|---|
| Hypervisor | Oracle VM VirtualBox |
| Mininet | 2.2.1 |
| Iperf | 2.0.5 |
| Gnuplot | 4.6 |
| OpenFlow | 1.3 |
| RAM | 8GB |

Table 1 System Configuration Specifications for Experiment

In this experiment, a mesh topology consisting of five switches for six scenarios is implemented as shown in Table 2. Only the number of hosts configured to the network is incrementally changed under each scenario. The steps to be followed to perform this experiment are described in detail in this section. The tool used in this experiment to obtain network related statistics is iPerf [18].

The scripting language used to create a network topology is Python. Figure 4 represents the python code used to set up a network environment that consists of a controller, switches and hosts. The python code specifies the connection between all the different components of the generated network i.e. hosts to switches, switches to other switches and switches to OpenMuL controller.

| Experimental Scenario | No. of Hosts |
|---|---|
| Experimental Scenario 1 | 50 |
| Experimental Scenario 2 | 100 |
| Experimental Scenario 3 | 200 |
| Experimental Scenario 4 | 400 |
| Experimental Scenario 5 | 600 |
| Experimental Scenario 6 | 800 |

Table 2 Scenario Table for experiments

```
from mininet.topo import Topo
class MyTopo(Topo):

    def __init__(self):
        # Initialize topology
        Topo.__init__(self)

    # Add switches
    for s1Switch in range(1):
        s1Switch = self.addSwitch('s1')
    for s2Switch in range(1):
        s2Switch = self.addSwitch('s2')
        self.addLink(s1Switch,s2Switch)
    for s3Switch in range(1):
        s3Switch = self.addSwitch('s3')
        self.addLink(s2Switch,s3Switch)
        self.addLink(s1Switch,s3Switch)
    for s4Switch in range(1):
        s4Switch = self.addSwitch('s4')
        self.addLink(s1Switch,s4Switch)
        self.addLink(s2Switch,s4Switch)
        self.addLink(s3Switch,s4Switch)
    for s5Switch in range(1):
        s5Switch = self.addSwitch('s5')
        self.addLink(s1Switch,s5Switch)
        self.addLink(s2Switch,s5Switch)
        self.addLink(s3Switch,s5Switch)
        self.addLink(s4Switch,s5Switch)
#for loop to add hosts and links to the switches
for h1_ in range(0,10):
    h1=self.addHost('h1_%s' % (h1_+1))
    self.addLink(h1,s1Switch)
for h2_ in range(10,20):
    h2=self.addHost('h2_%s' % (h2_+1))
    self.addLink(s2Switch, h2)
for h3_ in range(20,30):
    h3=self.addHost('h3_%s' % (h3_+1))
    self.addLink(s3Switch, h3)
for h4_ in range(30,40):
    h4=self.addHost('h4_%s' % (h4_+1))
    self.addLink(s4Switch, h4)
for h5_ in range(40,50):
    h5=self.addHost('h5_%s' % (h5_+1))
    self.addLink(s5Switch, h5)
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Figure 4 Python script for creating topology

The python code for creating the custom topology defines a class “MyTopo” which is a child class of the class “Topo” predefined in Mininet libraries. The code begins by invoking the constructors of both parent and child class for the purpose of initializing the topology. The “self.addSwitch” command is used for adding switches to the topology. Since the attempt is to create a mesh topology for reduced end-to-end delay, each switch being created is linked to other switches that are part of the topology.

Steps for running the experiment is provided further so that researchers can recreate the experiment for further expansion.

Step 1: Start the controller by running the following set of commands in the specified order: cd openmul ; ./mul.sh init ; ./mul.sh start l2switch. Figure 5 shows how the command prompt looks like once the controller is running.

```
openmul@openmul:~/openmul$ ./mul.sh start l2switch
[MUL startup script]
OpenMUL is stopped...
OpenMUL l2switch mode is running..
```

Figure 5 Running the controller

Step 2: Once the controller is running, the second step requires to setup the network topology with the help of the python script mentioned in Figure 4. While invoking the script, details about the controller being used such as its IP and port number are also provided once the command is fired: sudo mn --custom mash.py -- topo mytopo -controller=remote,ip=127.0.0.1, port=6653

```
*** Adding hosts:
h1_1 h1_2 h1_3 h1_4 h1_5 h1_6 h1_7 h1_8 h1_9 h1_10 h2_11 h2_12 h2_13 h2_14 h2_
15 h2_16 h2_17 h2_18 h2_19 h2_20 h3_21 h3_22 h3_23 h3_24 h3_25 h3_26 h3_27 h3_
28 h3_29 h3_30 h4_31 h4_32 h4_33 h4_34 h4_35 h4_36 h4_37 h4_38 h4_39 h4_40 h5_
41 h5_42 h5_43 h5_44 h5_45 h5_46 h5_47 h5_48 h5_49 h5_50
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(h1_1, s1) (h1_2, s1) (h1_3, s1) (h1_4, s1) (h1_5, s1) (h1_6, s1) (h1_7, s1) (
h1_8, s1) (h1_9, s1) (h1_10, s1) (s1, s2) (s1, s3) (s1, s4) (s1, s5) (s2, h2_1
1) (s2, h2_12) (s2, h2_13) (s2, h2_14) (s2, h2_15) (s2, h2_16) (s2, h2_17) (s2
, h2_18) (s2, h2_19) (s2, h2_20) (s2, s3) (s2, s4) (s2, s5) (s3, h3_21) (s3, h
3_22) (s3, h3_23) (s3, h3_24) (s3, h3_25) (s3, h3_26) (s3, h3_27) (s3, h3_28)
(s3, h3_29) (s3, h3_30) (s3, s4) (s3, s5) (s4, h4_31) (s4, h4_32) (s4, h4_33)
(s4, h4_34) (s4, h4_35) (s4, h4_36) (s4, h4_37) (s4, h4_38) (s4, h4_39) (s4, h
4_40) (s4, s5) (s5, h5_41) (s5, h5_42) (s5, h5_43) (s5, h5_44) (s5, h5_45) (s5
, h5_46) (s5, h5_47) (s5, h5_48) (s5, h5_49) (s5, h5_50)
```

Figure 6 Topology Created using Mininet

Once the command is run, it creates the topology specified within the python script as shown in Figure 6. To ensure that all hosts are able to communicate with each other, the mininet command pingall is used. This will make each host ping all the other hosts existing within the network.

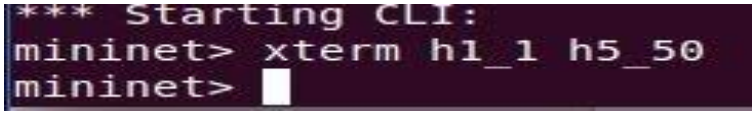


Figure 7 Opening Windows of Two Hosts

Step 3: The next step involves choosing any two hosts as client and server. Then the command shown in Figure 7 is used for creating two windows: one for server and the other for client. It has been assumed that h1_1 is the server and the h5_50 is the client. The naming convention followed can be understood as follows: h means host, 1_1 implies switch 1 host 1 and 5_50 implies switch 5 and host 50.


Figure 8 Command for Server Window

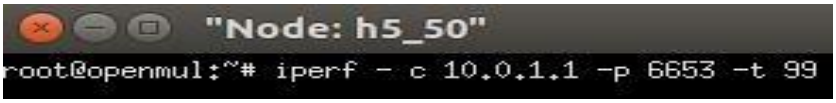


Figure 9 Command for Client Window


Figure 10 Command Run in Server Window to Obtain Jitter Log


Figure 11 Command Run in Client Window to Obtain Jitter Log

Step 4: This step involves generating traffic between server and client with the help of the tool iPerf. The first half of this step involves activating the server window by running the command shown in Figure 8. This command prompts the server to listen to the client and write all data related to communication between them to the text file "Output". The second half of this step involves generating traffic at the client end using the command shown in Figure 9. Here 99 is time in seconds, –c stands for client and -p stands for port number. The IP and port number of the server should also be known to run this command.

Step 5: After generating traffic and recording its output in 'Ouput.txt', jitter data is collected. The first half of this step involves activating the server by the use of the command shown in Figure 10. Here, "jitter" is the name of the file used for storing the results. The second half of this step involves setting up of exchange of traffic between the server and client by use of this command shown in Figure 11.

Step 6: This step involves extracting only the relevant information from throughput and jitter log files so that they can be further used for visual interpretation. Figure 12 represents collected throughput data for a particular scenario and Figure 13 represents collected jitter data for a scenario.

For filtering, grep and awk are used as shown in Figure 14. This command selects the specified throughput data from the file "Output" and writes it to the file "outputresults". Figure 15 shows the command that is used for extracting the jitter values from the jitter file and writes it to the file "jitterresults".

Step 7: This step involves representing extracted data from the throughput and jitter files graphically. For this purpose, a file named "plot.plt" consisting of the commands shown in Figure 16 is created. Then, "plot.plt" is run using the following command: "gnuplot -p plot.plt". This command plots a graph using all the throughput values obtained for a scenario.

```
------------------------------------------------------------
Server listening on TCP port 6653
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[230] local 10.0.0.1 port 6653 connected with 10.0.0.2 port 49040
[ ID] Interval       Transfer     Bandwidth
[230]  0.0- 1.0 sec   700 MBytes  5.87 Gbits/sec
[230]  1.0- 2.0 sec   396 MBytes  3.32 Gbits/sec
[230]  2.0- 3.0 sec   486 MBytes  4.08 Gbits/sec
[230]  3.0- 4.0 sec   278 MBytes  2.33 Gbits/sec
[230]  4.0- 5.0 sec   614 MBytes  5.15 Gbits/sec
[230]  5.0- 6.0 sec   749 MBytes  6.28 Gbits/sec
[230]  6.0- 7.0 sec   697 MBytes  5.85 Gbits/sec
[230]  7.0- 8.0 sec   674 MBytes  5.65 Gbits/sec
[230]  8.0- 9.0 sec   717 MBytes  6.01 Gbits/sec
[230]  9.0- 10.0 sec   656 MBytes  5.50 Gbits/sec
[230] 10.0- 11.0 sec   701 MBytes  5.88 Gbits/sec
```

Figure 12 Format of Output file Generated Based on Communication between Client and Server.

```
------------------------------------------------------------
Server listening on UDP port 6653
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[229] local 10.0.0.1 port 6653 connected with 10.0.0.2 port 52097
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total
Datagrams
[229]  0.0- 1.0 sec  1.19 MBytes  9.98 Mbits/sec   0.097 ms    0/  849 (0%)
[229]  1.0- 2.0 sec  1.19 MBytes  10.0 Mbits/sec   0.066 ms    0/  851 (0%)
[229]  2.0- 3.0 sec  1.19 MBytes  10.0 Mbits/sec   0.385 ms    0/  850 (0%)
[229]  3.0- 4.0 sec  1.19 MBytes  9.95 Mbits/sec   0.218 ms    0/  846 (0%)
[229]  4.0- 5.0 sec  1.19 MBytes  9.97 Mbits/sec   0.310 ms    0/  848 (0%)
[229]  5.0- 6.0 sec  1.18 MBytes  9.90 Mbits/sec   0.080 ms    0/  842 (0%)
[229]  6.0- 7.0 sec  1.19 MBytes  9.98 Mbits/sec   0.230 ms    0/  849 (0%)
[229]  7.0- 8.0 sec  1.19 MBytes  9.96 Mbits/sec   0.156 ms    0/  847 (0%)
[229]  8.0- 9.0 sec  1.19 MBytes  9.97 Mbits/sec   0.177 ms    0/  848 (0%)
[229]  9.0- 10.0 sec  1.18 MBytes  9.93 Mbits/sec   0.211 ms    0/  844 (0%)
[229] 10.0- 11.0 sec  1.19 MBytes  10.0 Mbits/sec   0.062 ms    0/  850 (0%)
[229] 11.0- 12.0 sec  1.18 MBytes  9.88 Mbits/sec   0.329 ms    0/  840 (0%)
[229] 12.0- 13.0 sec  1.16 MBytes  9.71 Mbits/sec   0.126 ms    0/  826 (0%)
[229] 13.0- 14.0 sec  1.14 MBytes  9.54 Mbits/sec   0.239 ms    0/  811 (0%)
```

Figure 13 Format of Output file Generated for Jitter Based on Communication between Client and Server.

```
openmul@openmul:~$ cat Output|head -100|grep sec|awk '{print$3,$5}' >outputre
sults
```

Figure 14 Command to Extract Time and Throughput Values from “Output.txt” File

```
openmul@openmul:~$ cat jitter|head -106|grep sec|awk '{print$3,$9}' >jitterres
ults
```

Figure 15 Command to Extract Time and Jitter Values from “jitter.txt” File

For generating graph for jitter values the script shown in Figure 17 is used and is named as plot1.plt. Then, “plot1.plt” is run using the following command: “gnuplot -p plot1.plt”. This command plots a graph using all the jitter values obtained for a scenario.

```
set xlabel "Interval_Time (In seconds)"
set ylabel "Bandwidth_Utilization (in GBps)"
plot "outputresults" title "TCP Flow-50 Nodes" with linespoints
```

Figure 16 Script for Printing Throughput Values

```
set xlabel "Interval_Time (In seconds)"
set ylabel "Jitter (in ms)"
plot "jitterresults" title "Jitter-50 Nodes" with linespoints
```

Figure 17 Script for Printing Jitter Values

All the above-mentioned steps are run sequentially for all the scenarios specified under Table 2 using the system configurations specified in Table 1.

## 5. PERFORMANCE ANALYSIS

This section is meant for the evaluation of results procured for each of the six scenarios with incremental changes in the number of hosts configured to the network. The results generated for the first scenario are shown in Figure 18. It can be seen from the graph that the controller is able to handle the load over the network with ease. The throughput maintained an average value between 4-6GBps.

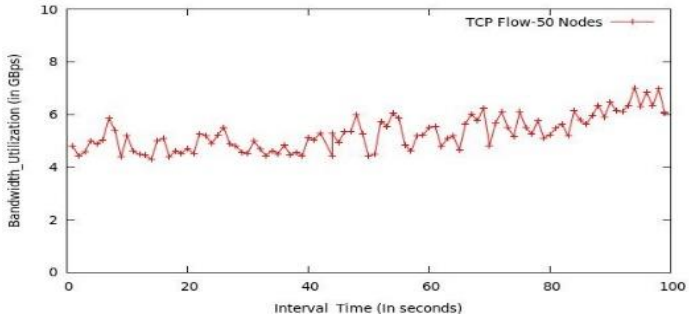


Figure 18 Throughput of Scenario with 50 Nodes

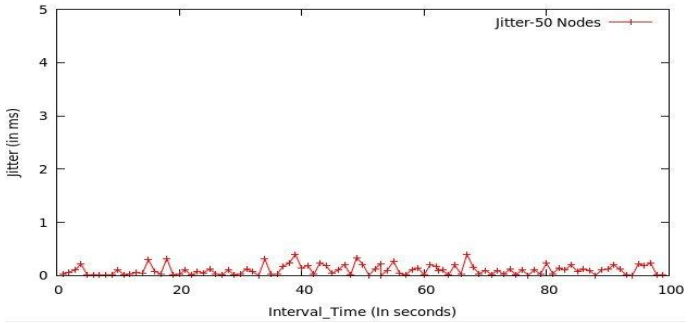


Figure 19 Jitter for Scenario with 50 Nodes

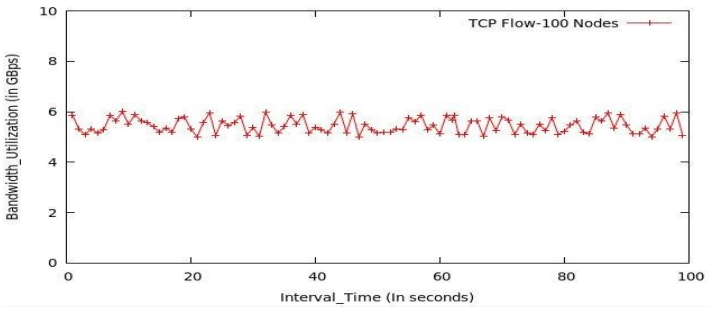


Figure 20 Throughput for Scenario 100 Nodes

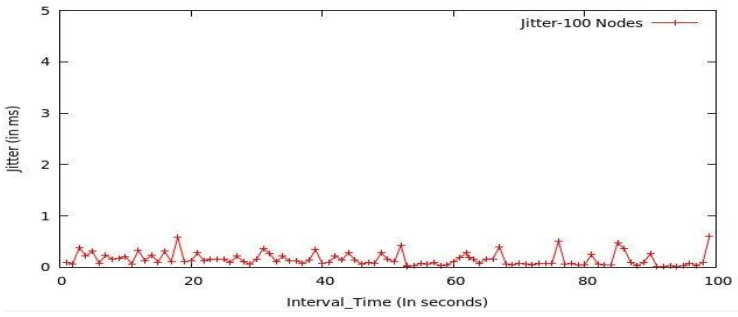


Figure 21 Jitter for Scenario with 100 Nodes

Figure 19 illustrates jitter values obtained in the case of scenario 1 containing 50 nodes. Jitter in a network is a measure of latency over time. A spike in jitter values indicate an anomaly, that is, some packets took longer time to travel from one host to another. For scenario 1, the jitter values stayed between 0.0 to 0.4ms. No major spikes are observed under this scenario. Hence, it can be assumed that there is no delay in the delivery of packets.

The performance of the controller was then evaluated for the scenario consisting of 100 hosts connected to five different switches. The throughput values stabilized in comparison with scenario 1. It was observed that the network was stable with an increase in throughput values with an average of 5-6GBps as shown in Figure 20.

Figure 21 illustrates jitter values obtained in the case of scenario 2 containing 100 nodes. The jitter values fluctuate between 0.1 to 0.6ms. The jitter values have increased when comparison to the previous scenario indicating a higher rate of delay in the delivery of packets. Although it is still very small in magnitude.

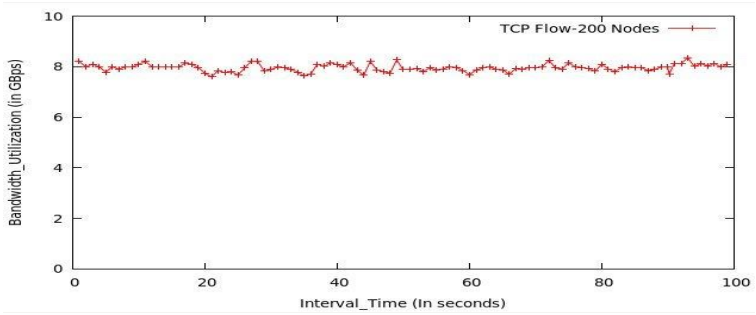


Figure 22 Throughput for Scenario with 200 Nodes

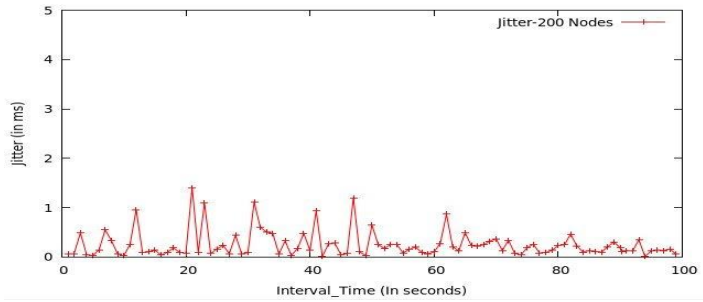


Figure 23 Jitter for Scenario with 200 Nodes

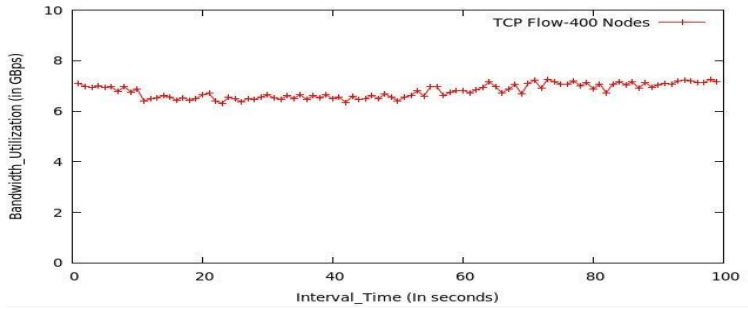


Figure 24 Throughput for Scenario with 400 Nodes

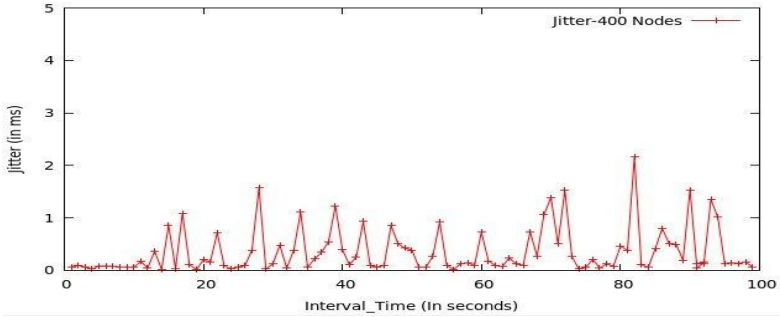


Figure 25 Jitter for Scenario with 400 Nodes

The performance of the controller is further assessed for the scenario containing 200 hosts. As shown in Figure 22, it was noted that the throughput values continue to rise with an increase in the number of hosts. The average throughput was 8 Gbps for this scenario.

Figure 23 illustrates jitter values obtained in case of scenario 3 containing 200 nodes. The jitter values fluctuated between 0.1 to 1.4ms. Most peaks were concentrated in the first half of the graph and jitter is observed to have reduced towards the second half.

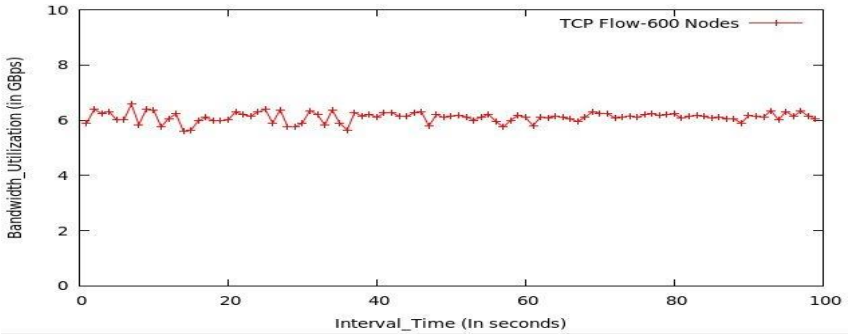


Figure 26 Throughput for Scenario with 600 Nodes

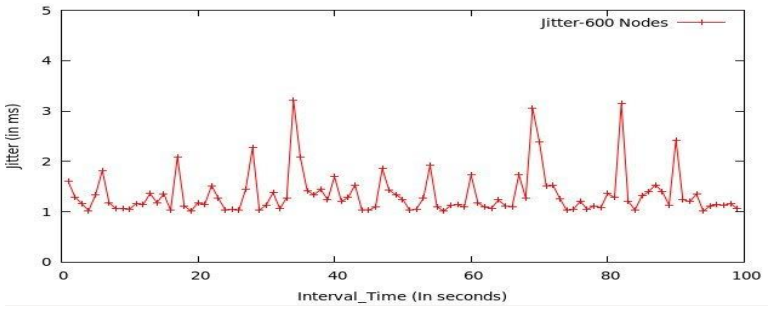


Figure 27 Jitter for Scenario with 600 Nodes

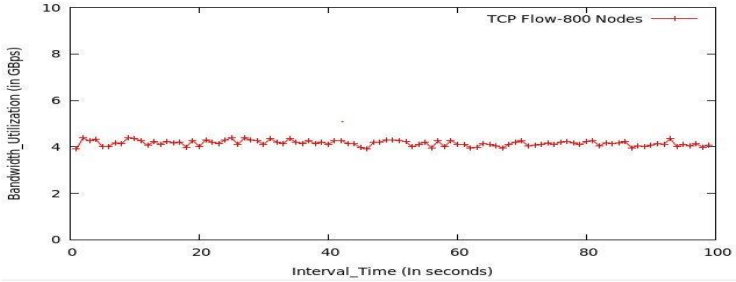


Figure 28 Throughput for Scenario with 800 Nodes

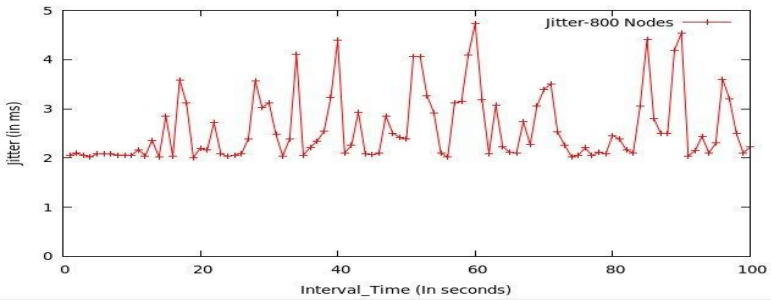


Figure 29 Jitter for Scenario with 800 Nodes

Moving forward, more hosts were added to the simulated network and the output was observed for the scenario with 400 nodes. It was observed that average throughput reduced and stayed between the range of 6.5-7 Gbps as shown in Figure 24.

Figure 25 illustrates jitter values obtained in case of scenario 4 containing 400 nodes. The jitter values fluctuate between 0.1 to 2ms. The spikes in the jitter values have increased by a considerable amount when compared with previous scenarios. This implies that with increasing number of nodes some packets take longer to travel from one host to another.

While testing 600 nodes scenario, it was observed that the throughput decreased in comparison to previous scenarios reaching an average of 6 Gbps as shown in Figure 26. There were a few packet drops in the beginning but the throughput stabilized over time.

Figure 27 illustrates jitter values obtained in the case of scenario 5 containing 600 nodes. The jitter values fluctuate between 1 to 3ms. This is a considerable increase in jitter values in comparison to previous scenarios indicating higher rate of delay in delivery of packets from one host to another.

While testing the 800 nodes scenario, it was observed that the throughput further decreased in comparison to the previous scenario reaching the average of 4 Gbps as shown in Figure 28.

Figure 29 illustrates jitter values obtained in the case of scenario 6 containing 800 nodes. The jitter values fluctuate between 2 to 4.5ms which is the highest in comparison to all previous scenarios. The number of spikes is also the highest under this scenario indicating that a lot of packets took much longer to be delivered than the minimum delay time which was 2ms for this scenario.

## 6. CONCLUSION

The aim of this paper was to present the results of experimentation with scalability of OpenMUL controller under different scenarios. With a focus on the throughput and jitter values, the authors have provided a detailed explanation of how to conduct the experiment including all details related to testbed setup as well as the list of commands needed for execution of experiments in multiple network scenarios. As discussed under the performance analysis section, the controller behaves in a stabilized manner as the number of hosts increases during experimentation. The authors have also provided a comparison of OpenMUL controller with respect to other popular controllers in the domain of SDN in terms of supported programming languages as well as supported northbound and southbound APIs, etc. In conclusion it can be said that OpenMUL can help leverage a lot of functionalities offered by the SDN architecture as it offers multiple APIs for running vast varieties of applications, faster communication with hardware in data plane and high scalability.

## REFERENCES

[1] Khan M.A., Goswami B., Asadollahi S. (2020) Data Visualization of Software-Defined Networks During Load Balancing Experiment Using Floodlight Controller. In: *Announce S., Gohel H., Vairamuthu S. (eds) Data Visualization*. Springer, Singapore. DOI: https://doi.org/10.1007/978-981-15-2282-6_9.
[2] Sameer et al. (2018) RYU controller's scalability experiment on software defined networks. In *IEEE International Conference in Current Trends in Advanced Computing (ICCTAC)*, 1-5.
[3] Asadollahi, S. and Goswami, B (2017) Implementation of SDN using Open Daylight Controller. In *Proceeding of An International Conference on Recent Trends in IT Innovations-Tec'afe*, 218-227.
[4] Asadollahi, S., & Goswami, B. (2017). Experimenting with scalability of floodlight controller in software defined networks. In: *International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)* (pp. 1–5). IEEE, Mysore, India.
[5] Sameer, M and Goswami, B (2018) Experimenting with ONOS Scalability on Software Defined Network. In *Journal of Advanced Research in Dynamical & Control Systems*, Vol. 10, 14-Special Issue, 1820-1830.
[6] Manuel, Tony and Goswami, B (2019). Experimenting with Scalability of Beacon Controller in Software Defined Network. In *International Journal of Recent Technology and Engineering (IJRTE)*, Volume-7 Issue-5S2, 550-555.
[7] Ankit Kumar, Bhargavi Goswami, Peter Augustine (2019), Experimenting with Resilience and Scalability of Wi-Fi Mininet on Small to Large SDN Networks. In *International Journal of Recent Technology and Engineering (IJRTE)*, SCOPUS, Volume-7, Issue-6s5, pp. 201-207.
[8] Erickson, D (2013) The beacon open flow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 13-18.
[9] Saleh Asadollahi, Bhargavi H Goswami, (2017) Revolution in Existing Network under the Influence of Software Defined Network. In *Proceedings of the 11th INDIACom*, Pages: 1012-1017, IEEE, New Delhi, India.
[10] Asadollahi, S. and Goswami, B (2017) Software Defined Network, Controller Comparison. *Proceedings of Tec'afe 5* (2), 211-217.
[11] S Asadollahi, B Goswami, Software Defined Network, Controller Comparison. In *IJIRCCE*, Vol.5, Special Issue 2, April 2017, Pg. No. 211 – 217
[12] McKeown et al (2008) OpenFlow: enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, 2008, 69-74.
[13] S Das, B Goswami, S Asadollahi, Investigating Software-Defined Network and Networks-Function Virtualization for Emergent Network-oriented Services. In *IJIRCCE*, Vol.5, Special Issue 2, April 2017, Pg. No. 201 – 205, DOI:10.15680.
[14] OpenMul Architecture. Available at https://openmul.wordpress.com/2012/10/02/mul-architecture-in-a-nutshell/. Accessed February 2020
[15] OpenMUL: SDN Controller. Available at http://www.openmul.org/openmul-controller.html. Accessed March 2020
[16] Mininet: Emulator. Available at https://www.mininet.org. Accessed January 2019)
[17] OpenMUL: SDN Controller. Available at www.openmul.org/openmul-vm.html Accessed January 2019
[18] iPerf: Networks tool. Available at https://www.iperf.fr. Accessed January 2019

AUTHORS

**Shivalika Singh** Has academic background in Physics, Mathematics, Electronics and Computer Science which helps her develop a sense of appreciation and understanding of overarching branches of science. She is engineer by profession, but she enjoys both, research and software development as being a widely applicable profile in research and software development.

**Bhargavi Goswami** is a profound researcher and academician having more than ten years of experience as Assistant Professor with a forty plus research publications and two patents. Currently, she is working in the domain of Data Communication Networks, Software-Defined Networks and Neighborhood Area Networks of Smart Grids. She is known for her unique research implementation skills. Her paper includes detailed research procedures which can be recreated and experimented further and therefore, followed widely among newbies in the research domain.

**Joy Paulose** has served education industry for more than 25 years. Currently he is Professor and Head of the Department of Computer Science at Christ University. His error free administration and remarkable management skills are comparable to none. His contribution in the domain of research-oriented skill development in Computer Science Department has improved the university ranking principally making other departments follow the model and strategies designed and developed by him with rigorous implementation and analysis.

**Libin Thomas** is an PhD Scholar at Christ (Deemed to be University), Bangalore, India. He is currently working on Machine Learning, Mobile Communications and Ad Hoc Networks. His expertise is in IoT and computer networks. He has experience as Delivery Consultant trainee and a Teaching Assistant. Currently he is a researcher working in the fields of Vehicular Ad hoc Networks and Machine Learning.