Software Defined Network: Stress Testing using Fuzzy Inputs

Rafael Lourenco and Yagnik Suchak

University of California, Davis
{rlourenco, yvsuchak}@ucdavis.edu

1. ABSTRACT

Software Defined Network is considered to be the latest revolution in the field of communication networks. SDN decouples control plane & data plane thus enabling a lot of new opportunities for innovation. It promises to dramatically simplify network management and enable innovation through network programmability. There is a lot of research going on in the field of SDN, especially the control plane. There are a number of SDN controllers available in the market but what lacks is a study that shows the comparison of how various controllers would behave when implemented in the same network. Considering that SDN is a large scale system in itself, the space of possible inputs(e.g., packet headers and inter-packet timings) is huge. Owing to the number of variables involved in SDN system, we propose a system that is capable of stress testing of SDN using fuzzy inputs. In this project we tested ONOS and Opendaylight (two of the most popular SDN controllers) using fuzzy inputs and compared the performance of both the controllers in terms of CPU utilization and Memory consumption.

2. INTRODUCTION

Today communication networks are experiencing revolution. It goes under the name of Software Defined Networks. It enables 3rd party programs running over commodity computers to take control of the network and tailor its behavior to specific application. A successful example of this vision is OpenFlow which today is de facto standard protocol for controlling programmable switches [8]. In fact, there is a growing body of people across both academia and industry that believe that SDN will become the new norm for networks. Thus, SDN comes across as one stop solution for the problems that traditional network faces.

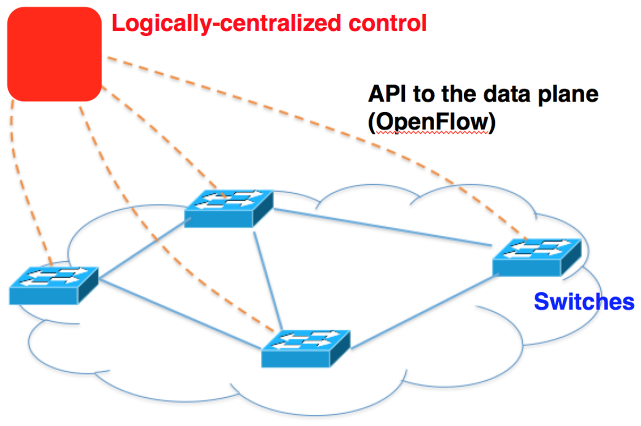

Logically-centralized control
API to the data plane (OpenFlow)
Switches

Figure 1: General idea of the SDN concept.

Although SDN networks haven't been widely deployed yet (the major exception being Google's B4 network), there are diverse plans from companies like AT&T, Verizon, China Mobile, Telecom Italia, Telefonica, Vodafone and others [13] to make it a practical reality. The characteristics of SDN (and its complementary sibling technology Network Function Virtualization - NFV) are attractive because they will allow more flexibility and adaptability of the network, improve its operation, and establish ripe conditions for the creative processes that lead to innovative technologies. In fact, due to the much expected official standardization of SDN, it is expected that new players enter the market (among them, the open software community), increasing competition and, thus: lowering the Capital Expenditures, improving system quality, and accelerating time-to-market of products.

If on the one hand, SDN enables new functionalities through programming, on the other, it increases the risk of software bugs. This might make communication unreliable and it is a major hurdle for the success of SDN. Thus, the ultimate wide adoption of SDN also depends on having effective ways to test SDN networks so as to achieve high reliability. There are certain difficulties in automating the testing of SDN. Firstly, even though the control plane is separated from data plane and is centralized, the system is inherently still distributed and asynchronous in nature [3]. There are multiple events happening at different switches and end hosts leading to inevitable delays which affects communication with the controller. Secondly, the centralized controller has an indirect view of the traffic and experiences unavoidable delays in installing rules in the switches/elements. Other than these the space of possible inputs (e.g., packet headers, inter-packet timings, packet processing time etc) and other external factors like user behavior (e.g., mobility) and higher-layer protocols (e.g., TCP) may affect the correctness of OpenFlow programs [3, 1].

3. MOTIVATING EXAMPLE

Traditional communication networks are prone to vulnerabilities due to several aspects. First of all, the different pieces of hardware utilized are many a times extremely function-specific making them prone to design shortcomings that might lead to problems. Second, the software running on those physical elements is also function-specific, relying on some sort of configuration triggered at some level by a human operator and evolves with time through updates. Third, the network topology might create bottlenecks, cross hazardous geographies, face different maintenance schedules and evolve in unpredictable manners in order to fulfill market requirements. Actually, even in a disaster-free, attack-free, bottleneck-free, evolution-predictable, perfectly-tested hardware and software network, the geography of such system might span through such a large area and serve billions of individual data packets that it becomes impossible to assume consistency between the network nodes. All these make network testing and debugging a cumbersome endeavor.

With regards to the modern SDNs, many of the problems above still exists. Particularly, on the one hand a big part of the system – the control plane – is executed on mass production, multi-purpose servers and clusters, meaning that these pieces of hardware tend to be more reliable. This brings the system bug rate down. But on the other hand, the control plane implementation depends on new abstractions, protocols, and softwares. These new applications are at least as prone to errors as any piece of software. Thus, the current higher bug count in these newly proposed softwares is likely to rebalance the system bug rate up. In fact, it can be expected that in the long run these software faults will become more rare which in turn would reduce the error probability of SDNs. Although, this is highly dependent on well matured standards and efficient testing techniques.

In fact, current testing efforts have already revealed many problems regarding widely adopted SDN technologies. In [4], the authors criticize the underlying premise on which OpenFlow operates: enforcing that the Controller has to have detailed knowledge of each switching element at all times. The authors showed that this is a costly proposition and makes the OpenFlow protocol unsatisfactory for high performance system when general purpose switches are used (in a common paradigm where the switching tables need to be frequently updated). In [14], the authors also put OpenFlow to test and conclude that setting flow rules reactively in a Data Center scenario makes the performance of the system unacceptable with only eight switches. The examples above illustrate that we are currently in an important SDN standardization phase, where testing techniques need to be effective in order to generate reliable systems and standards.

4. RELATED WORK

In [7, 9, 15, 17] the authors show that verification of controllers, network applications,some topology-specific properties (connectivity, loop freedom, access control), or network devices can be used to detect and avoid possible SDN bugs.

More specifically, [7, 9] propose two tools, VeriFlow and OFTEN, to verify correctness properties violations on the system. The first is based on an offline analysis while the second can perform online checking. The verification constraints include: security and reachability issues, configuration updates on the networks, loops, black holes, etc. In order to check for unexpected behavior, the authors in [15] use the Alloy tool for formal relational models. That is, in case the protocol under-specifies system aspects, this tool might be able to reveal specific sequence of events that generate unwanted behavior.

In [16], the authors propose two tools to test SDN elements: the first, OFlops takes an individual switch and test its behaviour in the presence of high traffic load; the second,

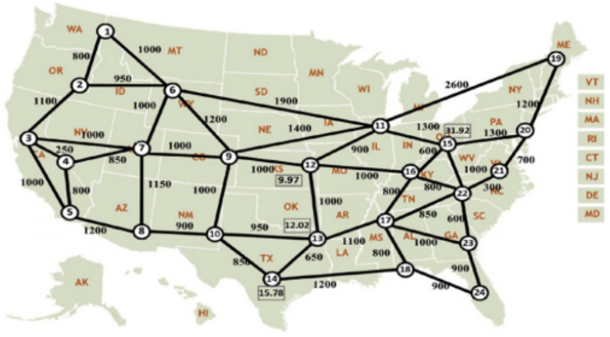


Figure 2: US 24 nodes topology.

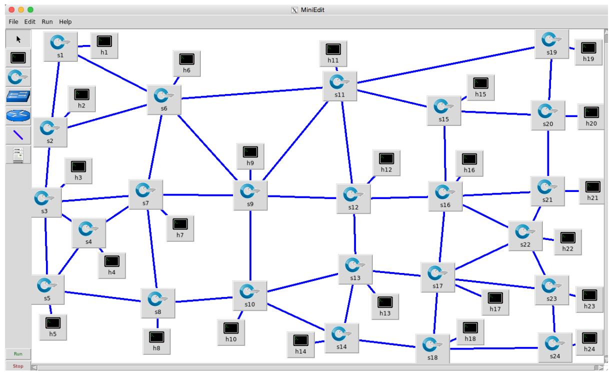


Figure 3: The MiniEdit Window with a US24 node network with one host attached to each switch.

CBench, does the same for a controller. In [6], the authors show a different controller benchmark test, OFCBenchmark tool; it is argued that this implementation is more efficient than CBench in terms of performance and revealed bugs.

The NICE suite, proposed in [3], tries to implement a symbolic execution methodology similar to KLEE in an SDN environment. Although, due to the explosive number of system states the system requires many different optimizations, approximations, and simplifications. NICE is intended to test network applications that run on top of a controller.

In general, the study of how the traffic profile in the network links affect the system wasn't studied in the SDN scenario. In fact, the traffic intensity in a network can have hazardous effects on the whole system. A very common type of attack against networks and servers is the well-known Deny Of Service (DOS) and its many variations [5]. A thorough study of the impact of different traffic shapes on the network as whole was done in [2]. Currently, it is paramount to perform a deep analysis of how an entire SDN system performs under different types of traffic, regardless of the Data and Control planes separation, and considering specific topologies.

5. TECHNICAL APPROACH

As mentioned in previous sections, we developed a tool capable of generating random (fuzzy) packets at network nodes (covering the most general traffic loads/shapes of real networks) and push these random packets to network nodes in a simulated SDN. Considering SDN is still in primitive stage, a rapid prototyping workflow is a key to unlocking its full potential. In order to emulate SDN we use Mininet which supports this assertion [10]. Mininet is a network emulator which runs a collection of end-hosts, switches, controllers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. Mininet comes with a simple GUI editor called MiniEdit that allows creation and updation of network topologies. Although useful, MiniEdit doesn't provide fine grain manipulation of elements and specific configuration of the topology, which must be done manually.

In order to emulate the SDN we need a control plane (SDN controller) and a data plane (hosts and switches). For our experimental analysis we simulated the 24 node USA network topology [Figure 1] in the data plane. Mininet allows us to configure the topology using Python scripting. As far as control plane is concerned, we could have used the default controller that comes with Mininet setup but to make the analysis more general we used other well known controllers. The chosen ones were OpenDayLight and ONOS, which are two of the most popular and promising controllers available nowadays in the SDN market. Mininet provides a way to configure a controller placed on a remote machine (physical or virtual) instead of the default controller. The mininet file (.mn extension) that consist of customised topology and information about remote controller can be programmed in Python. Following, the mininet command that runs a custom SDN with US24 topology and a remote controller can be seen. The resulting SDN can be viewed in MiniEdit [Figure 2]. One thing to note here is that the remote controller is not shown in the figure because it is not instatiated by the topology configuration itself (being placed remotely).

```
sudo mn -custom /mininet/custom/sdnFuzz.py -topo
US24topo -mac -arp -switch ovsk -controller remote
```

In order to simulate network traffic we used randpkt and the Pyhton library dpkt. The randpkt tool is part of the Wireshark packet analyzer Suite (the de facto packet analyzer tool nowadays). According to the tool's manual pages, it "produces very bad packets". In fact, the tool generates a .PCAP file with an user specified number of packets of a certain protocol type with varying sizes. Each of these packets follow the structure defined for its protocol but contain random data in all of its fields. In this sense, we were able to store the random packets for future studies and comparisons. Thus, by creating many randomized packets of a certain type, we tested the controller to see how well they handle malformed packets. When creating packets of a certain type, randpkt uses a sample packet that is stored internally to randpkt. For example, if we choose to create random ARP packets, randpkt will create a packet which contains a predetermined Ethernet II header, with the Type field set to ARP. After the Ethernet II header, it will put a random number of bytes with random values. Since these packets will be generated by the hosts, it is of interest to have the largest amount of random information in each of them. Considering the TCP/IP pile of protocols, the selected level of packet encapsulation was the Level 2 (Data Link Layer). At this level, Ethernet packets are transmited and, whenever inside one single IP network, only the Ethernet packet source and destination MAC addresses are needed to send the packet between themselves.

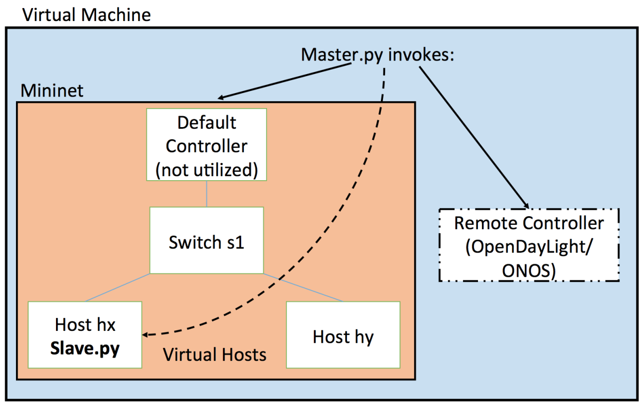
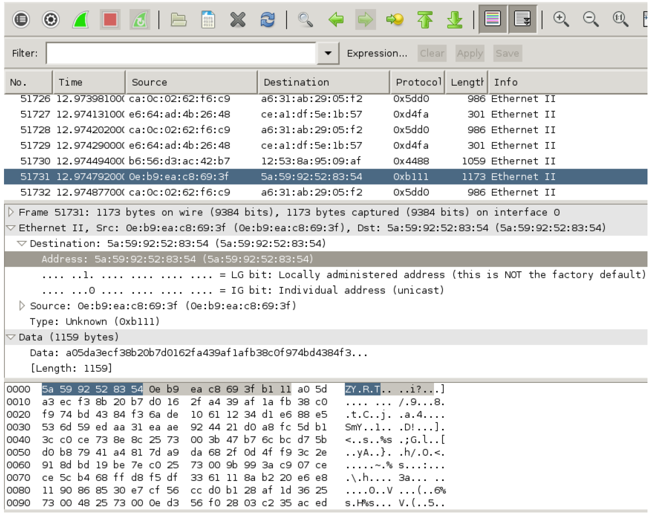


Figure 4: Schematic view of the experimental setup.



```
No.   Time         Source             Destination        Protocol Length Info
51726 12.97398100( ca:0c:02:62:f6:c9  a6:31:ab:29:05:f2  0x5dd0    986 Ethernet II
51727 12.97413100( e6:64:ad:4b:26:48  ce:a1:df:5e:1b:57  0xd4fa    301 Ethernet II
51728 12.97420200( ca:0c:02:62:f6:c9  a6:31:ab:29:05:f2  0x5dd0    986 Ethernet II
51729 12.97429000( e6:64:ad:4b:26:48  ce:a1:df:5e:1b:57  0xd4fa    301 Ethernet II
51730 12.97449400( b6:56:d3:ac:42:b7  12:53:8a:95:09:af  0x4488   1059 Ethernet II
51731 12.97479200( 0e:b9:ea:c8:69:3f  5a:59:92:52:83:54  0xb111   1173 Ethernet II
51732 12.97487700( ca:0c:02:62:f6:c9  a6:31:ab:29:05:f2  0x5dd0    986 Ethernet II

Frame 51731: 1173 bytes on wire (9384 bits), 1173 bytes captured (9384 bits) on interface 0
Ethernet II, Src: 0e:b9:ea:c8:69:3f (0e:b9:ea:c8:69:3f), Dst: 5a:59:92:52:83:54 (5a:59:92:52:83:54)
  Destination: 5a:59:92:52:83:54 (5a:59:92:52:83:54)
    Address: 5a:59:92:52:83:54 (5a:59:92:52:83:54)
    .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
    .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  Source: 0e:b9:ea:c8:69:3f (0e:b9:ea:c8:69:3f)
  Type: Unknown (0xb111)
Data (1159 bytes)
  Data: a05da3ecf38b20b7d0162fa439af1afb38c0f974bd4384f3...
  [Length: 1159]

0000  5a 59 92 52 83 54 0e b9  ea c8 69 3f b1 11 a0 5d   ZY.R.T.. ..i?...]
0010  a3 ec f3 8b 20 b7 d0 16  2f a4 39 af 1a fb 38 c0   .... ... /.9...8.
0020  f9 74 bd 43 84 f3 6a de  10 61 12 34 d1 e6 88 e5   .t.C..j. .a.4....
0030  53 6d 59 ed aa 31 ea ae  92 44 21 d0 a8 fc 5d b1   SmY..1.. .D!...].
0040  3c c0 ce 73 8e 8c 25 73  00 3b 47 b7 6c bc d7 5b   <..s..%s .;G.l..[
0050  d0 b8 79 41 a4 81 7d a9  da 68 2f 0d 4f f9 3c 2e   ..yA..}. .h/.O.<.
0060  91 8d bd 19 be 7e c0 25  73 00 9b 99 3a c9 07 ce   .....~.% s...:...
0070  ce 5c b4 68 ff d8 f5 df  33 61 11 8a b2 20 e6 e8   .\.h.... 3a... ..
0080  11 90 86 85 30 e7 cf 56  cc d0 b1 28 af 1d 36 25   ....0..V ...(..6%
0090  73 00 48 25 73 00 0e d3  56 f0 28 03 c2 35 ac ed   s.H%s... V.(..5..
```

Figure 5: A portion of The Wireshark screen displaying a stream of random packets.

Since the .PCAP files follow a specific structure, we used the "dpkt" library to enable packet parsing in our Python scripts. Because the random packets need to be transmitted through the network, each packet needs to have its randomly generated source and destination MAC address strings altered before sending. The dpkt library offers easy and useful methods to perform these changes. Also, the Socket Python libary is used to send each packet to the destination host once it is ready to be transmitted. Once the socket is binded to a ethernet interface, this library allows the sending of raw streams, making it possible to transmit pre-computed Ethernet packets through this socket.

The Mininet simulator emulates different hosts by simply creating various special bash instances on the same machine on which it is running, each of them with separate ethernet cards. These bash instances represents different hosts that can only communicate between themselves through the Mininet emulated network topology. In this sense, it is possible to run practically any type of programs in each of the hosts in a reliable manner.

Thus, we designed two Python scripts that:

1. *Master.py* invokes the remote controller and waits for

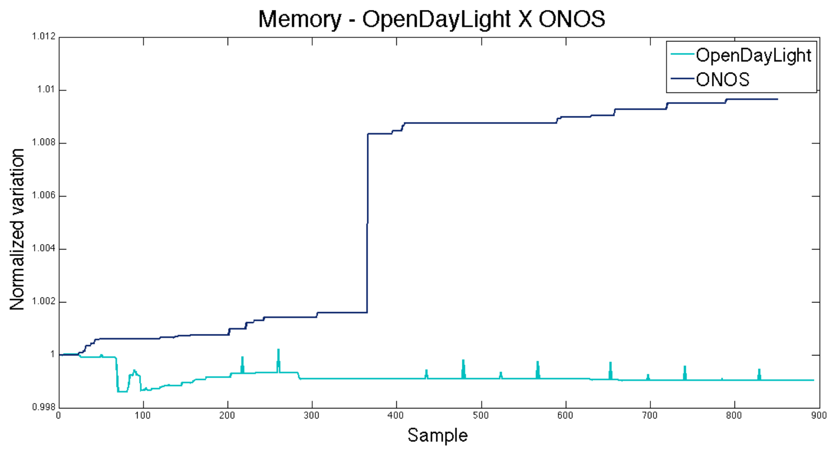


Figure 6: Memory consumption variation.

it to become operational (the boot-up process might take up to a minute).

2. Register the Process ID of the controller's main instance, which will be later on used for monitoring its performance.

3. Call Mininet and provide the necessary inputs to create the network topology shown in Figure 2.

4. Create a working directory for each specific run using a master script (master.py) on Mininet VM. This allows us to store each test run files in case they need to be re-utilized.

5. Randomly determines pairs of source-destination streams. Each pair represents a one way communication stream. In this sense, host A might send packets to G which, in turn, might send packets to C. For this work, only one-to-one streams were simulated (no multicast or broadcast).

6. Iterates through the list of hosts and, in each of the host bash instances, call the *Slave.py* script that:

   (a) Creates its separate working directory for the specific host in the unified file system.

   (b) Generates a number of random packets using randpkt and store them in PCAP file saved inside the host directory.

   (c) Retrieves each random packet and change its destination MAC address to the one selected randomly in step 5.

   (d) Sends the packet to the destination through the Mininet-emulated network.

   (e) Depending on the size of the packet and the transmission time, waits for a certain amount of time to maintain an average transmission rate in order to keep the throughput below an *upperLimit*.

   (f) Monitors the performance of the controller. If the memory, virtual memory, and CPU consumption doesn't increase by more than 10% within each 10 packets, increase the *upperLimit* rate in order to allow a higher average throughput.

   (g) Keeps sending random packets until a time period (predetermined by the *Master.py* script) is achieved.

7. Finally, the *Master.py* script will enter the monitoring state. In this stage, resource consumption of the Controller (memory, CPU, etc) is monitored, along with whether it crashes or not. The behavior is recorded for the duration of the experiment (by default, 100 seconds).

The Wireshark tool was used to monitor the traffic in each of the network interfaces of each element of the Mininet topology. This tool is very useful for packet inspection and, when used with the appropriate setup, it is possible to analyze traffic flows from source to destination.

With the aforementioned setup, the following sections sum up the achieved results.

6. RESULTS

The two selected controllers were tested several times using the previously mentioned US 24 nodes topology. Each of the controllers were minimally customized in order to avoid misconfigurations. The ONOS controller was utilized as it is provided in the 1.1 Blackbird Version that can be found at [11]. This controller comes with a big list of functionalities already working, as a very interactive Web GUI, for example. The OpenDayLight controller of version Helium SR3 [12] had to have the following features installed in order to have it properly communicating with Mininet and to provide support to an equivalent Web GUI interface:

```
odl-restconf odl-l2switch-switch odl-mdsal-apidocs
odl-dlux-core
```

In these two scenarios, the graphs 6 and 7 describe the perfomance comparison between these two controllers. It is evident that the OpenDayLight controller outperforms ONOS in terms of CPU utilization. In order to better understand how these controllers perform, it is important to perceive how the OpenFlow protocol behaves:

1. When a packet arrives at a OpenFlow switch, it verifies if the addresses in the packet are in its Flow Tables.

2. If they are, the switch redirect the packet (flow) according to the rules it already knows.

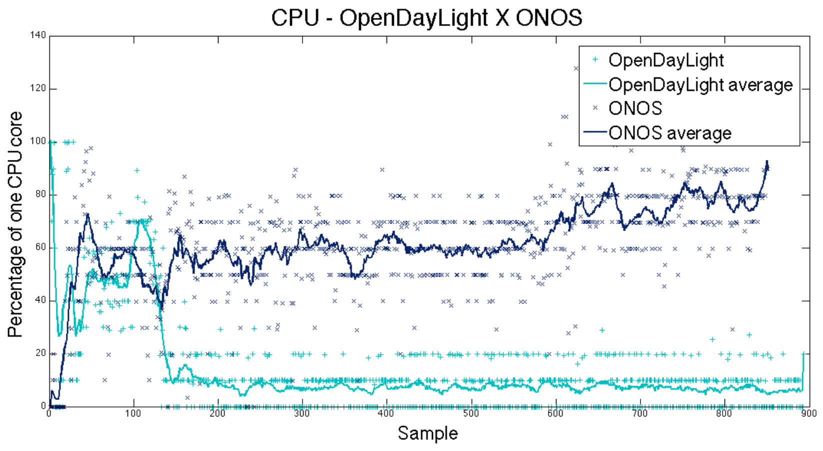


Figure 7: CPU consumption comparison.

3. If they are not, the switch sends the packet to the controller.

4. When the controller receives a packet from any of the switches, it computes the port this packet should be redirected to and informs the switch.

5. After being informed by the controller, the switch redirects the packet to the proper outbound port.

In this sense, the OpenDayLight controller has a peak of CPU utilization in the first samples when it needs to inform each of the switches how to behave. It is important to notice that the random information of the packets doesn't seem to influence the behavior of this controller. On the other hand, the ONOS controller has an ever increasing CPU utilization.

The behavior of the ONOS controller might be explained by two factors: *(a)* it is performing some sort of packet inspection and, since the packets are random, they might be leading the controller to some high processing state from where it gets no useful information; *(b)* it is configuring the switches to keep a very small flow table which needs to be updated frequently.

Finally, the memory consumption of the ONOS controller increases slightly during the execution of the experiment while the OpenDayLight memory consumption decreases by a small amount. On the other hand, the ONOS controller starts the experiment utilizing close to 650MB of memory, while the OpenDayLight controller utilizes close to 1.4GB of memory: this is a impressive difference that might be explained by which features are enabled by default in each controller (this study did not compare the controllers features).

7. INDIVIDUAL CONTRIBUTION

The team divided the papers mentioned in the references equally and discussed the summaries after reading them. After that, the implementation of various tools were divided as follows: Rafael did the installation of Mininet and OpenDayLight controller whereas Yagnik took care of ONOS controller and Packeth installation. Both the team members worked together on Miniedit and Python scripting to successfully create the 24 Node USA network topology.

The Milestone#1 report writing was divided as follows: Rafael wrote Introduction, Motivating example and Related work whereas Yagnik took care of Technical approach, Evaluation Methodology, Individual contribution and Milestone#2 Goals.

For the Milestone#2, Rafael did Wireshark's randpkt installation(it took a lot of efforts) and then both the team members worked on creation/updation of the existing Python scripts. As far as report writing is concerned, both the team members discussed the technical approach and came up with the detailed steps mentioned in Complete Technical Approach.

The final report was written by both Rafael and Yagnik. The final presentation was mostly done by Yagnik while the debbuging of the final scripts was mostly done by Rafael.

8. LIMITATIONS AND DIFFICULTIES

The results that we got in this experiment are obtained on an emulated network that runs on a single virtual machine resulting into a resource constraint (physical and logical). Mininet uses a single Linux kernel for all virtual hosts so it is not possible to run any other operating system (Windows or BSD) software. Also, Mininet hosts share the host file system and PID space so process communication among various hosts is complicated.

One of the biggest hurdle that we faced in this project was the communication between master script and the salve script. We emulated the US 24 node network so at any instance of time, there are 25 scripts(1 master and 24 slave scripts) running on one virtual machine that shares the file system and PID space. This made the overall communication very complicated. In order to overcome this difficulty we decided to call each of the slave script just once from the master script and monitor the controller behavior from all the 25 scripts simultaneously.

9. CONCLUSION

In this project two of the most popular SDN controller namely ONOS and Opendaylight were tested using fuzzy inputs. Both the controllers behavior were monitored while the network was flooded with the same set of random packets. On the basis of the preliminary results found using our experimental analysis we conclude that Opendaylight controller outperforms ONOS controller in CPU utilization. With regards to memory consumption, ONOS increases its consumption while OpenDayLight decreases but, on the other hand, ONOS requires half of the memory utilized by OpenDayLight by default. Finally, further tests in real environments are desirable in order to achieve data rates higher than those allowed by the one single machine being shared among 24 different virtual hosts.

10. REFERENCES

[1] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 31. ACM, 2014.

[2] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 71–82. ACM, 2002.

[3] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, et al. A nice way to test openflow applications. In *NSDI*, volume 12, pages 127–140, 2012.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[5] P. Ferguson. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. 2000.

[6] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries. A flexible openflow-controller benchmark. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 48–53. IEEE, 2012.

[7] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[8] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[9] M. Kuzniar, M. Canini, and D. Kostic. Often testing openflow networks. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 54–60. IEEE, 2012.

[10] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[11] ONOS. Onos, July 2015. URL `https://wiki.onosproject.org/display/ONOS11/ONOS+Wiki+Home`.

[12] OpenDayLight. Opendaylight, July 2015. URL `http://www.opendaylight.org/software/downloads`.

[13] S. Perrin. The future of carrier sdn networks, July 2015. URL `http://www.lightreading.com/carrier-sdn/sdn-architectures/the-future-of-carrier-sdn-networks/a/d-id/715577`.

[14] R. Pries, M. Jarschel, and S. Goll. On the usability of openflow in data center environments. In *Communications (ICC), 2012 IEEE International Conference on*, pages 5533–5537. IEEE, 2012.

[15] N. Ruchansky and D. Proserpio. A (not) nice way to verify the openflow switch specification: formal modelling of the openflow switch using alloy. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 527–528. ACM, 2013.

[16] R. Sherwood and K. Yap. Cbench controller benchmarker, 2010.

[17] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of NSDI*, volume 14, pages 87–99, 2014.