Network Softwarization: Course Project Spring 2019

Fault Tolerant Software Defined Network Using OpenDaylight on SAVI Testbed

Manush Nandan Mohankumar

Shivani Sivashankar

University of Toronto

Youtube Link -https://youtu.be/zbBcjT8sdxY

I. INTRODUCTION

This report is a combined result of a literature survey and an implementation project. The survey paper is based on the study of competent software defined network controllers available in the industry. We then discuss about the key performance indicators (KPI) which can reason their competence, and which are generally used to analyze a network. The survey gives an insight of various controllers and the comparison between them. As a result, it agrees with the existing statement that OpenDaylight (ODL) is one of the most widely used and efficient controllers.

The implementation part of the project, thus, employs OpenDaylight as the controller of a custom defined topology created on SAVI testbed. The topology involves the nodes positioned at different geographical locations within the country giving the network a realistic approach. The features of ODL, its graphic interface unit, API generating environment, parameter fields and statistics, features of clustering are exploited in the project to better understand it as a controller in a small, well-defined fish-topology and analyzing the statistics fetched from its various nodes and interfaces.

The implementation aims to create a more resilient and fail-safe network. Thus, the network is deliberately made to fail, and backup ways are created tested. Using a virtual controller called Virtual Tenant Network (VTN) is also used in case the backup fails as well. On a concluding note, we discuss about the future scope of the said technologies and tools in line with Lifecycle Service Orchestration (LSO) UniMgr and Open Network Automation Platform (ONAP) mentioning about the benefits of self-healing, orchestration and automation for such an implementation, scaled to real-life.

II. PROBLEM STATEMENT & MOTIVATION

SDN is becoming prevalent in the networking domain and OpenDaylight is an efficient controller used in academia and industry. With its many advantages, SDN still needs to be reliable and fault-tolerant with the huge amount of data and complex applications it must handle. SDN is also vulnerable because of the same reason why it is a strong and advantageous technical initiative: cloud-native architecture. On failure of network, this project intends to deal with the problem using an alternate way of performance providing alternate flows and analyzing the metrics while doing so. If the fault is due to the scenario where the flows disappear due to hard time, another feature called Virtual Tenant Network (VTN) is used.

III. SDN OVERVIEW

In this section, we will be discussing the elements comprising a Software Defined Networks based on the planes on which these elements are placed. The Fig 1 gives the architectural view of an SDN.

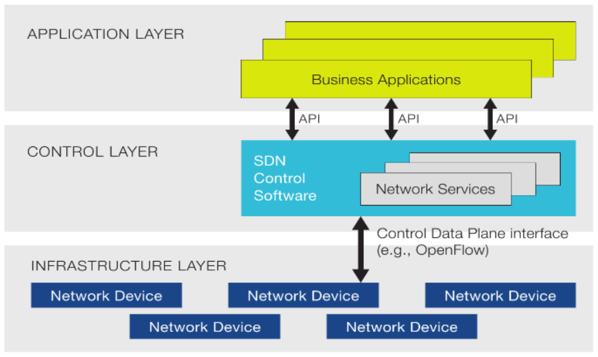


Fig 1 – SDN Architectural view [1]

The data plane comprises of switches, routers etc, which is responsible for the forwarding action based on the decision from the control plane which is the driving force in SDN. The decision is given by the centralised control logic (controller) which is considered to be the brain of the network. The network applications are available at the management layer or the application layer. They look after implementing control logic into the data plane by installing it. Also, the applications in order to implement these logics get the information of the underlaying infrastructure from the controller. The devices in data plane communicate with the controller by means of southbound interface. This interface installs flows in the devices of data plane and provides information of the devices in the data plane to the controller. The northbound interface provides the interaction between the controller and the application layer. However, there is no defined common protocol for the northbound interface as it was present in southbound interface.

IV. OPEN-SOURCE SDN CONTROLLERS

Open-source SDN controllers are controllers which are developed by the members of an open-source community. Each member will contribute and work together to achieve the goals set by the community to meet the SDN industry standards. Such, controllers can be deployed either as centralised where only one central server will manage the network or as distributed where more than one server work together to manage the network. The centralised controllers are simple and easy to manage but they have limited capacity since only one server is used and experiences scaling issues. On the other hand, distributed controller addresses this issue, but more effort must be put in order to manage it. So, based on the type of network the required deployment type is selected by the network architect. This section compares some of the famous open-source controllers and finds a controller which can compete with all the industry needs without leaving behind the performance [2].

4.1 Beacon Controller

Beacon controller was developed by Stanford University in association with Big Switch Network in 2010. The Beacon controller can be implemented in C#, Java and Python at the earlier levels. But C# was eliminated as an implementation language in Beacon due to the lack of official support for SDN based applications. Also, Python was eliminated later since it couldn’t support multi-threading in Beacon. Thus, Beacon is based on Java programming language. Since from it’s introduction, this open-source controller gained popularity in the research and academia. This controller used a similar OpenFlow 1.0 protocol named OpenFlowJ implemented using Java. The major objectives of this controller were to provide developer side productivity and high performance. The first objective was achieved by providing a rich set of libraries which eased the work of the developers. The second objective was achieved by providing OSGi specifications in a pool so that the developers can use any specification as per the need as well as the performance is increased by means of pipelining the messages in which a queue is generated which contains packets from all the elements, any worker thread when it’s free it would execute the next packet in the queue irrespective of the elements from which the packets arrived.

4.2 NOX and NOX-MT controller

NOX controller was a single thread first open source controller developed by Nicira networks in 2008 during the time when OpenFlow was first introduced. This controller was able to get the perception of the issue of the system-wide network and translate into a software issue. Also, this controller turned out to be a platform for developers and researchers to develop innovative network applications [4]. This controller was reintroduced in 2011 with the multi-thread functionality known as the NOX-MT. The reintroduction was a direct consequence to answer the capacity shortage of NOX controller. This controller provides optimization using I/O batching method and also stores the match between MAC tuple and port number of each number in a hash table so that the read operation goes through this table. Whenever, a new source is introduced the hash table needs to be updated with the inclusion of new MAC. However, NOX-MT couldn't solve the issues like excessive use of dynamic memory allocation and redundancy of multiple responses for a request of NOX controller.

4.3 Ryu Controller

Ryu is an open-source controller which contains the flexibility by handling the traffic tasks easily. The main advantage of Ryu controller is that, the components are customizable and are used for the development of Network Management and Control applications by means of Python. Also, since the components are customizable, they can modify according to the needs of the organization and can be deployed to control the elements of the network for a particular application. Though, Ryu comes with handy features it is good only for small scale network organizations and research. This is due to the fact that the modularity of Ryu is below average, and the performance drastically reduces when the number of nodes increases, brings the question of how stable this controller is to handle the advanced networks [5].

4.4 ONOS Controller

Open Networking Operating System (ONOS) is a java-based controller which is centralised logically but, physically distributed. Hence, this open-source controller comes under the category of distributed controller, since, it contains more than one server and the logic is implemented by more than one server unlike other controllers which have only one server hence, they are centralised [6]. This controller was developed to address the scalability, availability and performance issues found in centralised controller by providing high throughput of 1M requests and high availability of 99%. Also, this controller is based on 2 protocols. One protocol is used to provide a global network view by building a network architecture such that it provides fault tolerance and scalability features. The support of cache memory is used to reduce the time for repetitively used read operations. The polling issue which is mostly found in many centralised controllers are addressed in this controller by means of implementing a Hazelnut based pub-sub notification environment.

4.5 OpenDaylight Controller

OpenDaylight is a community-based controller under Linux Foundation collaborative projects. This controller was founded in 2013 but the first official release was in 2014. This controller was developed with aim of eradicating 'vendor locking' so that it can support many protocols other than OpenFlow. The main aim for the collaborative initiative was to develop a controller to provide a rich set of protocols, models and features to live up to industry needs in the field of SDN. This controller is a software application which is inspired by Beacon controller and runs as a JVM on systems which have the capability to support Java. The controller is designed using Model Driven Software Engineering (MDSE) by OMG. MDSE describes a framework in which the platform independent modules communicate with each other using data modelling language YANG. ODL uses NETCONF and RESTCONF as the model-driven network. ODL architecture contains northbound plugin and southbound plugins which are separated by means of using Service Abstraction Layer (SAL). The role of the northbound plugin is to manage the topology, flow and statistics of the underlaying network. The southbound plugin looks after the OpenFlow, NETCONF client and PCEP etc. This controller has good modularity compared to other controllers as well as has very good industrial collaboration.

V. COMPARISON OF THE CONTROLLERS

The controllers are compared with the latency performance on the basis of average response time for a single request [2]. The performance analysis in this publication was performed by using Cbench to evaluate the parameters irrespective of the number of switches that were used in the network. In latency mode of Cbench, the experiment was conducted in such a way that each switch involves in the execution of a single request followed by the next assuming the load to be low. The figure 2 shows the graph values for the latency in a single thread environment. The average response time in this graph is in microseconds.

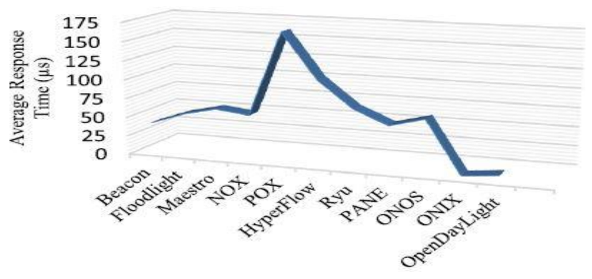


Fig 2 – Latency of controllers in a single thread environment [2]

From the figure, it is clear that ONIX, Beacon and OpenDaylight controllers have the lowest latency values compared to the other controllers. The values of latency in multi-threaded environment was also observed to have similar patterns as found in single-thread environment. Though, ONIX has the least latency value and has good feature set, it is not an open-sourced controller as like Beacon and OpenDaylight. Since, OpenDaylight is regarded to meet almost all SDN industry requirements by means of community-based industrial collaboration headed by Linux foundations, it is selected as a good-choice open-source controller for this project.

VI. OPENDAYLIGHT OVERVIEW

As discussed in section 4.5, OpenDaylight offers many modules and each module possess the capability of providing a particular service. The main interface of this controller is Service Abstraction Layer (SAL). Each module has been developed using the concept of SDN i.e multi-vendor sub-project development approach. The module providing the service can be used by the developer by installing the feature in the CLI of the ODL application by the command *feature:install [module_name]*. Developers using this controller also have the privilege of modifying the module to suit their network which shows the personalization and flexibility characteristics of this controller. All, the models created in ODL application are platform-independent and the service can be pushed into the platform by using the java interfaces [7].

6.1 Structural and Architectural Overview

Majority of these services are built on a layer called SAL, on top of which runs a provider-consumer model. Figure 3 shows the architecture of the OpenDaylight Oxygen release (fully equipped release at the time of doing this project) referenced from the OpenDaylight Oxygen release documentation [8].

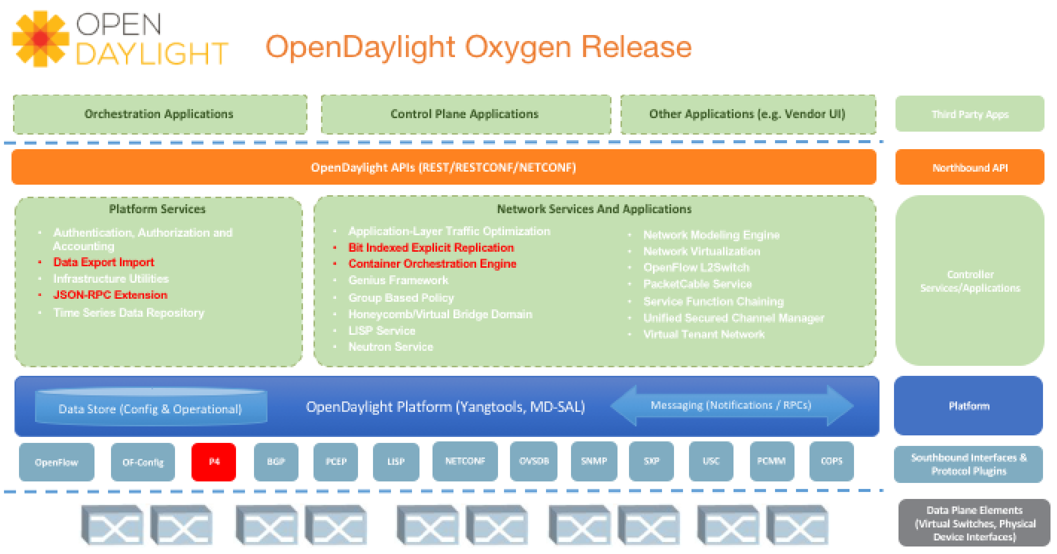

OPEN DAYLIGHT
OpenDaylight Oxygen Release
Orchestration Applications
Control Plane Applications
Other Applications (e.g. Vendor UI)
Third Party Apps
OpenDaylight APIs (REST/RESTCONF/NETCONF)
Northbound API
Platform Services
Authentication, Authorization and Accounting
Data Export Import
Infrastructure Utilities
JSON-RPC Extension
Time Series Data Repository
Network Services And Applications
Application-Layer Traffic Optimization
Bit Indexed Explicit Replication
Container Orchestration Engine
Genius Framework
Group Based Policy
Honeycomb/Virtual Bridge Domain
LISP Service
Neutron Service
Network Modeling Engine
Network Virtualization
OpenFlow L2Switch
PacketCable Service
Service Function Chaining
Unified Secured Channel Manager
Virtual Tenant Network
Controller Services/Applications
Data Store (Config & Operational)
OpenDaylight Platform (Yangtools, MD-SAL)
Messaging (Notifications / RPCs)
Platform
OpenFlow
OF-Config
P4
BGP
PCEP
LISP
NETCONF
OVSDB
SNMP
SXP
USC
PCMM
COPS
Southbound Interfaces & Protocol Plugins
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Fig 3 – OpenDaylight Oxygen Architecture [8]

The role of the Service Abstraction Layer (SAL) is to provide communication between the applications, models inside the controller and the underlaying devices. The ODL controller is designed using MDSE as said in section 4.5, which provides the relationship between modules and generates API/codes when the model is invoked for a service. This framework developed by Object Management Group (OMG) preferred to use YANG as a solution for unified modelling language [9].

The RESTCONF or REST is similar to RESTful like APIs which are used to post, delete or get any flows, statistics or any functional commands into the devices via controller using HTTP requests to the IP address of the controller listening to port 8181. The contents in this REST APIs can be html, xml or json. The SAL in an ODL controller can be of two types; Model-driven SAL (MDSAL) or API-driven SAL (ADSAL).

AD-SAL type is very useful when the developer can have direct access to the controller. Also, this type of SAL has dedicated APIs for northbound and southbound operations separately. The main characteristic of AD-SAL is that it can provide routing on request as well as service adaption by it’s APIs to a certain extent. However, it is stateless, limited only to flow capable devices and as reactive flow programming where every time the device needs to request the controller to receive any events.

MD-SAL type is helpful when the developer can’t have direct access to the controller but use REST APIs to establish a direct access. Also, MD-SAL uses one common REST API to perform both northbound and southbound operations. In contrast to AD-SAL, MD-SAL can only provide routing on request and service adaption is provided by plugins triggered using REST APIs. Also, it can store the data in a volatile or permanent APIs which means it has proactive flow programming where once the flows are pushed into the device until and unless only when the device don’t know what to do next will contact the controller. Since, may industrial developers find MD-SAL to have cutting edge over AD-SAL, the oxygen release emphases more on MD-SAL than the AD-SAL approach.

6.2 Available plan choices for ODL based SDN

An SDN based architecture is used in data centers or even in Network as a Service Provider where dynamic creation of virtual networks are possible if the need of scalability arises. So, ODL based SDN architecture must be selected based on a particular need. So, the following discussion is about different plan choices available for ODL based SDN. And, based on the need, the required plan choice is selected [7].

6.2.1 Physical and Logical centralised ODL controller

Centralised controller implementation is usually used in SDN because by this way the global view of network is obtained by the controller, but the way of implementing the controller can be either logically or physically centralised. In physically centralised controller, the capacity of the controller is fixed, since the capacity depends on the size of the server. But, if the demand surpasses the capacity, then the controller will experience a single point of failure and also scalability can’t be achieved in such instances. This drawback can be eradicated if the controller is logically centralised, which means that only the logic of the controller is centralised, and the physical servers can be distributed. In such a case, the physical servers are connected by means of east and westbound interfaces.

6.2.2 In-band and Out-of-band signalling

In this section, we will be discussing about the communication pattern between the controller and the forwarding devices in the data plane. In ODL based data center networks, the controller is constructed as physical device parallel to the data plane. But, if the networks are in different continents for example, then the separate physical controller won't be feasible in this case. So, two design options of controller channels are present

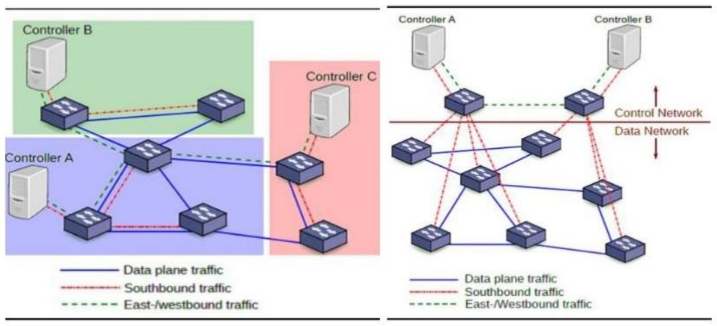


Fig 4 – In-band and Out-of-band signalling scenario [10]

In the In-band signalling ODL controller, the controller traffic is sent along with the data traffic, so it would be assumed to be data traffic which is the first disadvantage. Another disadvantage is that, when a data plane experiences a failure, the connected controller also experiences the failure. In the Out-of-band signalling, this drawback is removed, and control traffic is transmitted to another controller using east and west interfaces, so that control traffic doesn't go into the forwarding device. The figure below shows the in-band signalling scenario on the left and out-of-band signalling scenario on the right.

6.2.3 Proactive and Reactive flow management

In an ODL controller, the flows can be pushed into the forwarding device in the data plane either in a proactive manner or in a reactive manner. In the reactive approach, whenever, a packet reaches the forwarding device, it requests the controller for the action to be performed by it. The controller, having the global view of the network, responds with the action to be performed by the forwarding device. Though this approach requires less TCAM of the forwarding device, the disadvantage is that, whenever, a packet with new match condition arises, it should request the controller for the action to be performed which increases the round-trip time. On the other hand, in the proactive approach, all the flows are pushed into the forwarding device and this device forwards the packet based on these flows read from the TCAM. But, in a scenario were the number of flows is too high, the size of TCAM required will also be high, and since, the cost of TCAM is way too expensive it is not a good design model to go with approach. So, the best approach, would be to push the frequently used flows into the forwarding device so that the capacity of TCAM is not exceeded as well as the RTT would less than experienced in reactive approach.

6.2.4 Multiple Controllers

A reliable network must always ensure performance or at least quick revival in the case of failure. In order to prevent a single point of failure using a single controller, the concept of multiple controller design can be used. In this way, the above-mentioned drawback is prevented as well as more reliability can be achieved. But, two possible models can be designed. Either, all the controllers can be connected to all the devices or each controller can be connected to a set of devices in the data plane. If the design, consists of multiple controllers, then the ODL contains the clustering feature which makes one of the controllers as master while others as the slave and work in a synchronized fashion. In this way, if one controller fails, the other controller balances the load of the failed controller and helps in healing it. It also increases dependability on the network.

6.3 Virtual Tenant Network

Virtual Tenant Network (VTN) feature in OpenDaylight provides a possibility to create a multi-tenant network and has a good advantage of creating the multi-tenant network without the information of the underlaying infrastructure. Once, this feature is installed in the CLI of the controller, the model begins to abstract the required information of the underlaying infrastructure and map to create a VTN by means of configuring individual switches.

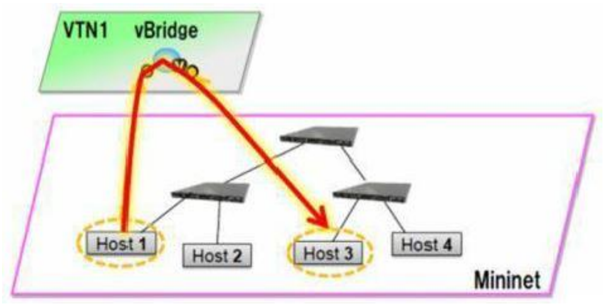


Fig 5 – VTN example using Mininet [11]

VTN doesn’t need additional requirements in a device in order to execute its functionality. VTN feature is generally executed by means of two components; VTN Manager and VTN Co-ordinator. VTN Manager is present in the control layer and abstracts the underlaying infrastructure information from the controller and send this information to the VTN Co-ordinator present in the application layer to virtualize a VTN to the user. The above figure shows a VTN example using mininet, in which a VTN of vBridge and vInterfaces between the two host h1 and h3 is created. This VTN resemblances the functionality of switches that are connected to both the hosts virtually.

VII. KEY PERFORMANCE INDICATORS IN SDN

The following section breezes through the available KPI that can be considered for the performance analysis of an SDN on all the levels. With competent IT service providers and cloud providers around, network virtualization and automation are crucial for Telco players. Apart from rebuilding the work model using the strategies of cloud providers and DevOps, virtualization also gives network providers the ability to enhance and manage their infrastructure better with improved agility and flexibility. It can reduce the operational and capital cost and significantly in the long run and also ensure high resiliency. It is possible to upgrade individual components of a virtualized network and thus make it easier to operate once the entire deployment is completed. Hence, virtualization is evidently the future of the telecommunication industry wherein, SDN and Network Function Virtualization (NFV) are key concepts.

An important point to note in a simple SDN is that, the controller is the sole point of contact with all control information and communication to all the components in the network. Consequently, it also becomes a single Point of Failure (PoF) in a given network. Thus, it is critical to monitor the controller and the performance of the network at the same time. The controllers communicate with the underlay while deploying, conducting the overlay network and exposing northbound Application Programming Interfaces (APIs) to program its network using a set of web services and applications. These APIs act as abstraction layers to the application and management systems that lie over the SDN technology stack. So, it is important to check if the Service Level Agreements (SLA) have been met along with the periodic health check of the controller.

Due to all these reasons, it is essential to define Key Performance Indicators (KPI) for networks which will imply the network realization and maintenance. This shall be in perfect alignment with Fault

Monitoring, Configuration, Accounting Performance and Security (FCAPS) model of network management. Considering these KPIs and how they impact the network is vital while deploying a automated network. In a traditional network, we use capacity planning, monitoring, troubleshooting, security etc. to assert the performance and we replicate this in a virtualized network as well.

These KPI include the attributes of the network and its components such as awareness, topology, configuration, fault tolerance, error detection and correction, optimization of cost, back-up routes and devices, QoS demands and service quality and finally, a well-planned well-known process leading to autonomy. Although introducing such aspects into the controller makes it way more complex, it reduces the probability of error to a greater extent [13]. Also, with autonomy comes decisiveness which means the programmers do not have to manually modify the controllers after the first installation.

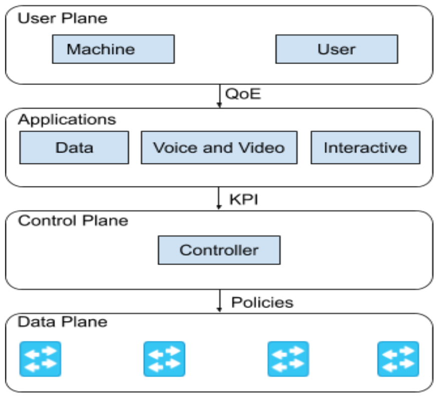


Fig 6 - Dependency of QoE and KPI on underlying policies

More recently network based QoE management has sought to apply the same principles to apply QoE. Many applications that would not fit within the traditional categorization of multimedia can still benefit from QoE based management. This paper presents a case study of a web-based mapping application to illustrate how a QoE management strategy can be used to adapt network parameters based on application *KPI/QoE* metrics. QoE is used to measure and express perception of the users. QoE is more than just a performance indicator service as it discovers the level of users satisfaction and their response to the service. While QoE is a user centric and depends on multiple impact factors such as network features, computing power, application-specific features and human cognition [12], the network perspective of the QoE can be mapped to measurable application KPIs. Using KPIs as proxies for QoE, minimum performance thresholds can be mapped to network metrics through a QoE model. This allows SDN to become application aware and dynamically adapt network parameters to the applications QoE needs.

7.1 List of Key Performance Indicators

Between now and reaching the goals of successful implementation of an SDN project, there are in fact evaluation and assessment stages for NFV/SDN technologies from three perspectives: interoperability, programmability, and security. Projects need to analyze the requirements and the KPIs of the platform by means of Use Cases. This is an effective but challenging approach, which is not trivial to apply to all the use cases in a similar way, due to potential and inevitable differences in the specific use cases. However, it has provided valuable indications on requirements for any further development. This bottom-up approach has also allowed collection of various Key Performance Indicators (KPIs). On a spherical view, KPIs can be calculated using a use case-specific combination of metrics. The metrics are based on every lamina of implementation and can be easily mapped to its corresponding OSI layer. Hence, although KPI can be done right from the physical layer onwards, on a higher-level overview, KPI can be bifurcated into the following:

Platform-related KPIs: These refer to the measures, functionalities, or qualities that have a general value, and can hence be reused independently from a specific application, and Application-specific KPIs: These refer to the measures, features, or qualities that have an essential meaning or importance during the execution of a specific application.

Each platform or testbed and application come with their own set of metrics that can be evaluated to decide KPI and thus analyze performance, e.g OpenStack has pre scripted codes to fetch and analyze it service daemons. The metrics used to decide on KPI generally used by developers and testers are briefed in the following section.

7.1.1 Operating System

The operating system that is typically used to host the virtual infrastructure manager OpenStack is Linux. The most typical KPIs monitored on Linux server machines can be listed as follows [14]

1) Average CPU usage – the average single CPU usage (time average of single CPU core) and the average aggregate CPU usage (time average of the aggregate over all CPU cores)
2) Maximum CPU usage – the maximum CPU usage observed; can be monitored for single and aggregate cases
3) Average drive space – the amount of storage used by the system and the running applications
4) Maximum drive space - the maximum amount of disk space allocated (although not a KPI as such, this indicates how much is space is available at a time, and enables to monitor any changes that might have diverse results)
5) Minimum free memory – the minimum available system memory
6) Maximum number of virtual machines – it indicates the limits in terms of possible instances of VMs on the platform (although not a KPI as such, this indicates how many VMs are allowed at a time, and enables to monitor any changes that might have diverse results)

7.1.2 CPU

1) CPU interrupt time: amount of time since the system was last started, in 100-nanosecond intervals. The interrupt-time count begins at zero when the system starts and is incremented at each clock interrupt by the length of a clock tick.
2) CPU idle time: the amount of time the CPU was not busy, or, otherwise, the amount of time it executed the System Idle process. Idle time actually measures unused CPU capacity.
3) CPU iowait time (IO wait): a sub-category of idle (%idle is usually expressed as all idle except defined subcategories), meaning the CPU is not doing anything. Therefore, as long as there is another process that the CPU could be processing, it will do so.
4) CPU nice time: time spent running processes with positive nice value (ie low priority). This means that it is consuming CPU but will give up that CPU time for most other processes.
5) CPU soft irq time (Soft IRQ): time typically used to complete queued work from a processed interrupt because they fit that need very well - they run with second-highest priority, but still run with hardware interrupts enabled

6) CPU steal time: the percentage of time a virtual CPU waits for a real CPU while the hypervisor is servicing another virtual processor. Your virtual machine (VM) shares resources with other instances on a single host in a virtualized environment.
7) CPU system time: amount of time for which a CPU was used for processing instructions of a computer program or operating system, as opposed to elapsed time, which includes for example, waiting for input/output (I/O) operations or entering low-power (idle) mode
8) CPU user time: time spent on the processor running your program's code (or code in libraries); system CPU time is the time spent running code in the OS kernel on behalf of your program

7.1.3 ICMP

1) ICMP loss
2) ICMP ping
3) ICMP response time

7.1.4 Memory

1) Available memory: refers to how much RAM is not already being used by the computer. Because loading the operating system takes up memory, your available memory drops right after your computer boots up.
2) Free swap space: Swap space is a common aspect of computing today, regardless of operating system. Linux uses swap space to increase the amount of virtual memory available to a host.
3) Total memory: The total amount of memory in a Linux computer is the RAM plus swap space and is referred to as virtual memory
4) Total swap space: memory to swap enough of relatively infrequently used pages of memory out to a special partition on the hard drive specifically designated for “paging,” or swapping
5) Incoming network traffic on network interfaces
6) Outgoing network traffic on network interfaces Operating System

7.1.5 Network [14]

1) Host or virtual machine boot time
2) Network availability: measures the “reachability” of one measurement point from another measurement point at the network layer (for example, using ICMP ping). The underlying routing and transport infrastructure of the provider network will support the availability measurements, with failures highlighted as unavailability or unreachability
3) Health: measures the number and type of errors that are occurring on the provider network, and can consist of both router-centric and network-centric measurements, such as hardware failures or packet loss
4) Utilization: Optimal utilization of resources like bandwidth, buffer window, etc.
5) Controller updation latency: On changing any aspect of the network, the controller needs to be informed and corresponding modifications must be made. The delay it this updation will cost the decision-making time in the network with a single controller.
6) Link up time: measures the time for a failed link to recover
7) Fail safety: On failure of a route, the recovery would either mean instantaneous link up in a more pragmatic situation, provide an alternate route while allowing least loss of packets. Parallelly it is also important that a fail-safe algorithm revives the initial path as it is presumably the best route performance-wise.

In this project, we shall use the network-based KPI to analyze our SDN using the tools supported by ODL controller.

VIII. IMPLEMENTATION RESULTS

We have implemented a custom SDN network with OpenDaylight controller in Smart Applications on Virtual Infrastructure (SAVI) test bed. SAVI test bed is build based on Openstack, and we decided to deploy our topology in this test bed using the credentials provided by University of Toronto, in order to get real time delays which, we didn’t get it using Mininet. We have divided the entire implementation into three stages. The first stage involves creation of topology and providing custom flows in the switches. The second stage deals with the creation of alternate flows in case of a failure and the third stage deals with the creation of Virtual Tenant Network (VTN).

8.1 Creation of customized OpenDaylight based SDN topology

The first part involves the creation of the OpenDaylight Oxygen-SR4 (latest fully equipped release) controller in one of the Virtual Machine with *Ubuntu-18-04* image and *m1.large* flavour in the *CORE* region of the SAVI test bed which can be seen in Figure 7. Since, this a private test bed and in order to communicate with the outside world, floating IPs are assigned to the controller along with the remote IP.

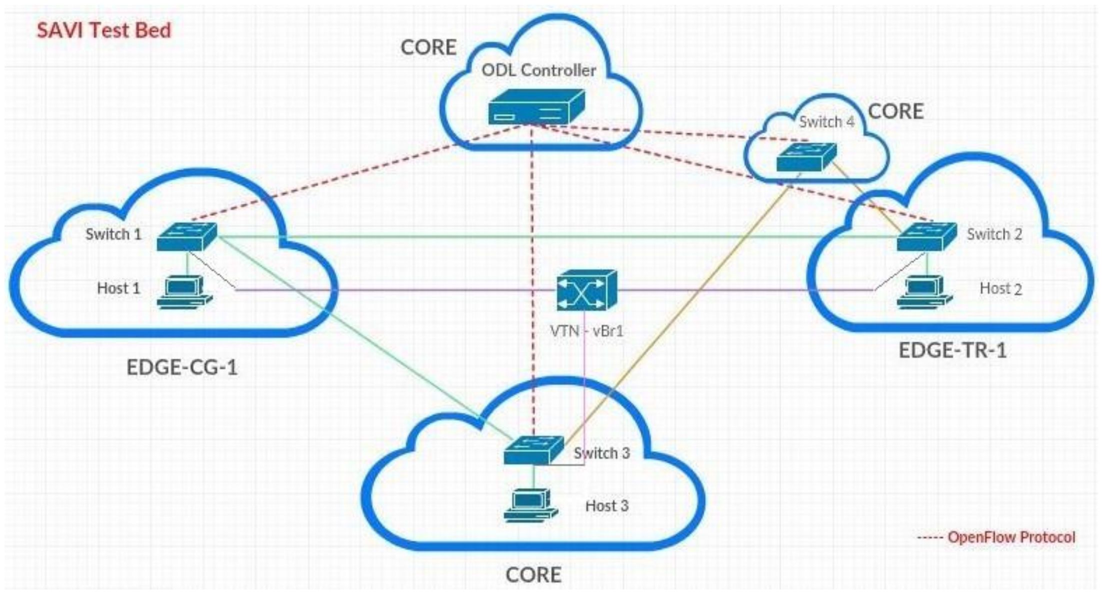
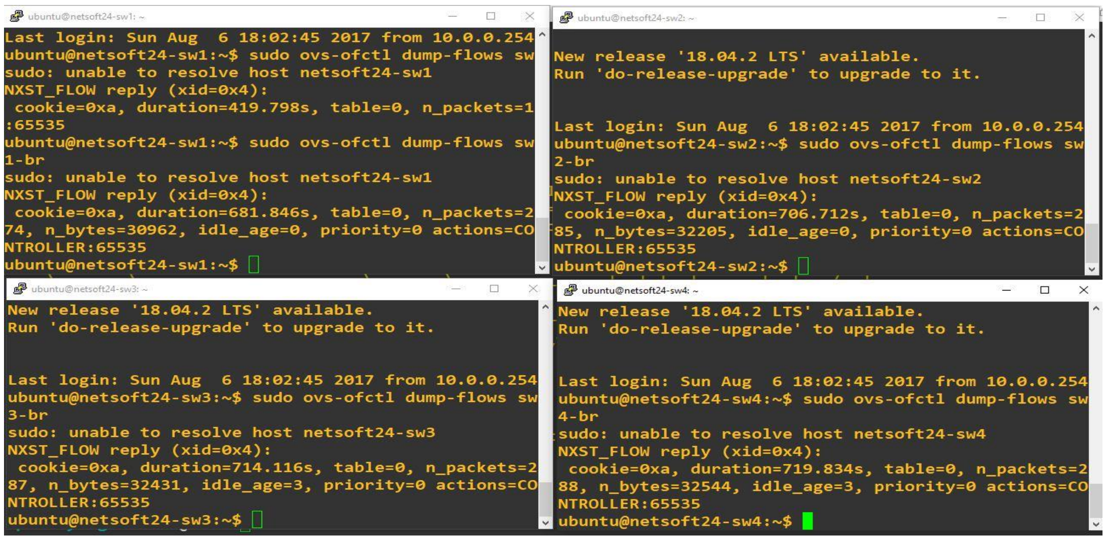


Fig 7 – Proposed Implementation in SAVI Test Bed

Once, the controller is deployed we ran topologyfinal.py python file in the terminal of SAVI, to create an overlay topology with 3 hosts and 4 switches. Each of these elements were deployed in a virtual machine using *ECE1508 overlay* image and *m1.small* flavour. A pair of a host and a switch were deployed in *CORE*, *EDGE-CG-1* and *EDGE-TR-1* regions of the SAVI test bed and another switch (switch 4) was deployed in the region *CORE*. Once, these virtual machines were deployed, bidirectional VXLAN interfaces were created between H1-SW1, SW1-SW3, SW1-SW2, SW3-H3, H2-SW2, SW2-SW4 and SW4-SW3. The Open vSwitch enabled switches are connected to the ODL controller using *OpenFlow 10* protocol. Once, this basic deployed is completed, the required features of OpenDaylight controller can be installed in the controller CLI using the command *feature:install [module_name]*.

Though, the controller provides the L2 switching feature, we have disabled it, since the VXLAN interface between the switches form a closed loop which would result in a condition where the hosts can't be pinged. Thus, the switches by default doesn't contain any flows except the flow rule which forwards the packet to the controller which can be seen in the figure 8.



```
Last login: Sun Aug  6 18:02:45 2017 from 10.0.0.254
ubuntu@netsoft24-sw1:~$ sudo ovs-ofctl dump-flows sw
sudo: unable to resolve host netsoft24-sw1
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=419.798s, table=0, n_packets=1
:65535
ubuntu@netsoft24-sw1:~$ sudo ovs-ofctl dump-flows sw
1-br
sudo: unable to resolve host netsoft24-sw1
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=681.846s, table=0, n_packets=2
74, n_bytes=30962, idle_age=0, priority=0 actions=CO
NTROLLER:65535
ubuntu@netsoft24-sw1:~$
```

```
New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Sun Aug  6 18:02:45 2017 from 10.0.0.254
ubuntu@netsoft24-sw2:~$ sudo ovs-ofctl dump-flows sw
2-br
sudo: unable to resolve host netsoft24-sw2
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=706.712s, table=0, n_packets=2
85, n_bytes=32205, idle_age=0, priority=0 actions=CO
NTROLLER:65535
ubuntu@netsoft24-sw2:~$
```

```
New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Sun Aug  6 18:02:45 2017 from 10.0.0.254
ubuntu@netsoft24-sw3:~$ sudo ovs-ofctl dump-flows sw
3-br
sudo: unable to resolve host netsoft24-sw3
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=714.116s, table=0, n_packets=2
87, n_bytes=32431, idle_age=3, priority=0 actions=CO
NTROLLER:65535
ubuntu@netsoft24-sw3:~$
```

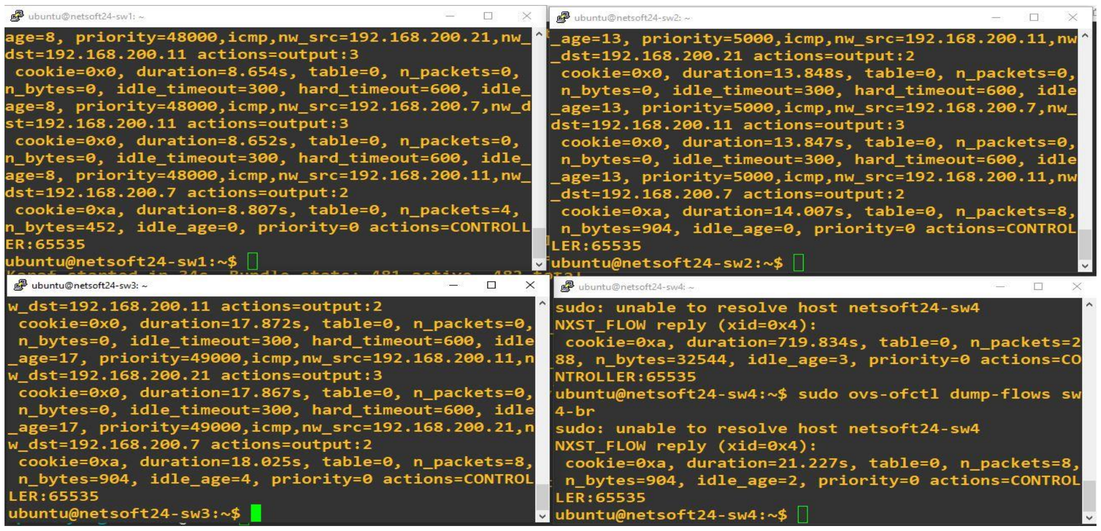
```
New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Sun Aug  6 18:02:45 2017 from 10.0.0.254
ubuntu@netsoft24-sw4:~$ sudo ovs-ofctl dump-flows sw
4-br
sudo: unable to resolve host netsoft24-sw4
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=719.834s, table=0, n_packets=2
88, n_bytes=32544, idle_age=3, priority=0 actions=CO
NTROLLER:65535
ubuntu@netsoft24-sw4:~$
```

Fig 8 – Switches with no L2 flow rules

But the switches need flow rules in order to forward the packets, but, in this case, if the flow rules are not present, the switch will still contact the controller and the controller can communicate with the REST APIs, in order to get the action to be performed by the switch. This is the Reactive flow approach, but the RTT experienced in this scenario would be higher than the Proactive approach, where the flows will be pushed into the switch by the controller using REST APIs. So, we selected the Proactive approach and pushed the flows using the *POSTMAN* application. However, the flows can also be pushed using *YANGMAN* application inside the GUI of the OpenDaylight Controller or by any programming language scripts. Since, the basic authorization of the OpenDaylight Controller is 'username: admin', 'password: admin', this is used as authorization for the communication between REST APIs and the controller. The content and accept format can be in JSON, XML or HTML and must be given as headers in the *POSTMAN* application. Since, we are pushing the flows, *PUT* along with the necessary URL is used to push the response. The acknowledgement of a successful push can be Status – 201 Created or 200 OK.

We have created our custom flows in such a way that all the packets pass only through SW1, SW2 and SW3. However, SW4 is used as an alternate path switch which will be used in the next stage. The successful flow rule creation is showed in the below figure.



```
age=8, priority=48000,icmp,nw_src=192.168.200.21,nw_
dst=192.168.200.11 actions=output:3
 cookie=0x0, duration=8.654s, table=0, n_packets=0,
n_bytes=0, idle_timeout=300, hard_timeout=600, idle_
age=8, priority=48000,icmp,nw_src=192.168.200.7,nw_d
st=192.168.200.11 actions=output:3
 cookie=0x0, duration=8.652s, table=0, n_packets=0,
n_bytes=0, idle_timeout=300, hard_timeout=600, idle_
age=8, priority=48000,icmp,nw_src=192.168.200.11,nw_
dst=192.168.200.7 actions=output:2
 cookie=0xa, duration=8.807s, table=0, n_packets=4,
n_bytes=452, idle_age=0, priority=0 actions=CONTROLL
ER:65535
ubuntu@netsoft24-sw1:~$
```

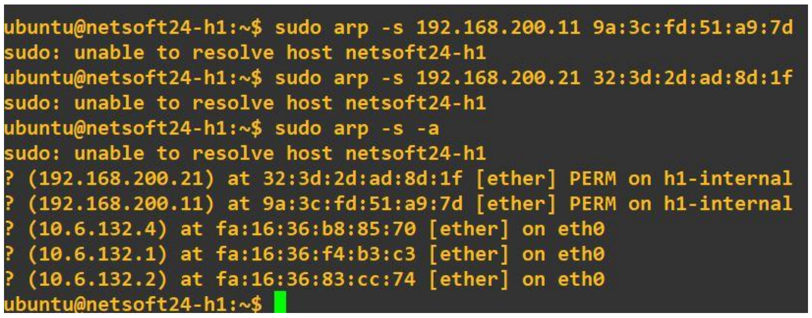
```
_age=13, priority=5000,icmp,nw_src=192.168.200.11,nw
_dst=192.168.200.21 actions=output:2
 cookie=0x0, duration=13.848s, table=0, n_packets=0,
 n_bytes=0, idle_timeout=300, hard_timeout=600, idle
_age=13, priority=5000,icmp,nw_src=192.168.200.7,nw_
dst=192.168.200.11 actions=output:3
 cookie=0x0, duration=13.847s, table=0, n_packets=0,
 n_bytes=0, idle_timeout=300, hard_timeout=600, idle
_age=13, priority=5000,icmp,nw_src=192.168.200.11,nw
_dst=192.168.200.7 actions=output:2
 cookie=0xa, duration=14.007s, table=0, n_packets=8,
 n_bytes=904, idle_age=0, priority=0 actions=CONTROL
LER:65535
ubuntu@netsoft24-sw2:~$
```

```
w_dst=192.168.200.11 actions=output:2
 cookie=0x0, duration=17.872s, table=0, n_packets=0,
 n_bytes=0, idle_timeout=300, hard_timeout=600, idle
_age=17, priority=49000,icmp,nw_src=192.168.200.11,n
w_dst=192.168.200.21 actions=output:3
 cookie=0x0, duration=17.867s, table=0, n_packets=0,
 n_bytes=0, idle_timeout=300, hard_timeout=600, idle
_age=17, priority=49000,icmp,nw_src=192.168.200.21,n
w_dst=192.168.200.7 actions=output:2
 cookie=0xa, duration=18.025s, table=0, n_packets=8,
 n_bytes=904, idle_age=4, priority=0 actions=CONTROL
LER:65535
ubuntu@netsoft24-sw3:~$
```

```
sudo: unable to resolve host netsoft24-sw4
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=719.834s, table=0, n_packets=2
88, n_bytes=32544, idle_age=3, priority=0 actions=CO
NTROLLER:65535
ubuntu@netsoft24-sw4:~$ sudo ovs-ofctl dump-flows sw
4-br
sudo: unable to resolve host netsoft24-sw4
NXST_FLOW reply (xid=0x4):
 cookie=0xa, duration=21.227s, table=0, n_packets=8,
 n_bytes=904, idle_age=2, priority=0 actions=CONTROL
LER:65535
ubuntu@netsoft24-sw4:~$
```

Fig 9 – Switch 1, 2, 3 with custom flow rules.

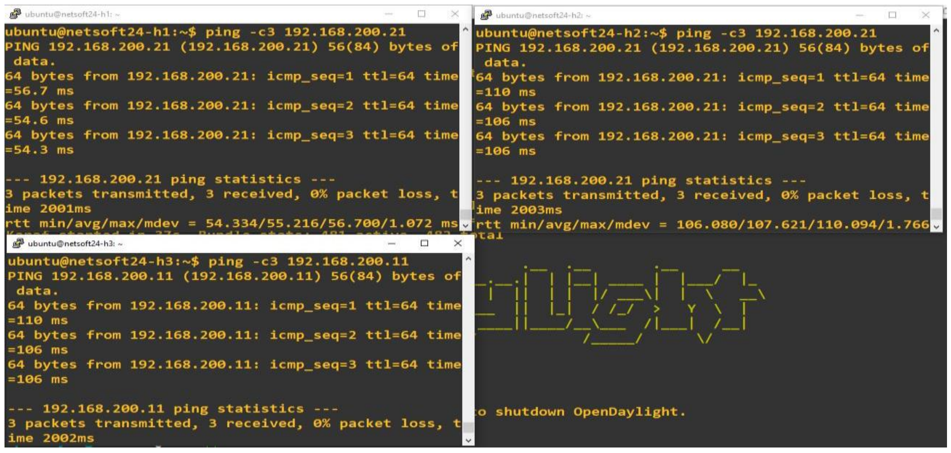
By L2 switching, usually the actions field of a switch will be *NORMAL*, where the switch will be sending packets and learning the destination MAC address simultaneously in order to send the packets properly to the destination. With custom flow rules, an additional path is created such that ARP is able to gather the MAC of the destination before sending the ICMP packets. The below figure shows the ARP status of H1 on H1-internal, where the H2 and H3 overlay IP and Internal MAC addresses are given. Similarly, it is done for H2 as well as H3.



```
ubuntu@netsoft24-h1:~$ sudo arp -s 192.168.200.11 9a:3c:fd:51:a9:7d
sudo: unable to resolve host netsoft24-h1
ubuntu@netsoft24-h1:~$ sudo arp -s 192.168.200.21 32:3d:2d:ad:8d:1f
sudo: unable to resolve host netsoft24-h1
ubuntu@netsoft24-h1:~$ sudo arp -s -a
sudo: unable to resolve host netsoft24-h1
? (192.168.200.21) at 32:3d:2d:ad:8d:1f [ether] PERM on h1-internal
? (192.168.200.11) at 9a:3c:fd:51:a9:7d [ether] PERM on h1-internal
? (10.6.132.4) at fa:16:36:b8:85:70 [ether] on eth0
? (10.6.132.1) at fa:16:36:f4:b3:c3 [ether] on eth0
? (10.6.132.2) at fa:16:36:83:cc:74 [ether] on eth0
ubuntu@netsoft24-h1:~$
```

Fig 10 – Host 1 ARP status

Once, the basic functionality initiation steps are completed, the hosts can ping each other. The below picture shows the successful ping of all three hosts with the OpenDaylight controller running in the background.



```
ubuntu@netsoft24-h1:~$ ping -c3 192.168.200.21
PING 192.168.200.21 (192.168.200.21) 56(84) bytes of
 data.
64 bytes from 192.168.200.21: icmp_seq=1 ttl=64 time
=56.7 ms
64 bytes from 192.168.200.21: icmp_seq=2 ttl=64 time
=54.6 ms
64 bytes from 192.168.200.21: icmp_seq=3 ttl=64 time
=54.3 ms

--- 192.168.200.21 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, t
ime 2001ms
rtt min/avg/max/mdev = 54.334/55.216/56.700/1.072 ms
```

```
ubuntu@netsoft24-h2:~$ ping -c3 192.168.200.21
PING 192.168.200.21 (192.168.200.21) 56(84) bytes of
 data.
64 bytes from 192.168.200.21: icmp_seq=1 ttl=64 time
=110 ms
64 bytes from 192.168.200.21: icmp_seq=2 ttl=64 time
=106 ms
64 bytes from 192.168.200.21: icmp_seq=3 ttl=64 time
=106 ms

--- 192.168.200.21 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, t
ime 2003ms
rtt min/avg/max/mdev = 106.080/107.621/110.094/1.766
```

```
ubuntu@netsoft24-h3:~$ ping -c3 192.168.200.11
PING 192.168.200.11 (192.168.200.11) 56(84) bytes of
 data.
64 bytes from 192.168.200.11: icmp_seq=1 ttl=64 time
=110 ms
64 bytes from 192.168.200.11: icmp_seq=2 ttl=64 time
=106 ms
64 bytes from 192.168.200.11: icmp_seq=3 ttl=64 time
=106 ms

--- 192.168.200.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, t
ime 2002ms
```

```
o shutdown OpenDaylight.
```

Fig 11 – Successful hosts ping results

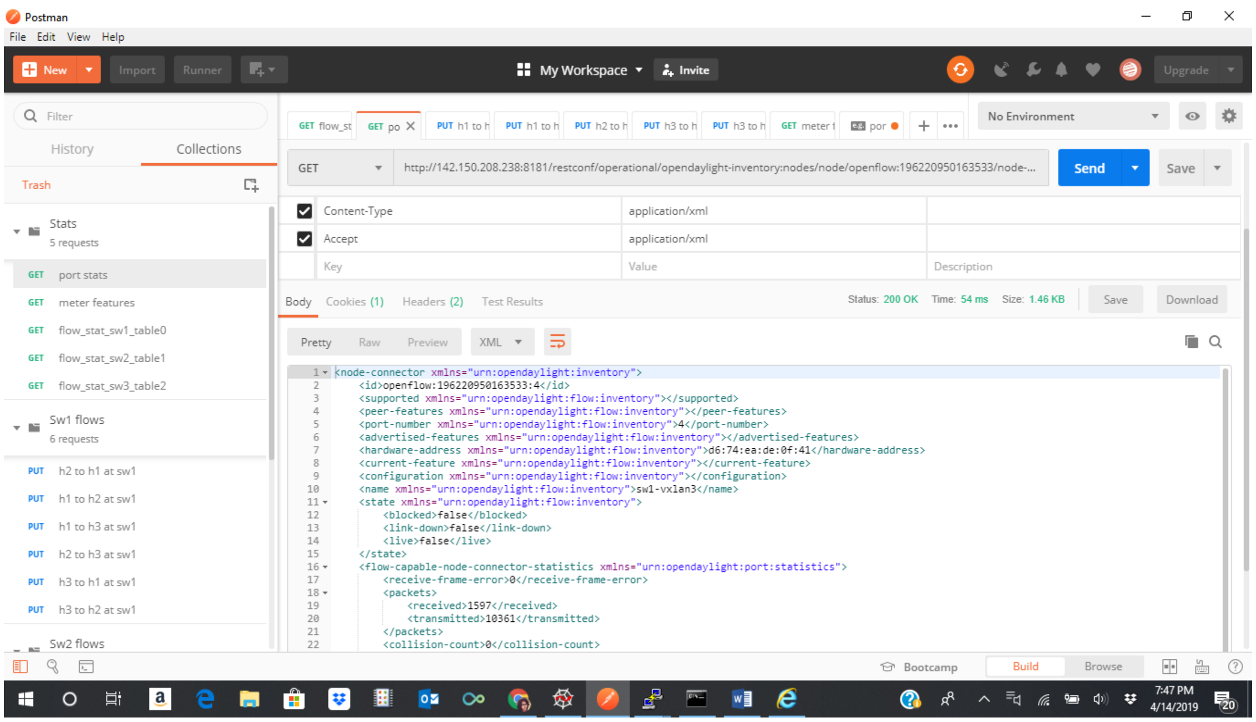
8.2 Obtaining metrics for analysis and inference

Above realized is a miniaturized implementation of an SDN. With ODL as controller, we can use its in-built GUI, DLUX to manage and visualize the topology creation. Each node along with their interfaces can be viewed on the localhost of the controller. Number of active flows, packets matched for QoS and packets looked up for forwarding are the flow statistics that can be found here. However, DLUX only has limited parameters it measures and does not give data that can be easily tracked and analyzed. Yangman, another feature of DLUX, is an API generator of ODL via which can fetch real-time statistics from the

VMs. The drawback of Yangman is the fact that it does not include any fields for authorization, adjustment of content type or filters.

Another tool that could be used is Postman which was the external API generator environment used to push the initial flows into the OVS switches. Postman is the solution to all the drawbacks of the aforementioned tools of DLUX. It can return response of a command in JSON, XML, HTML or TXT formats and includes authorization fields which is necessary in ratified environments like ODL or SAVI testbed. We can obtain numerous metrics including port numbers, number of received and transmitted packets and bytes, number of erroneous frames, collision count, number of packet drops, link state at the particular interface, port status, CRC errors, link status, port ids etc., apart from the flow statistics which could be obtained from DLUX. We can also include certain meters for checking if QoS thresholds are met and check their statistics.

A snippet of the Postman interface below shows the output response obtained for port statistics metrics for SW1 on VXLAN3, which is connected to SW2, collection of flow rules and content-type filtering (Fig 12).


GET http://142.150.208.238:8181/restconf/operational/opendaylight-inventory:nodes/node/openflow:196220950163533/node-...

	Key	Value	Description
✓	Content-Type	application/xml	
✓	Accept	application/xml	

Status: 200 OK Time: 54 ms Size: 1.46 KB

Stats (5 requests): GET port stats, GET meter features, GET flow_stat_sw1_table0, GET flow_stat_sw2_table1, GET flow_stat_sw3_table2

Sw1 flows (6 requests): PUT h2 to h1 at sw1, PUT h1 to h2 at sw1, PUT h1 to h3 at sw1, PUT h2 to h3 at sw1, PUT h3 to h1 at sw1, PUT h3 to h2 at sw1

Sw2 flows

```
<node-connector xmlns="urn:opendaylight:inventory">
    <id>openflow:196220950163533:4</id>
    <supported xmlns="urn:opendaylight:flow:inventory"></supported>
    <peer-features xmlns="urn:opendaylight:flow:inventory"></peer-features>
    <port-number xmlns="urn:opendaylight:flow:inventory">4</port-number>
    <advertised-features xmlns="urn:opendaylight:flow:inventory"></advertised-features>
    <hardware-address xmlns="urn:opendaylight:flow:inventory">d6:74:ea:de:0f:41</hardware-address>
    <current-feature xmlns="urn:opendaylight:flow:inventory"></current-feature>
    <configuration xmlns="urn:opendaylight:flow:inventory"></configuration>
    <name xmlns="urn:opendaylight:flow:inventory">sw1-vxlan3</name>
    <state xmlns="urn:opendaylight:flow:inventory">
        <blocked>false</blocked>
        <link-down>false</link-down>
        <live>false</live>
    </state>
    <flow-capable-node-connector-statistics xmlns="urn:opendaylight:port:statistics">
        <receive-frame-error>0</receive-frame-error>
        <packets>
            <received>1597</received>
            <transmitted>10361</transmitted>
        </packets>
        <collision-count>0</collision-count>
```

Fig 12 - Snippet of Postman interface to obtain statistics

8.2.1 RestAPI to collect statistics

In order to capture, plot and analyze data obtained in the metrics, we have used RestAPI scripted in Python. In scenarios where there are port failures (enforced in our case using 'noforward' command), there is a consequent packet loss as we can expect. We use the number of transceived packets to check for ratio of loss. While we run script txPkts_sw3.py we can see using the animation tool of Python that even on the port failure in the first case, SW3 continues to transmit data with changed ports due to the new rules injected into it for fail-safety. In **Fig**. we can notice the regular increase in the packets from A to B with stagnation from B to C where the packets were not transmitted using this interface. On reviving back to 'forward', the port is up and starts forwarding again. However, this time the packets are throttled to a slow-start which again increases as the number of transmitted packets add up.

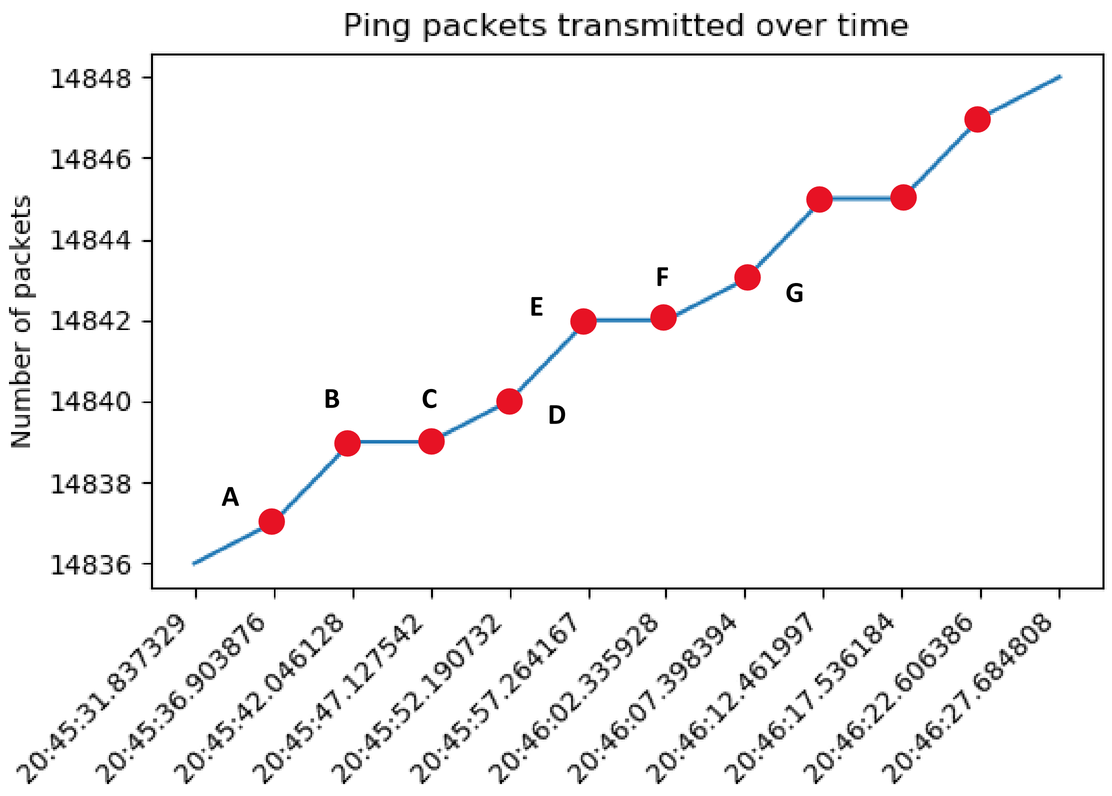


Fig 13 - Transmitted packets on SW3 port 2

We can thus infer that the topology, flows and the fail-safety procedure work the way we intended. To make the data more reliable, we have run the ping commands in a set of 50 to 10,000. Assuming that in a real network, a failure happens only once to be noticed and fixed, we introduced the port down and revived it at a random time once in every run.

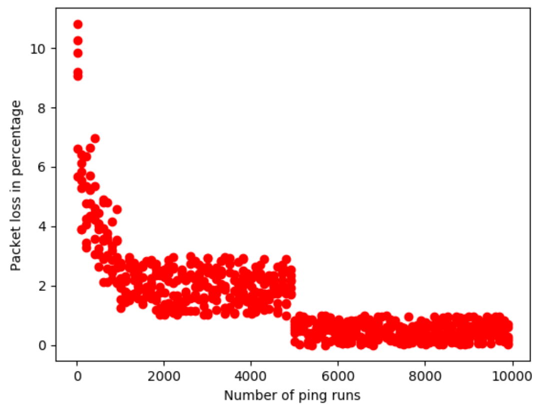


Fig 14 - Percentage of packet loss with increasing number of Ping runs

8.3 Making the custom network Fault Tolerant

This part consists of deliberately making port 2 of Switch 3 “No Forward”, such that the VXLAN interface between SW1 and SW3 can’t be used and hence no host can’t ping any other host. This condition is visualized by the following figure.

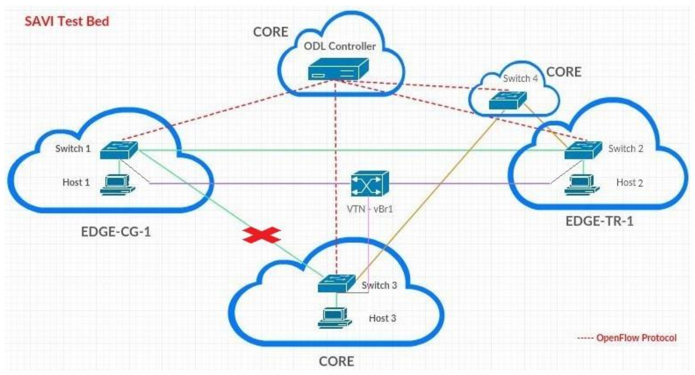
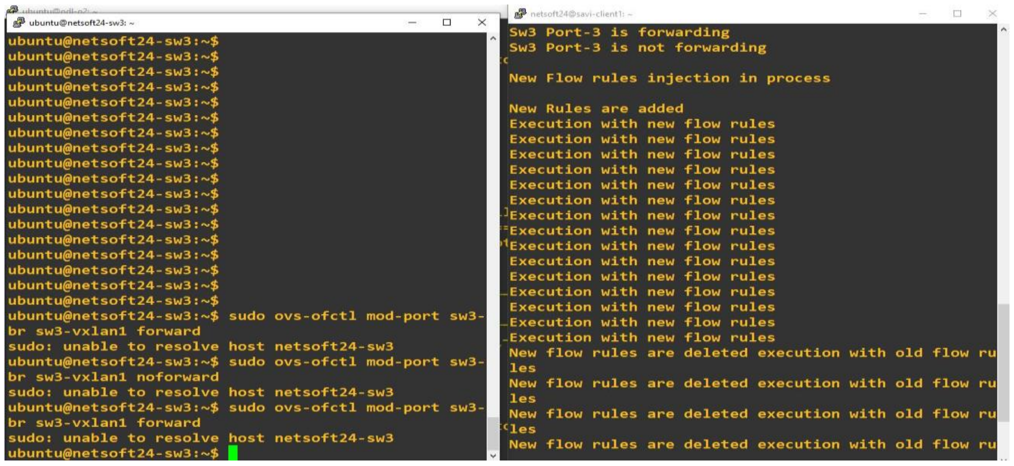


Fig 15 – Fault condition in the topology

The red cross mark in the VXLAN interface between SW1 and SW3 represents that no packets are transmitted through this link during the *NO-FWD* fault condition. However, *restapi.py* python code will be running in the background and fetching the port 2 Switch 3 statistics which can found in the below figure. If the condition *NO-FWD* arises, the alternate routes via Switch 4 is pushed into all the switches with priority higher than the previous rule. The statistics of port 2 is checked every second and if it comes back to the normal state, the new rules which are injected are removed and the ping continues with the old rules.


```
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$
ubuntu@netsoft24-sw3:~$ sudo ovs-ofctl mod-port sw3-
br sw3-vxlan1 forward
sudo: unable to resolve host netsoft24-sw3
ubuntu@netsoft24-sw3:~$ sudo ovs-ofctl mod-port sw3-
br sw3-vxlan1 noforward
sudo: unable to resolve host netsoft24-sw3
ubuntu@netsoft24-sw3:~$ sudo ovs-ofctl mod-port sw3-
br sw3-vxlan1 forward
sudo: unable to resolve host netsoft24-sw3
ubuntu@netsoft24-sw3:~$
```

```
Sw3 Port-3 is forwarding
Sw3 Port-3 is not forwarding

New Flow rules injection in process

New Rules are added
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
Execution with new flow rules
New flow rules are deleted execution with old flow ru
les
New flow rules are deleted execution with old flow ru
les
New flow rules are deleted execution with old flow ru
les
New flow rules are deleted execution with old flow ru
```

Fig 16 – SW-3 Port-2 Statistics and Code generating alternate flows

The following image shows the new flow rules injected into switch 3 with higher priority, such that packets are sent out or received through port 4 and not port 2. Also, it can be clearly, seen that due to the higher priority, the new flow rule with same condition as that of old flow is above it.

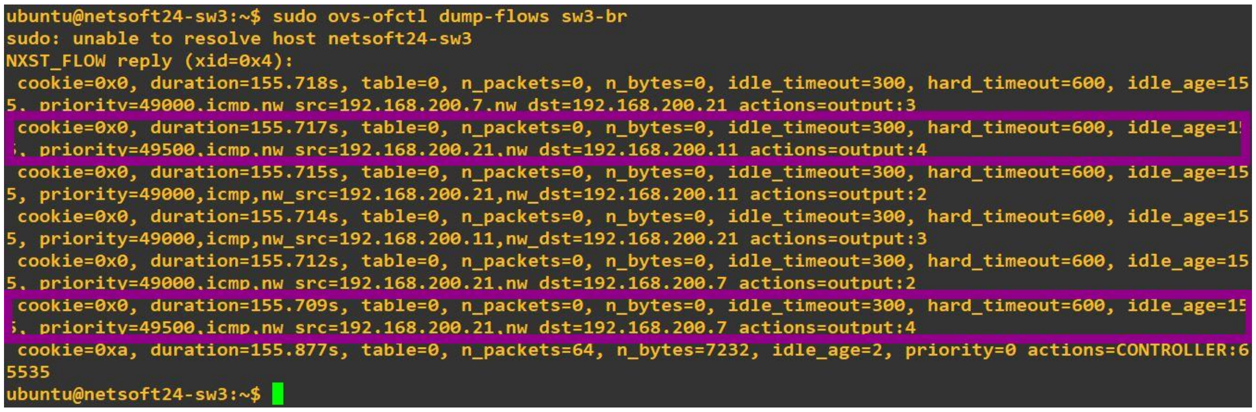
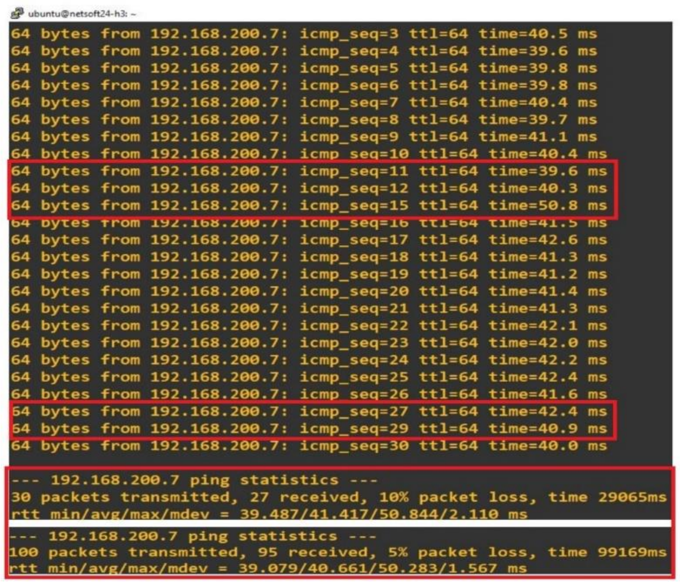

```
ubuntu@netsoft24-sw3:~$ sudo ovs-ofctl dump-flows sw3-br
sudo: unable to resolve host netsoft24-sw3
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=155.718s, table=0, n_packets=0, n_bytes=0, idle_timeout=300, hard_timeout=600, idle_age=15
5, priority=49000,icmp,nw_src=192.168.200.7,nw_dst=192.168.200.21 actions=output:3
 cookie=0x0, duration=155.717s, table=0, n_packets=0, n_bytes=0, idle_timeout=300, hard_timeout=600, idle_age=1
, priority=49500,icmp,nw_src=192.168.200.21,nw_dst=192.168.200.11 actions=output:4
 cookie=0x0, duration=155.715s, table=0, n_packets=0, n_bytes=0, idle_timeout=300, hard_timeout=600, idle_age=15
5, priority=49000,icmp,nw_src=192.168.200.21,nw_dst=192.168.200.11 actions=output:2
 cookie=0x0, duration=155.714s, table=0, n_packets=0, n_bytes=0, idle_timeout=300, hard_timeout=600, idle_age=15
5, priority=49000,icmp,nw_src=192.168.200.11,nw_dst=192.168.200.21 actions=output:3
 cookie=0x0, duration=155.712s, table=0, n_packets=0, n_bytes=0, idle_timeout=300, hard_timeout=600, idle_age=15
5, priority=49000,icmp,nw_src=192.168.200.21,nw_dst=192.168.200.7 actions=output:2
 cookie=0x0, duration=155.709s, table=0, n_packets=0, n_bytes=0, idle_timeout=300, hard_timeout=600, idle_age=15
, priority=49500,icmp,nw_src=192.168.200.21,nw_dst=192.168.200.7 actions=output:4
 cookie=0xa, duration=155.877s, table=0, n_packets=64, n_bytes=7232, idle_age=2, priority=0 actions=CONTROLLER:6
5535
ubuntu@netsoft24-sw3:~$
```

Fig 17 – SW-3 Flow-Table



```
ubuntu@netsoft24-h3: ~
64 bytes from 192.168.200.7: icmp_seq=3 ttl=64 time=40.5 ms
64 bytes from 192.168.200.7: icmp_seq=4 ttl=64 time=39.6 ms
64 bytes from 192.168.200.7: icmp_seq=5 ttl=64 time=39.8 ms
64 bytes from 192.168.200.7: icmp_seq=6 ttl=64 time=39.8 ms
64 bytes from 192.168.200.7: icmp_seq=7 ttl=64 time=40.4 ms
64 bytes from 192.168.200.7: icmp_seq=8 ttl=64 time=39.7 ms
64 bytes from 192.168.200.7: icmp_seq=9 ttl=64 time=41.1 ms
64 bytes from 192.168.200.7: icmp_seq=10 ttl=64 time=40.4 ms
64 bytes from 192.168.200.7: icmp_seq=11 ttl=64 time=39.6 ms
64 bytes from 192.168.200.7: icmp_seq=12 ttl=64 time=40.3 ms
64 bytes from 192.168.200.7: icmp_seq=15 ttl=64 time=50.8 ms
64 bytes from 192.168.200.7: icmp_seq=16 ttl=64 time=41.5 ms
64 bytes from 192.168.200.7: icmp_seq=17 ttl=64 time=42.6 ms
64 bytes from 192.168.200.7: icmp_seq=18 ttl=64 time=41.3 ms
64 bytes from 192.168.200.7: icmp_seq=19 ttl=64 time=41.2 ms
64 bytes from 192.168.200.7: icmp_seq=20 ttl=64 time=41.4 ms
64 bytes from 192.168.200.7: icmp_seq=21 ttl=64 time=41.3 ms
64 bytes from 192.168.200.7: icmp_seq=22 ttl=64 time=42.1 ms
64 bytes from 192.168.200.7: icmp_seq=23 ttl=64 time=42.0 ms
64 bytes from 192.168.200.7: icmp_seq=24 ttl=64 time=42.2 ms
64 bytes from 192.168.200.7: icmp_seq=25 ttl=64 time=42.4 ms
64 bytes from 192.168.200.7: icmp_seq=26 ttl=64 time=41.6 ms
64 bytes from 192.168.200.7: icmp_seq=27 ttl=64 time=42.4 ms
64 bytes from 192.168.200.7: icmp_seq=29 ttl=64 time=40.9 ms
64 bytes from 192.168.200.7: icmp_seq=30 ttl=64 time=40.0 ms

--- 192.168.200.7 ping statistics ---
30 packets transmitted, 27 received, 10% packet loss, time 29065ms
rtt min/avg/max/mdev = 39.487/41.417/50.844/2.110 ms

--- 192.168.200.7 ping statistics ---
100 packets transmitted, 95 received, 5% packet loss, time 99169ms
rtt min/avg/max/mdev = 39.079/40.661/50.283/1.567 ms
```

Fig 18 – H3 to H1 ping during fault occurrence and recovery

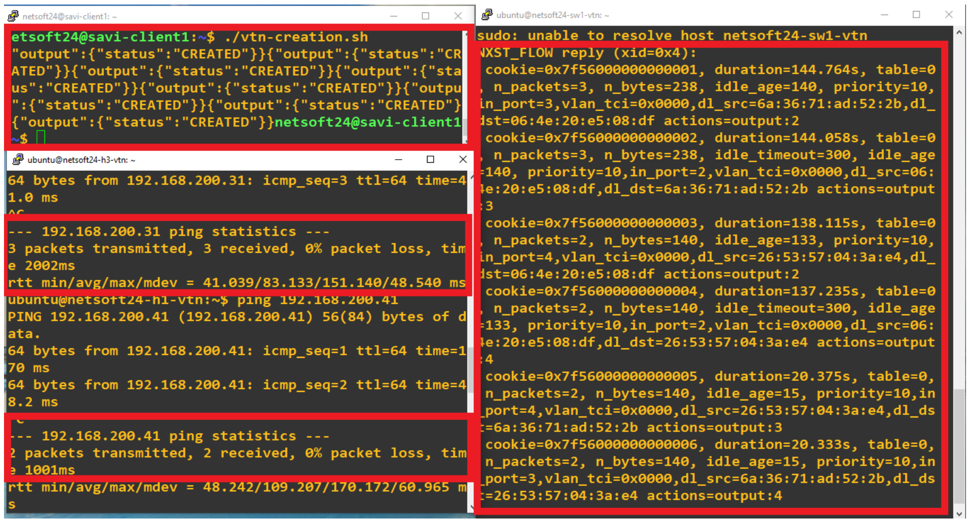
The above image shows the ping between h3 and h1, where the first few sequences are transmitted/received using port 2 and once when that port is made NO-FWD, the successive sequences are transmitted/received through port 4. When, the fault is fixed, the old rules are retained and new rules are deleted since, the first set of rules provide the shortest path, and the alternate set of rules provides the second shortest path in this case. However, since the modification of flow rules are taking place on the fly, 2-3 packets are dropped during each transition. A detailed analysis regarding this, will be discussed in the following parts.

8.4 Virtual Tenant Network for increased resilience

By providing the custom flows to any switches, the developer can provide value for the hard-time out. In that case, once the topology exceeds the value of hard time-out then there is a possibility that the flow rule may have been removed from the switch which can cause a fault in the system and the hosts can't transmit or receive packets since, the deletion of one flow rule may affect the ping. In that case, OpenDaylight controller provides a feature named Virtual Tenant Network (VTN), which creates a virtual tenant network by abstracting the information of the underlaying infrastructure and thus forming a logical abstraction plane.

In our custom topology, to show functioning of this scenario we have modified it, by disabling switch 4. Since, the VTN provides a L2/L3 switching, elimination of switch 4 rules out the possibility of closed loop. Also, we injected the flows with short hard-timeout, so that the flow rules disappeared. However, when the controller is restarted the flow-rules appear. But this is not the case that takes place in an active network. Once, the flow rules are disappeared, we run the *vtn-creation.sh* bash script, to create the Virtual Tenant Network which can be seen in the figure 6. This script uses curl commands to create a VTN named vtn1, then a vBridge named vbr1 and three virtual interfaces named if1, if2 and if3 which are connected to the ports of the switch 1, switch 2 and switch where it is connected to the hosts by VXLAN interface. Once, this VTN is created, a virtual bridge and VLAN interfaces are created between three hosts. This vbr1 maps to the underlaying switches 1, 2, 3 and virtualizes as single switch with the mapping knowledge of all the hosts and uses L2 switching without any custom flow rules.

In the below figure, the top left window shows the successful creation of the VTN, the bottom left window shows the successful ping between h1 and h3 and h1 and h2. The right window shows the flow rule created in switch 1 by the VTN. From these flow rules, it is evident that VTN is able to map the switches and abstract the MAC address of three hosts such that, it can provide the rules with source and destination MAC matching. The VTN feature can be modified by providing specific policies, VLAN, MAC mapping and can also be used in service functions chaining.


netsoft24@savi-client1: ~
ubuntu@netsoft24-h3-vtn: ~
ubuntu@netsoft24-sw1-vtn: ~

```
etsoft24@savi-client1:~$ ./vtn-creation.sh
"output":{"status":"CREATED"}}{"output":{"status":"CR
ATED"}}{"output":{"status":"CREATED"}}{"output":{"sta
us":"CREATED"}}{"output":{"status":"CREATED"}}{"outpu
":{"status":"CREATED"}}{"output":{"status":"CREATED"}
{"output":{"status":"CREATED"}}netsoft24@savi-client1
~$
```

```
64 bytes from 192.168.200.31: icmp_seq=3 ttl=64 time=4
1.0 ms
^C
--- 192.168.200.31 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, tim
e 2002ms
rtt min/avg/max/mdev = 41.039/83.133/151.140/48.540 ms
ubuntu@netsoft24-h1-vtn:~$ ping 192.168.200.41
PING 192.168.200.41 (192.168.200.41) 56(84) bytes of d
ata.
64 bytes from 192.168.200.41: icmp_seq=1 ttl=64 time=1
70 ms
64 bytes from 192.168.200.41: icmp_seq=2 ttl=64 time=4
8.2 ms
^C
--- 192.168.200.41 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, tim
e 1001ms
rtt min/avg/max/mdev = 48.242/109.207/170.172/60.965 m
s
```

```
sudo: unable to resolve host netsoft24-sw1-vtn
NXST_FLOW reply (xid=0x4):
 cookie=0x7f56000000000001, duration=144.764s, table=0
, n_packets=3, n_bytes=238, idle_age=140, priority=10,
in_port=3,vlan_tci=0x0000,dl_src=6a:36:71:ad:52:2b,dl_
dst=06:4e:20:e5:08:df actions=output:2
 cookie=0x7f56000000000002, duration=144.058s, table=0
, n_packets=3, n_bytes=238, idle_timeout=300, idle_age
=140, priority=10,in_port=2,vlan_tci=0x0000,dl_src=06:
4e:20:e5:08:df,dl_dst=6a:36:71:ad:52:2b actions=output
:3
 cookie=0x7f56000000000003, duration=138.115s, table=0
, n_packets=2, n_bytes=140, idle_age=133, priority=10,
in_port=4,vlan_tci=0x0000,dl_src=26:53:57:04:3a:e4,dl_
dst=06:4e:20:e5:08:df actions=output:2
 cookie=0x7f56000000000004, duration=137.235s, table=0
, n_packets=2, n_bytes=140, idle_timeout=300, idle_age
=133, priority=10,in_port=2,vlan_tci=0x0000,dl_src=06:
4e:20:e5:08:df,dl_dst=26:53:57:04:3a:e4 actions=output
:4
 cookie=0x7f56000000000005, duration=20.375s, table=0,
 n_packets=2, n_bytes=140, idle_age=15, priority=10,in
_port=4,vlan_tci=0x0000,dl_src=26:53:57:04:3a:e4,dl_ds
t=6a:36:71:ad:52:2b actions=output:3
 cookie=0x7f56000000000006, duration=20.333s, table=0,
 n_packets=2, n_bytes=140, idle_age=15, priority=10,in
_port=3,vlan_tci=0x0000,dl_src=6a:36:71:ad:52:2b,dl_ds
t=26:53:57:04:3a:e4 actions=output:4
```

Fig 19 – VTN for the proposed topology

IX. CONCLUSION & FUTURE REMARKS

9.1 Difficulties faced

We decided to deploy our topology in SAVI testbed in order to get a realistic approach spanning geographical areas. However, Savi testbed is built around OpenStack and there is not a lot of resources available integrating ODL with Openstack interface which involves custom topology. Since only the members of this project under Linux foundation had access to key resourceful articles for the use of OpenDaylight in environments other than Mininet, we faced difficulties in the initial process of deploying and achieving the objectives. However, we were able to successfully achieve the requirements with constant trial and error methods. The other problem faced was that, we could send only ICMP packets to analyze the network as it became difficult to send other data formats. Image frames could not be viewed even if transmitted over the server we used on SAVI as it was headless. The applications to transfer audio data could not be run on Putty. Hence, the QoS modifications could not be performed on time to have a larger collection of analytics.

9.2 Conclusion

Thus, in this report, we have surveyed the popular open-source controllers that are present in the industry and preferred OpenDaylight as a good-choice based on the outputs considering latency for a single thread system. We have discussed, the available plan choices in deploying an OpenDaylight based SDN followed by analyzing metrics on various layers of the OSI model which are used in the industry to evaluate competence of a software-defined network. Then, we implemented our proposed custom topology in SAVI testbed with custom flow rules and checked the ping between the three hosts. In order to check the fault tolerance of this network, a fault condition was created, and the custom network was checked, if it could survive the fault condition and still ping the ICMP packets. A simultaneous analysis was also performed with the available metrics including port and flow statistics. Finally, we checked if the custom topology could deploy a VTN in case of flow rules disappearing due to hard time out. Thus, to conclude, we were able to successfully prove that the custom topology deployed in SAVI testbed using OpenDaylight is fault tolerant, resilient and reliable.

9.3 Future Remarks

Virtual Tenant Network can be modified such that more specific flow rules can be obtained by matching of MAC address, VLAN IDs and providing policies. Also, Virtual Tenant Network can be used when service chaining functions has to be deployed. Therefore, in future, we believe that using these functionalities would help in resolving the scaling issues of the single controller or failure of a cluster of controllers. Also, with the need of scalability comes the requirement of orchestration which would be helpful in automatically scaling the with self-healing capabilities. This orchestration is much required in the User Network Interface. Thus, with OpenDaylight UniMgr feature and MEF LSO Legato and Presto APIs, this orchestration can be achieved. Also, with the huge amount of data, exponentially rising number of plugIns and increasing complexity of the applications that SDN is supposed to handle, automation will become a necessity. Platforms like ONAP can aid the efficient assessment and allocation of resources in an optimized fashion.

REFERENCES

[1] SDX Central. Understanding the SDN Architecture [online]. Sunnyvale, CA: SDxCentral. URL: https://www.sdxcentral.com/resources/sdn/inside-sdn-archi-tecture/

[2] M. Paliwal et al., ‘Controllers in SDN: A Review Report,’ IEEE Access, Digital Object Identifier 10.1109/ACCESS.2018.2846236Digital Object Identifier 10.1109/ACCESS.2018.2846236, June 2018.

[3] Open Service Gateway Initiative. Accessed: Nov. 11, 2017. [Online]. Available: https://www.osgi.org/

[4] S. Raju., ‘SDN Controllers Comparison’, Proceedings of Science Globe International Conference, 10th June, 2018, Bengaluru, India

[5] A. Saleh et al., ‘Ryu controller’s scalability experiment on software defined networks’, IEEE Intl Conference on Current Trends in Advanced Computing, Feb 2018.

[6] P. Berde, ``ONOS: Towards an open, distributed SDN OS,'' in Proc. 3rd Workshop Hot Topics Software Defined Networks, 2014, pp. 1-6

[7] Suad El-Geder, Ph.D Thesis on ‘Performance Evaluation using Multiple Controllers with Different Flow Setup Modes in the Software Defined Network Architecture’, Brunel University London, January 2017.

[8] OpenDaylight Oxygen release note https://www.opendaylight.org/what-we-do/current-release/oxygen

[9] Maksim Sisov, BE Thesis on ‘Building a Software-Defined Networking System with OpenDaylight Controller’, Helsinki Metropolia University of Applied Sciences, March 2016

[10] W. Braun and M. Menth, “Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices,” Future Internet, vol. 6, no. 2, pp. 302–336, 2014.

[11] OpenDaylight Oxygen User Guide https://docs.opendaylight.org/en/stable-oxygen/user-guide/virtual-tenant-network-(vtn).html

[12]https://www.ibm.com/support/knowledgecenter/SSSRR3_7.5.0/com.ibm.wbpm.wid.tkit.doc/mme/rmetricandkpitemplates.html

[13]https://books.google.ca/books?id=1fSbBQAAQBAJ&pg=PA12&lpg=PA12&dq=metrics+calculation+in+savi+testbed&source=bl&ots=22_XzcFAzM&sig=ACfU3U1Q27r_f5Ya-9ZafhFesSKj4f6JOg&hl=en&sa=X&ved=2ahUKEwiP1cuQpcvhAhUEQ6wKHeu4D2oQ6AEwA3oECAoQAQ#v=onepage&q=metrics%20calculation%20in%20savi%20testbed&f=false

[14]https://www.juniper.net/documentation/en_US/junos/topics/concept/measurement-points-kpi-and-baseline-settings-junos-nm.html