A Middleware for Integrating Legacy Network Devices into Software-Defined Networking (SDN)

Bhargava Sokappadu(✉) and Avinash Mungur

University of Mauritius, Reduit, Mauritius
sokappadu@gmail.com, a.mungur@umail.uom.ac.mu

Abstract. Software Defined Networking (SDN) definitely brings along benefits such as manageability, automation of network and management processes amongst others, however, at the expense of major drawbacks such as huge investment in SDN-capable hardware, vendor lock-in and backward incompatibility with legacy devices. SDN itself being based on a new concept, provides very few aspects in common with traditional networking devices with each SDN vendor usually limiting the SDN capabilities to their own devices only. Even with the introduction of open protocols such as OpenFlow with the aim to provide vendor neutrality, backward compatibility still remains a problem. This paper is geared towards addressing the main issues governing the migration towards SDN and hence provide the desired vendor neutrality, backward compatibility without compromising on networking features, security, ease of deployment and management inter alia. With this concept in mind, an SDN Middleware System has been conceptualized to offer the aforementioned features whereby the backend of the system would be responsible to intercept, inspect and process OpenFlow configurations from the SDN Manager and the SDN Controller and thereafter interpret these commands converting them into the desired configuration in legacy networking terms after which, the legacy nodes are configured with the equivalent of the legacy vendor OS.

Keywords: Software Defined Networking · OpenFlow · Middleware

1 Introduction

Software Defined Networking (SDN) has been one of the major recent changes that has been introduced into the networking era after a long influence of traditional networks. The main key of SDN in tandem with Network Function Virtualization (NFV) is geared to provide automation in the implementation, configuration and operation of networking nodes such as switches, routers, firewalls with minimal manual intervention contrasting against legacy networking principles [1].

Today, key players in the networking ground are massively campaigning software-defined solutions and laying forward advantages to encourage the adoption of SDN.

R. Zitouni et al. (Eds.): AFRICOMM 2020, LNICST 361, pp. 121–139, 2021.
https://doi.org/10.1007/978-3-030-70572-5_8

However, challenges in the migration towards SDN still prove to be undealt with such as huge investments, insufficient multi-vendor interoperability and backward compatibility with legacy systems amongst others [2]. The Open Networking Foundation (ONF) established itself to mitigate the issue of interoperability while providing a unified networking protocol known as the OpenFlow to uniform SDN controller platform. Still, it should be noted that not all network devices are SDN capable and there are yet many legacy network devices in current use.

In this context, this paper aims at assessing the feasibility of optimizing the way SDN works in such a way that legacy networking systems can still make optimal use of software-defined technology irrespective of vendor, hence, addressing the fundamental issues of interoperability and backward compatibility, while reducing the investments involved in the adoption of SDN technology. This would promote the development of a proposed SDN Middleware System that would be able to bridge the gap between SDN standards and legacy networks.

1.1 Problem Statement

The migration towards an SDN platform is highly dependent on the network architecture and inventory which can also imply that SDN adoption requires a major network refresh in order to have SDN capable devices that can support protocols such as OpenFlow. OpenFlow has been established to provide vendor neutrality but however, vendors are putting today their own SDN solutions that are to a major extent, proprietary [3, 4]. These proprietary solutions offer limited interoperability among different vendor devices and the operation is mostly limited to their in-house solutions only.

Hence, SDN is difficult to be deployed in legacy networks that are not SDN-aware such as routers, switches and firewalls which are still very widely in use today. Riverbed Global survey 2017 highlights that 85% of business decision makers claim to be still several years away from digital transformation due to part of their legacy infrastructure [5]. This in turn signifies that a majority of the market is still dependent on their current legacy infrastructure. Migration towards an SDN solution would be more than a paradigm shift and even with open protocols such as OpenFlow, there exists limited documentation for the configuration, implementation and operation of OpenFlow across a multi-vendor network topology and a suitable migration plan. Even Cisco who had once developed the OpenFlow based controller has announced the End of Life [6]. Very little attention is being given to the inclusion of legacy network devices and how they can fit into the SDN scenario.

2 Background Study

In this section, we will provide a brief overview of the SDN concepts and the SDN architecture. An overview of the OpenFlow protocol is also provided.

2.1 Software-Defined Networking Concepts

The SDN Architecture. The basis of SDN Architecture can be summarized into 3 principles:

1. Decoupling of controller and data planes.
2. Logically centralized control.
3. Exposure of abstract network resources and state to external applications [11].

Traditional routers and switches incorporate a strong amalgam between the control and data planes. This tandem rendered management operation such as configuration and troubleshooting very challenging. In order to alleviate these issues, traffic engineering approaches to separate control and data plane was a must. Over time, equipment vendors implemented packet forwarding logic directly into the hardware, separate from the control plane software. In addition, another issue that needs to be addressed with isolation of the control plane is to have a single management platform (later defined as the SDN controller) which would act as the "brain" of the network architecture. As compared to local control in conventional networks, the centralized SDN controller would therefore be responsible to provide control traffic to the network equipment via programmability in the control plane through the SDN controller and since a uniform control platform is maintained, a network-wide visibility, scalability and decision-making could be achieved [11].

Having key roles, SDN controllers are designed to provide better adaptive network path selection, while minimizing outages during network changes such as routing and providing enhanced security such as blocking suspected attack traffic. The SDN controllers assume the role of logically centralizing control procedures while providing standardized communication protocols – which is possible with the use of OpenFlow as open routing software. This would imply that a single server (the SDN controller) can store all the routing, switching rules and contain all the decisions while the networking devices being controlled would in turn rely on the intelligence of the SDN controller [7, 12–14]. Figure 1 depicts the typical SDN Architecture that has been devised and which is used by most SDN platforms including OpenFlow.

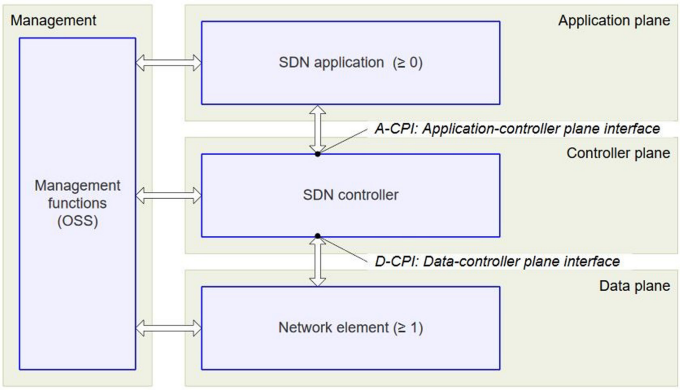

Fig. 1. SDN architecture overview [11]

The OpenFlow Protocol. The OpenFlow Protocol aka OpenFlow Switch Protocol set up by the ONF is the medium that defines how the OpenFlow controller communicates

with the OpenFlow switches. Similar to a traditional switch, the OpenFlow switch relies on basics such as routing and switching but with certainly a major variety of features. However, one major difference in the configuration between these two is that OpenFlow switches need to be managed by an OpenFlow controller which can configure the switch through the OpenFlow switch's tables. This is equally applicable to OpenFlow and OpenFlow-hybrid switches, where the latter can perform both traditional networking and SDN capabilities at the same time but would still require an OpenFlow controller for the OpenFlow segment to be operational.

1. Flow table.
2. Group table.

Figure 2 shows the SDN architecture within an OpenFlow switch and the main components of the OpenFlow switch and its interaction with the OpenFlow controller.

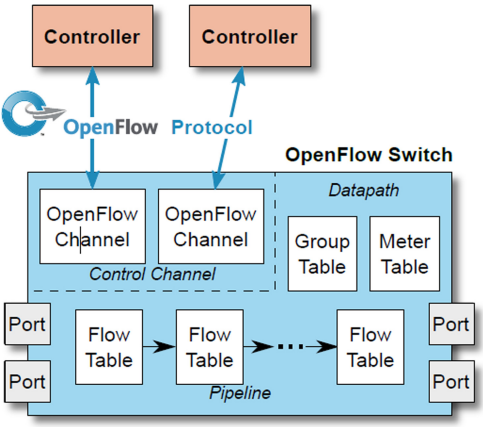

Fig. 2. Main components of an OpenFlow switch [15]

Each of the above tables may contain individually a set of different flow entries and group entries respectively. Therefore, the OpenFlow controller can add, update, delete, flow entries and group entries using the OpenFlow Switch protocol. A flow table would contain a set of flow entries where each entry would define parameters similar to the policy, condition and action triad but specific to the OpenFlow protocol such as match fields, actions on traffic, and instructions among others.

Within the flow table, the matching sequence of traffic compared against the flow entry starts in a top-down approach where the first matching entry in each table is executed first and if a policy and condition is matched, the corresponding actions as defined in the flow table entry are applied. Otherwise, if there is no suitable match, the result is based on the Table-miss flow entry parameters that define what action to take when no match has been found, for example this can be allow/drop on no match.

Traffic matching a flow entry can be as an action, forwarded to either a physical or virtual port and actions can be set to redirect traffic to a group that can provide additional

processing. In this case, a group entry would therefore contain a set of actions determined by the group type itself [15].

2.2 Current Research Developments

Currently, there exists very limited research material regarding the compatibility of legacy devices with OpenFlow controllers or other SDN venues. Most of these researches are geared towards the performance analysis of OpenFlow controllers in a lab network with limited deployment in practical network setups [16–18]. The issue of legacy device support is addressed partly by [19] where the use of home gateways to be integrated in an OpenFlow-based SDN network while at the same time exploiting the hardware available to accelerate traffic and its processes. However, it can be inferred that home gateways (CPEs) are not suitable to fit into an SDN topology since they lack advanced functionalities and granular access to physical registers which are usually vendor-locked. The "HARMLESS" approach as depicted in [20] is among the few that partially addresses the research problem concerns and the strategy here is to add a SDN switch that can add SDN capability to legacy network systems by including a layer of virtualization based on Tagging and Hairpinning. Among the few papers, [20] focusses on operability and cost of legacy networks in SDN networks but yet, we consider the approach to technically insufficient since HARMLESS would require additional hardware such as server for spine topology where the use of 10G switches are solicited with overwhelming port capacities and additional CPU which is at the expense of providing a low-cost solution. The research from [21] provides a good basis to use the OpenFlow configurations but provides very limited information on how the interpretation and conversion is realized from the OpenFlow, how the administration is handled and how it can be used for various configurations. The research lacks qualitative evaluation and test cases that would suit more than the TCMA performance. The incremental deployment of SDN as mentioned in [22] provides a limited practical approach on how SDN can be implemented into hybrid networks. Following the critical review of several papers [16–20, 23, 24] it is with concern that we conclude that limited importance is being given to the main aspects of SDN adoption which we believe to be primordial and hence form the very basis of this research which is geared towards support for legacy devices, multi-vendor support, device discovery, ability for multiple configurations, minimal required resources, ease of implementation and management, cost effectiveness, security, scalability and performance.

3 Proposed SDN Middleware System

Typically, a basic SDN topology would require the SDN controller in the Control Plane, the OpenFlow switch in the Data Plane with the network elements associated to the switch and the OpenFlow Manager in the Management/Application Plane that would be used to manage the configuration of the SDN controller. The functionalities of a practical SDN environment would be as per Fig. 3.

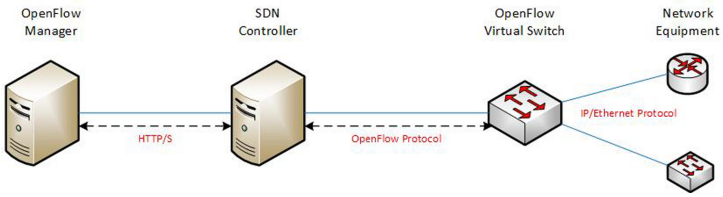

Fig. 3. Practical SDN setup

During the end-to-end network configuration of the switch to allow data plane process the traffic of the network elements, several protocols are used. First, the configurations executed in the OpenFlow Manager are triggered via the HTTP/s to the SDN controller. The SDN controller sends its configuration (flows) up to the OpenFlow switch via OpenFlow protocol while for the Data plane layer on the OpenFlow switches, it is the usual TCP/IP stack protocol that is preserved.

However, in order to have the control of non-SDN compliant devices, there is the need to have the middleware that would bridge the gap between the configuration of SDN and legacy devices and hence, through a single management console as well. This can be achieved by having terminal direct access to the manageable legacy switches via remote management protocols such as SSH. In this light, the approach of the proposed middleware is to be able relay the information input into the OpenFlow Manager which would be the management console, to both the SDN controller and the configuration of legacy devices. In this project, the middleware is highly leveraging on the fact that the configurations "pushed" from the OpenFlow Manager are in HTTP requests and responses while the parameters are most commonly sent via JSON scripts.

In the proposed solution, the middleware should therefore be able to effectively "tap" into the path across the flow sequence from source (OpenFlow Manager) to sink (Network Devices) and interpret the HTTP/S/JSON messages from the OpenFlow Manager parsing them into the proprietary language based on the operating system of the legacy switch. The middleware would therefore fit into the topology as proposed in Fig. 4.

The SDN Middleware System proposed will have several uses one of which is fundamentally to be able to gain management of a "hybrid" architecture – containing both SDN-capable and legacy network nodes. This is typically the observed scenario in practice where either campus or data center networks even though have part of their network SDN-ready, still retain some of the legacy networking devices such as routers and switches especially those that offer compatibility with older protocols/services.

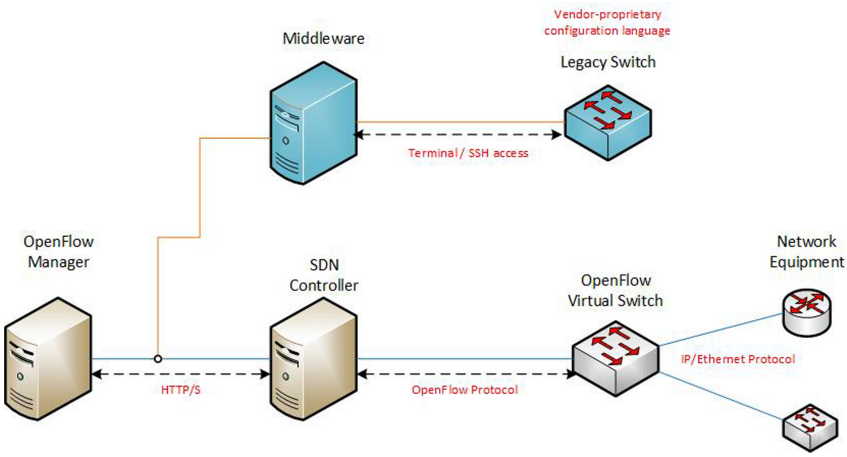

Fig. 4. SDN middleware in practical hybrid architecture

4 Design

In this section, the detailed design of the SDN Middleware System is provided comprising of its architecture and software component.

4.1 General Architecture

Figure 5 outlines the overview of the proposed architecture for the SDN Middleware System depicting the primary components within the architecture. The Middleware server would logically sit in between the path from the SDN Manager towards the network nodes while passing through the SDN controller for configurations. The first aim is that the Middleware server should be able to familiarize with the network by identifying the different legacy nodes that are connected. Next, it should be able to intercept and inspect the SDN messages that are being configured from the SDN manager along the path towards the sink (legacy nodes). The Middleware server would then process these SDN/OpenFlow messages into appropriate interpretation that would be then used as configuration parameters to be deployed to the legacy nodes.

4.2 Software Design

Initialization of the Middleware. As a basic requirement of the Middleware service, it should be able to connect and manage legacy network nodes in the first instance making this feature the very basis of the Middleware solution. This pre-requisite would involve the Middleware to have the ability to keep a known repository of the legacy device nodes and their respective Cookie mapping such that each device connected can be uniquely represented (to be discussed in the later section) that are connected and further details such as Management IP, Vendor are highly desirable. This would be similar to a topology inventory map closely related to a sort of a neighbor/discovery table of the

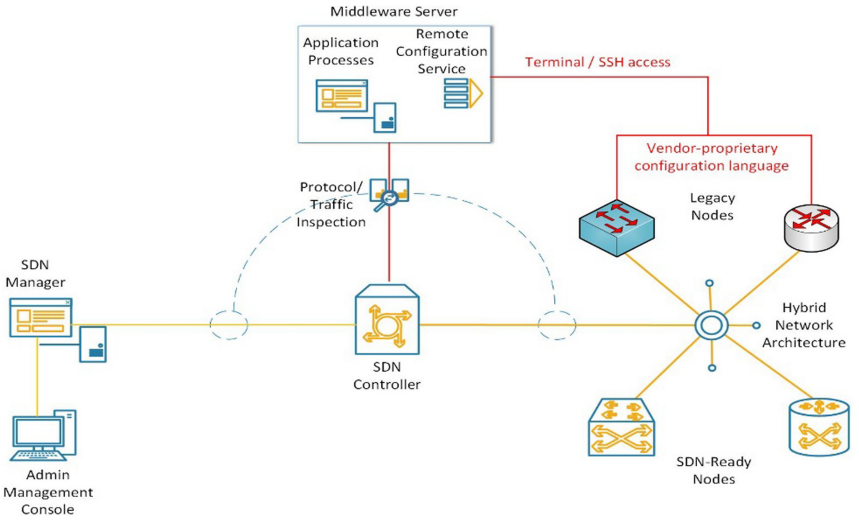

Fig. 5. SDN middleware general architecture

architecture. In order to address this issue, the use of device discovery is highly solicited – this includes the use of protocols such as LLDP for devices directly connected to the Middleware while SNMP for those that are remotely connected.

Interception of SDN Messages. Whenever a flow deployment is executed from. OFM, the OpenFlow Manager calls a corresponding RESTCONF API from the OpenDaylight controller which is specific to the configuration deployed. Further to this, the command is executed through a HTTP PUT request which embeds a set of parameters corresponding to the RESTCONF API in a sequence of JSON parameters. It is much easier to capture and decipher the HTTP commands rather than intercepting the OpenFlow protocol messages between the controller and the OpenFlow switches the former being more structured (JSON as compared to OpenFlow messages that are more complex to interpret). Therefore, the flow of the command is from the SDN manager towards the SDN controller and the PUT request is executed at the SDN controller.

The interception of these messages can either be done at port or service level or by tapping from the interface itself but at the expense of careful filtration and inspection of only related information using packet-capture and packet-inspection tools. The SDN Middleware System shall therefore leverage on intercepting these HTTP commands and strip the different parameter values which will be further processed to generate relevant configurations in the proprietary script of the legacy devices.

Inspection of SDN Messages. Following the interception of messages, the gathered raw data would have to be further filtered to obtain the desired parameters through the process of inspection. Deep-packet inspection would be used to further drill down into the details following the HTTP messages. The parameters that interest the most are the JSON objects with their member key and values that are located inside the HTTP PUT requests. These parameters will be the variables to be interpreted to build the commands for the legacy nodes.

Processing the SDN/OpenFlow Messages. OpenFlow configurations are sent from the SDN controller to the OpenFlow switch by making use of flow entries within flow tables as viewed in Sect. 2. The parameters that are used within the definition of flow entries are those that are actually embedded within the JSON and have been inspected earlier. The summary of operation of each of these fields are as per below:

- Match Fields – contain the parameters to be used to determine a match in ingress and egress port headers. For example, match source IP address or MAC address.
- Priority – contain the precedence of the flow entry.
- Counters – contain the hit count whenever packets match the parameters set.
- Instructions – set the action to be taken for e.g. drop, output port-no. etc.
- Timeouts – the maximum amount of time before the flow is disregarded by the switch.
- Cookie – denotes an opaque data value that maybe used to filter flow entries following flow modification or flow statistics but is not used for packet processing.
- Flags – Modify the way flow entries are managed.

One way the SDN controller can determine to which switch the configuration is sent is usually through the OpenFlow Device ID which should be unique throughout the SDN domain. Since the OFM/SDN controller can see only the switches connected via OpenFlow but does not show any visibility on legacy nodes, the SDN Middleware System shall itself run on OpenFlow as the back-end connector with the OFM and thus, the SDN Middleware will also be managed and configured using the OFM itself. Therefore, extrapolating this in practical scenario, there will be a single SDN Middleware System to control various legacy nodes and given that the SDN Middleware System runs on OpenFlow, the OFM will be showing only one OpenFlow switch which is the Middleware itself but there would be no way to select the specific legacy node. So therefore, there should be at least one unique identifier value that can be used to determine the specificity of each of the legacy device. In order to address this issue, given that the Cookie (this Cookie value is restricted to OpenFlow and not in reference to the HTTP cookies) value in a flow entry is an opaque value that is not used in decision making, the Middleware can make use of the Cookie value in order to determine the identity of the switch to which the configuration is to be sent. A different parameter could also be used as unique identifier for the legacy switches as well. The decision-making process in terms of OpenFlow Device ID and Cookie value is demonstrated as per Fig. 6.

Referring to Fig. 6, each legacy switch has been assigned a specific Cookie value (in hexadecimal) of 0x001 and 0x002 respectively. In this specific SDN domain, the OpenFlow Manager will be able to see two OpenFlow switches connected and active with the SDN controller. The SDN controller will still be differentiating among the different OpenFlow switches using the device ID (here ID = 1234 for the OpenFlow switch and ID = 1001 for the OVS underlay within the Middleware). Therefore, to send a configuration to OpenFlow switch, the Device ID is directly selected while to send configurations to legacy nodes, specifically to the Middleware, the Device ID of the latter (1001) would be selected. The Middleware will in turn pass the JSON parameters and the legacy node is determined by the Cookie value, for example, if a flow entry with Cookie value with 0x001 is encountered, the Middleware would know that this configuration is

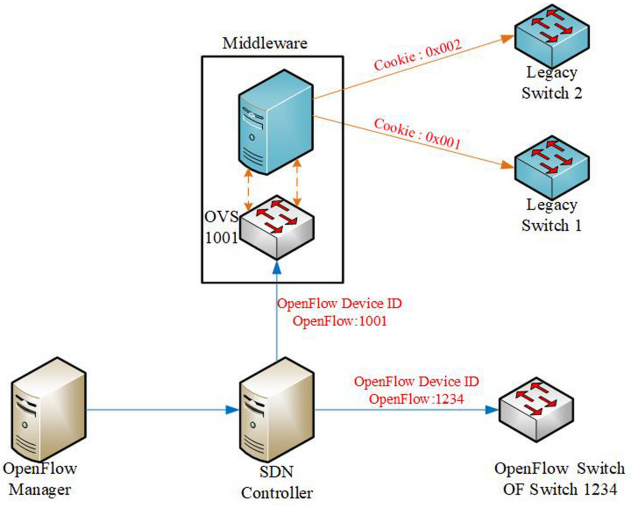

Fig. 6. Decision-making of the SDN controller and middleware

meant for Legacy Switch 1. This also means that a table of values of Cookie v/s Switch needs to be populated and maintained as a legacy node is added to the Middleware.

OpenFlow messages sent between the controller to the OpenFlow switch are characterized by their OpenFlow command message. For example, insertion of a flow entry into a table, it would correspond to an OFPFC_ADD message and a modification of an existing flow would be an OFPT_FLOW_MOD request. The different types of request provide information regarding the nature of the configuration is to be done on the OpenFlow switch i.e. whether it is an add request, delete request among others. This information in the OpenFlow header is also parsed by the Middleware to interpret into the type of command to be executed on the legacy node.

Preparation of the Command Block. The parameters that have been processed from the JSON into OpenFlow protocol will be stored in variables within a local repository of the Middleware. These variables will then be used in the command execution process in sending the configuration to the legacy nodes. Prior to that, it would be important to known how each of the different OpenFlow commands will be "converted" into the legacy command. For example, assuming the case where a command from the SDN manager is sent to the SDN controller to create an Access Control List (ACL) between two hosts, the Middleware should be able to interpret and parse the parameters into an ACL entry that can then be applied to different legacy nodes. However, the main issue is that OpenFlow messages are more elusive compared to legacy network configurations that are more easily interpretable in terms of keywords that can be found within the configuration lines. This is a major "limitation" of OpenFlow since OpenFlow messages are difficult to be interpreted, however the easiest approach is to program the Middleware such that based on the correlation of the retrieved parameters, it can interpret the logic behind the OpenFlow command. Below is a sample flow entry within an OpenFlow-based OVS.

```
cookie=0x777, duration=189.072s, table=200, n_packets=0,
n_bytes=0, priori-
ty=2,ip,nw_src=192.168.100.100,nw_dst=192.168.100.200 ac-
tions=drop
```

From the above, example of OpenFlow command, the following can be extracted:

- Cookie = 0x777
- Table ID, Priority = 200,2
- Network Source = 192.168.100.100
- Network Destination = 192.168.100.200
- Actions = Drop

The parameters of this flow entry within the flow table relates to the an OpenFlow entry to block traffic from source network 192.168.100.100 and destination network 192.168.100.200. Since the source and destination networks are of /32 subnet mask, this means that this corresponds to a host-to-host deny ACL entry when mapped to legacy network configuration. As shown in Fig. 7, the matched flow-entry components are mapped to the respective variables within the Middleware which will be used for the command execution process.

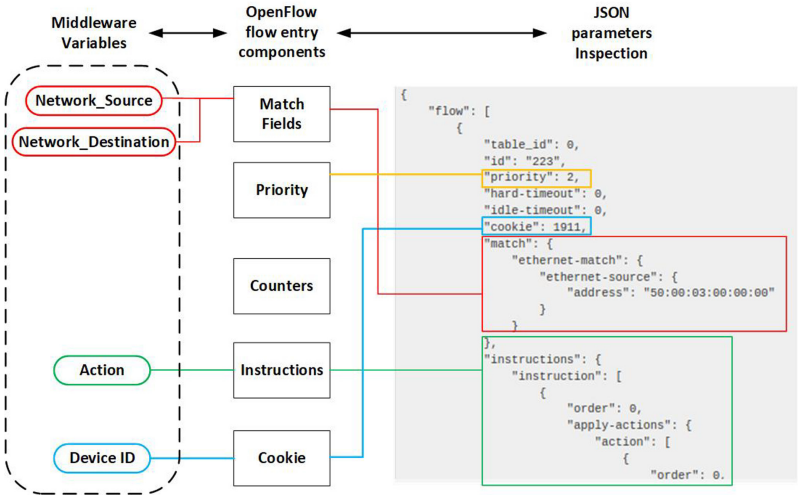

Fig. 7. Mapping of variables in middleware

Command Execution. Each different legacy device coming from a different vendor, OS/Firmware version or capability is most likely to have its own language set in terms of command configuration and this entails to exquisite command execution on per-device basis and a generic command set might not be applicable for all legacy nodes. Command execution implies pushing of the desired configuration into the legacy device for which,

other than the command itself, the privilege to apply the configuration is required. Thus, in this design, the command execution would be done by performing an SSH to the legacy device and executing the commands through a preconfigured script and scripting techniques which will contain the instructions based on the OS/Firmware and variables that have been stored previously. Along with the instructions and parameters, remote device access credentials are fundamental to be able to have the privilege to perform the configuration. Therefore, it will also be required that these credentials to be input and stored into a repository via the management console. This will be parsed through the Device Registration process.

5 Implementation

The implementation of the SDN Middleware System has been made using open-source based solutions as far as possible mostly to provide scalability for adjustments and limit the cost of development. The SDN Middleware System architecture is loosely based on a Linux platform atop having the following running features of:

- OpenFlow vSwitch to interface with the SDN Controller/Manager.
- Node-Red on Linux (Ubuntu) Platform which retains the backbone of the Middleware engine based on JavaScript.
- TShark component to provide packet-capture features.
- MySQL Database to provide repository.
- Apache HTTP server with PHP (LAMP) and JavaScript for providing Middleware GUI features.

5.1 Initialization of the SDN Middleware System

The first action of the Middleware would be to be able to perform neighbor discovery in order to gather information on the different legacy nodes that have been connected thereto. The protocols used depends on how the devices are connected to the Middleware and in this case, given that the legacy nodes are directly connected, LLDP protocol for neighbor discovery has been used, another major reason being LLDP support by multiple vendors. The implementation has been done as the LLDP service installed on the Linux and a Shell script polling devices at specific intervals of 30 s which is the default LLDP frequency timer. The output of this repeated process is then processed to extract the required parameters such as Vendor, System Name, and others which are then used to update the table “Inventory” within the database and this acts as the node inventory for listing the connected devices and also updating the entries for changes.

5.2 Interception of SDN Messages

The interception of messages involves the deep packet inspection of the egress traffic of the SDN Manager towards the SDN controller and for this purpose, TShark variant of Wireshark has been used to effectively sniff the traffic on the interface that is being

used to tap the traffic (in this scenario). In order to simplify the sniffing process, filters for TCP protocol ports at Transport layer and HTTP PUT requests at the level of the Application layer have been used to increase efficiency. These packets specifically would contain the parameters being sent to the SDN controller. Given the bulky output during HTTP/S inspection, the output following the inspection is first written in a file locally on the Middleware before further processing/inspection of SDN messages can begin. However, the payload of this HTTP request contains the actual data elements that are required for the Inspection part. This payload would be in raw HEX format.

5.3 Inspection of SDN Messages

The output of the Inspection is in raw format and embeds several other HTML codes along with the HTTP PUT request. The purpose of the Inspection layer is to strip the required information only, to have it handy in a format that would be suitable for further processing of the SDN messages. In this context, since the intercepted output is raw, unparsed format, the inspection would result in extraction of the payload in JSON format that would ease the processing steps. The content of the stored file will have its data payload extracted and the HEX converted to string format following which, with the help of indexing methods, only the required payload in the JSON format is extracted. This JSON script is then used for further processing and at the same time, logged to the Middleware database for logging and auditing purposes. This format serves as the base material that would be used in the building of the command blocks for code execution. The output of the Inspection process would give an output in a readable structure that contains specifically the parameters that will be used for further processing.

5.4 Processing of SDN Messages

This segment of the implementation will deal with the extraction of required parameters from the extracted JSON format into a set of variables that will thereby be used for command execution part. Thus, this involves deeper analysis of the inspected data to match against the components of Flow Entries within a flow table. This has been conceived in practice by the indexing of the keyword components of the JSON and storing each component and its respective defined parameter into a well-structured table within the Middleware's internal database. Similarly, this algorithm for parameter extraction has been developed using JavaScript in Node-Red with the output of this process used for storage and triggering for command preparation.

5.5 Command Preparation and Execution

This part of the implementation process is the most fundamental since it deals with the final aspect of the Middleware's process which is to successfully be able to send the configuration to the legacy device based on the language the configurations on the legacy device is based on. During the implementation, the language that has been tried to abide to is the Cisco IOS. As depicted in the earlier sections, the components being analyzed compose of an OpenFlow entry that is complementary to the Cisco IOS Access

Control List. Similarly, each vendor would have its own language set for configuration but yet, the parameters to be used within the configuration remain the same variables throughout the different vendors. Hence, this segment would interpret these parameter options as variables from the Middleware's database and embed them appropriately as per the language set of the vendor (in this case Cisco IOS) and send this configuration to be executed at the legacy device. When it comes to execution, it is primordial to have the privilege to be able to configure the device for which credentials such as Username and Password would be required. To address this issue, the implementation also involves a Device Registration process that would allow the entry of the device's credentials that are bound to the Cookie which form the device's identity and until the device has not been registered, no command execution will take place.

The command execution is invoked upon the value of the Cookie ID and the configuration has been implemented through a Shell Script that based on the Cookie ID would fetch the corresponding device details from the Middleware's database, such as, IP address, Username and Password. After gaining SSH access, the Shell Script pertaining to the device configuration itself is then executed into the device and the appropriate logs such as time and command execution are stored back into the database for logging and auditing purposes. Based on the value of the Cookie following the inspection process, the configuration values are extracted from the database and a Shell script is generated in an executable format.

6 Test Results

The test bench used has been in a scenario with OpenDaylight OpenFlow controller, OpenDaylight OpenFlow Manager, and legacy devices based on Cisco IOS and HP ProCurve switches. For test input data at the SDN Manager, the equivalent of Access Control List as an OpenFlow Flow entry has been configured and deployed on both the Cisco IOS and HP ProCurve switches and the same OpenFlow configuration has been extrapolated using a Flow entry to perform static NAT (SNAT). This has produced successful results in converting these commands to their legacy counterparts in Cisco IOS and HP ProCurve for the ACL part while the SNAT was successfully tested on a Cisco router. The SDN Middleware System has been implemented on a Virtual Machine with 4 vCPU, 8 GB RAM and 50 GB disk space. Figure 8 and 9 provide the timeline since the command is executed at the SDN Manager up to the configuration of the legacy node.

Fig. 8. Timeline of events for configuration size of 1.2 KB for an ACL

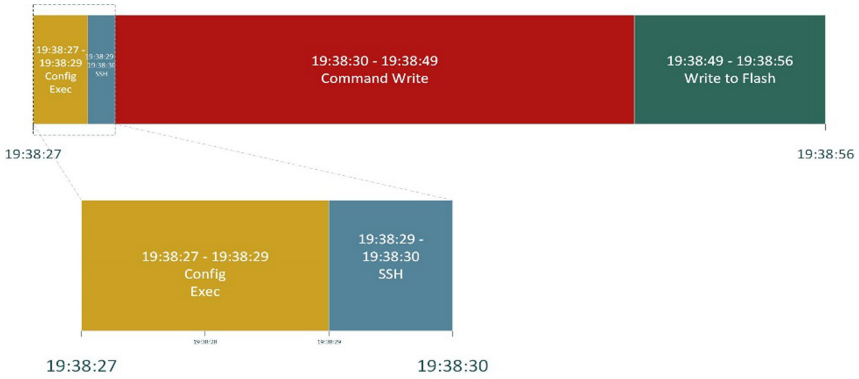

Fig. 9. Timeline of events for configuration size of 4.5 KB for an ACL

It can be inferred that the average of the process duration from the time the command has been executed from the SDN Manager until the configuration is saved is 12 s where the majority of this time is consumed by the legacy node to save the configurations to the flash (8 s) as compared to the performing the configuration onto the device itself (Command Write) takes only 2 s. SSH and Configuration Execution processes take only 1 s each which also means that the SDN Middleware System itself performs the Interception of the SDN messages, Inspection of the SDN messages, Processing of the OpenFlow commands and Preparation of command block processes in only 1 s altogether demonstrating a highly time-efficient system. The long time for writing the configuration to the flash is considered normal since these devices need to copy the configuration from the running memory to the flash and storing as the startup configuration file and by default these legacy nodes have limited resources as well and these results are therefore deemed acceptable since the Command Write process takes only 2 s relatively and this is where the configuration gets added onto the device but onto the volatile memory and it is normal that command execution onto volatile memory takes less time than saving the configuration.

From the above, it can be inferred that for a configuration file size of 4.5 KB, the mean end-to-end process is completed within 31 s where the majority of this time is consumed in writing the command to the device (19 s) while the least time is taken to establish the SSH session (1 s). A long time of 19 s for the command write is expected since the configuration 4.5 KB would contain bulkier configurations of more lines of command. The results for the time duration for the configuration script sizes of 1.2 KB and 4.5 KB are compared side-by-side as per Fig. 10 to give a comparison of the different times taken by each process with increase in configuration size.

The statistical summary in Fig. 10 gives a much more in-depth comparison of the different process times. It can be deduced that despite the variation in configuration size, there are two processes namely SSH establishment and Writing to Flash that have remained constant at 1 s and 8 s respectively for both configuration sizes. This would mean that SSH and Writing to Flash are independent of configuration size and this is theoretically correct since firstly, SSH establishment occurs before the Command

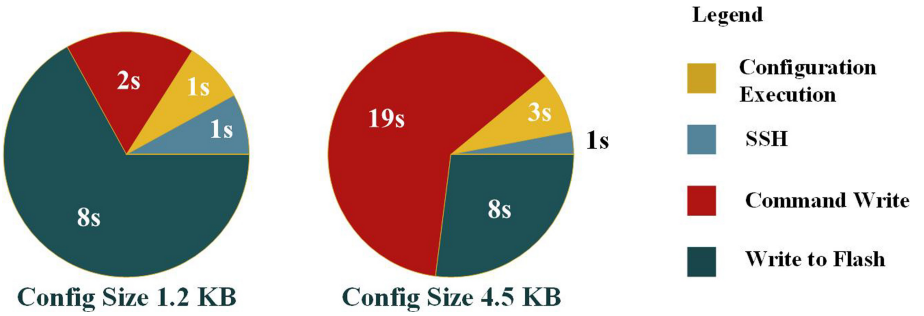

Fig. 10. Comparison of time duration for configuration sizes of 1.2 KB and 4.5 KB

Writing process and simple consists of key exchange and authentication. Next, writing to flash is dependent on the hardware resources in copying the running configuration into the startup configuration making these two processes as independent variables. However, executing the configuration onto the legacy device definitely depends on the configuration size as depicted by Fig. 10 where there is in an increase of 2 s is noted for a configuration increase from 1.2 KB to 4.5 KB which is deemed normal since a larger SDN command execution would imply more processing involved at the level of the Middleware in terms of packet interception, processing of SDN messages and extraction of the parameters in building the command block.

On a final note, it can be said that the SDN Middleware System metrics comply to the requirements in terms of performance metrics especially when comparing to the time involved in the configuration when manual intervention is required to configure the legacy device.

7 Evaluation of the SDN Middleware System

This section provides a cost and performance evaluation of the proposed SDN Middleware System.

7.1 Novelty and Contribution

Our paradigm approach compared to previous research works aims at providing a Middleware that can inspect, interpret, convert and execute the SDN functions while providing the dashboard for Device registration, selection, auditing among others altogether within a single solution. This method has been clearly shown to perform conversion of OpenFlow commands without the use of APIs and amidst a practical test case scenario of a hybrid network architecture where the SDN controller manages the SDN-ready switches while the SDN Middleware manages the legacy nodes while providing higher granularity of configuration.

7.2 Cost Evaluation

Throughout the design and implementation of this project, cost aspects have been given great considerations and as far as possible, any expense or purchase of software or hardware resources have been overridden at its best. For this purpose, a maximum number of open-source products have been solicited to establish the minimal cost target. For the implementation, there has been only EVE-NG and VMware Workstation Pro that are licensed products that have been used. Again, these can also be considered to be void as inclusion for the costs since EVE-NG has only been used to emulated the Cisco IOS virtual platform and same configuration has been tested on a hardware Cisco device as well while VMWare Workstation Pro has been used to host the Ubuntu Linux Kernel for the different services. Hardware requirements being minimum, our hosting environment has been a laptop and undoubtedly, the OVF could be deployed in any virtualized environment and no specific hardware are required as such. The requirements would be specific to a basic Virtual Machine resource provisioning. As mentioned during our choice of the following components, Ubuntu Server Platform, Node-Red, OpenDaylight Open Flow Manager, OpenDaylight SDN Controller, LAMP architecture, LLDP services among others, all of these have been built using open-source solutions rendering the SDN Middleware Architecture free from any CAPEX and OPEX.

7.3 SDN Middleware System Performance

This aspect covers the overall performance of the SDN Middleware System in terms of processing, memory and disk usage while in use. It is to be also noted that during the implementation, the SDN Manager, the SDN Controller, Open vSwitch and the Middleware components have been installed on the same virtual machine for ease of testing. The performance metrics of the system have been monitored and recorded over a period of approximately 10 h of intermittent usage. It can be inferred that the system uses a maximum of 17% of the provided resources. It can be deduced that the virtual machine, again encompassing all components utilizes an average of 5.4 GB of RAM (with an idle RAM usage, unused memory of 2.65 GB) which is reasonable for a system running the SDN Manager, SDN controller, LAMP, Node-Red and Open vSwitch in addition to normal Linux processes.

8 Conclusion

The prime motive behind this paper lies today's inevitable problem of the amalgam of operations between Software Defined Networking and legacy networking devices which pose not only a disparity in the technological timeline but also broaden the financial gap whilst leaving the network industry, the only choice of fully migrating to SDN in one go. This major hurdle remains among the primordial reasons of the reluctance or repulsion towards SDN today. For this reason, this paper has as main objective to the bridge gap between SDN and traditional networking devices by the deployment of a proposed SDN Middleware System capable of providing interoperability between OpenFlow-based SDN and legacy protocols and vendor neutrality without compromising

the financial aspect. In this paper, it has been shown how the SDN Middleware System has been devised to perform packet inspection to capture, inspect and understand the SDN messages into a meaningful form from where the required parameters are used to produce the equivalent legacy OS configuration. In addition, the legacy configuration can also be varied to suit the OS depending on the legacy device to provide compatibility across multiple vendors. The functionalities of the SDN Middleware System have been put to test to achieve an execution time of the automatic process of deployment from the SDN Manager up to the device configuration to be achieved within approximately 12 s which is undoubtedly much lower than that of a human-intervened manual process. At the same time, it has been illustrated how multiple configurations can also be executed through the SDN Middleware System and how it also provides the GUI platform for ease of management while interconnecting the different segments. It should be highlighted that all the implementation of the SDN Middleware System has been designed based on an open-source platform to render the Middleware to zero cost of software operation where only hardware resources are required. Taking all these aspects into consideration, it can definitely be concluded that the devised SDN Middleware System complies to the aims of this paper and based on the results, it can be classified as a major contribution in this field of research to establish the basis towards vendor neutrality and interoperability in the adoption of SDN.

References

1. Santana, G.A.A.: VMware NSX Network Virtualization Fundamentals. VMware Press, Palo Alto (2017)
2. Sokappadu, B., Hardin, A., Mungur, A., Armoogum, S.: Software defined networks: issues and challenges. In: Conference on Next Generation Computing Applications (NextComp), Mauritius (2019)
3. Open Networking Foundation: ONF Strategic Plan, ONF Board, Menlo Park, California, March 2018
4. Cisco: Cisco Application Policy Infrastructure Controller Data Sheet. Cisco Inc., San Francisco (2018)
5. Riverbed: Riverbed Future of Networking Survey Finds Legacy Networks Holding Back Cloud and Digital Transformation. Riverbed, San Francisco (2017)
6. Cisco Inc: End-of-Sale and End-of-Life Announcement for the Cisco Open SDN Controller 1.x. Cisco Inc. (2016)
7. Feamster, N., Rexford, J., Zegura, E.: The road to SDN: an intellectual history of programmable networks. ACM: Association for Computing Machinery (2013)
8. Open Networking Foundation: Open Networking Foundation Press Release. Open Networking Foundation, Oregon (2011)
9. Gartner: Gartner Identifies the Top 10 Strategic Technology Trends for 2014. Gartner Inc., Orlando (2013)
10. Fortinet: The Fortinet SDN Security Framework. Fortinet Inc, Sunnyvale (2016)
11. Open Networking Foundation: SDN Architecture Overview. Open Networking Foundation, Palo Alto (2014)
12. Caesar, M., Feamster, N., Rexford, J., Shaikh, A., van der Merwe, J.: Design and implementation of a routing control platform. In: Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2005)

13. Lakshman, T.V., Nandagopal, T., Ramjee, R., Sabnani, K., Woo, T.: The SoftRouter architecture. In: Proceedings of the 3rd ACM Workshop on Hot Topics in Networks (HotNets) (2004)
14. van der Merwe, J., Cepleanu, A., D'Souz, K., Freeman, B., Greenberg, A., et al.: Dynamic connectivity management with an intelligent route service control point. In: ACM SIGCOMM Workshop on Internet Network Management (2006)
15. Open Networking Foundation: OpenFlow Switch Specification Version 1.5.1. Open Networking Foundation, March 2015
16. Shamim, S., Shisir, S., Hasan, A., Hasan, M.: Performance analysis of different open flow based controller over software defined networking. Glob. J. Comput. Sci. Technol. **18**(1), 11–16 (2018)
17. Umenne, P., Lindinkosi, Z., Kingsley, A.O.: Emulating software defined network using Mininet and OpenDaylight controller hosted on Amazon web services cloud platform to demonstrate a realistic programmable network. In: Easychair (2018)
18. Kim, H., Kim, J., Ko, Y.-B.: Developing a cost-effective OpenFlow testbed for small-scale software defined networking. In: 16th International Conference on Advanced Communication Technology, Pyeongchang, South Korea (2014)
19. Miano, S., Risso, F.: Transforming a traditional home gateway into a hardware-accelerated SDN switch. Int. J. Electr. Comput. Eng. (IJECE) **10**(3), 2668–2681 (2020)
20. Csikor, L., Szalay, M., Retvari, G., Pongracz, G., Pezaros, D.P., Toka, L.: Transition to SDN is HARMLESS: hybrid architecture for migrating legacy ethernet switches to SDN. IEEE/ACM Trans. Netw. **28**(1), 275–288 (2020)
21. Hand, R., Keller, E.: ClosedFlow: OpenFlow-like control over proprietary devices. ACM (2014)
22. Hong,D.K., Ma, Y., Banerjee, S., Mao, Z.: Incremental deployment of SDN in hybrid enterprise and ISP networks. ACM (2016)
23. Franciscus, X.A.W., Gregory, M.A., Khandakar, A., Gomez, K.M.: Multi-domain software defined networking: research status and challenges. J. Netw. Comput. Appl. **87**, 32–45 (2017)
24. Sonchack, J., Adam, J.A., Keller, E., Jonathan, M.S.: Enabling practical software-defined networking security applications with OFX. In: Network and Distributed System Security Symposium (2016).