NET4901 - Final Report

Brad Fitzgerald - 100969645, Sam Cook - 101004349,
Samuel Robillard - 100967018, Josh Nelson - 100981092

Supervisor: Dr. St-Hilaire

April 16, 2019

Abstract

This report details our undergraduate research project regarding the research and design of a Software-Defined Network (SDN) Monitoring application, which we dubbed SDLens. We outline the state of monitoring in traditional networks, how the shift to SDN changes the approach to monitoring, and the opportunities and challenges that come with monitoring these emerging types of networks. We then provide some background information on the architecture of SDN and the protocols involved. This is followed-up by an in-depth discussion of SDLens, including the architecture of the application and the features we have developed. We detail the high-level design of SDLens features, and which monitoring capabilities they provide. Among these capabilities we have dynamic topology views, traffic flow tracing, performance graphs, and statistics tables, and others which we will detail in coming sections. We then conclude our report by summarizing our accomplishments, reflecting on the lessons learned, and offer suggestions as to how the project could be improved in future years.

Acknowledgements

We would like to thank our project supervisor, Dr. Marc St-Hilaire for giving us the opportunity to work on this project. We would also like to thank him for the guidance, encouragement, and confidence he had given us throughout the academic year. We would also like to thank many of the core professors who have taught us in the Network Technology program: Michael Anderson, Carolina Ayala, David Bray, Dr. Ashraf Matrawy, Dr. Wei Shi, and Dr. Richard Yu. The knowledge and skills we have acquired from their courses and consultation have helped us immensely throughout the duration of this project.

Contents

1 Introduction **6**
1.1 Problem Background 6
1.2 Problem Motivation 6
1.3 Problem Statement 7
1.4 Proposed Solution 7
1.5 Accomplishments 7
1.6 Overview of Remainder of Report 8

2 Technical Section **8**
2.1 Background and Terminology 8
2.2 Project Details 9

3 Application Architecture **9**
3.0.1 Web Application Architecture 10
3.0.2 Agent Architecture 11
3.0.3 Database Architecture 12

4 Key Application Features **12**
4.1 Interactive Topology Page 12
4.1.1 Topology Page Features 13
4.1.2 Throughput and Basic Flow Information 13
4.1.3 Switch and Host Information 14
4.1.4 Spanning Tree Topology 14
4.2 Topology Page Architecture and Design 15
4.3 Statistics Table 15
4.4 Interface Graphs 16
4.5 Startup Script 18
4.6 Flow Tracer 18
4.6.1 Flow Tracer Implementation Details 19

5 Conclusion and Recommendations **20**

List of Figures

1 SDN Architecture 9
2 SDLens Architecture 10
3 Web Application Work Flow 11
4 SDLens Topology Page 13
5 Throughput and Basic Flow Information 13
6 Switch Clicked 14
7 Host Clicked 14
8 Spanning Tree Active 15
9 Port Counters Table 16
10 Received and Transmitted Packets 17
11 Flow Tracer Options 19
12 Flow Tracer Topology View 19
13 Flow Trace Information Panel View 19

1 Introduction

1.1 Problem Background

Software Defined Networking (SDN) disaggregates the Control plane from the Data plane on a network device. In the network stack, the control plane makes the decisions about where traffic is sent, and the data plane forwards incoming traffic based on the decisions made by the control plane. Traditionally, the control plane is distributed on every device, each being responsible of making its own forwarding decision and managing its own policies. In SDN, the control plane is separated from the data plane, and no longer lives on every device. A logically centralized application, known as an SDN controller, is now responsible for the control plane policies of the entire network. This allows for greater levels of control and oversight into the network as the controller holds the state information for the devices in its domain. This disaggregation provides network operators with a central location they can use to interact with the network. The use of a controller also exposes an application programmable interface (API), that engineers can now use to create applications to manage the control plane.

Network monitoring is the practice of obtaining data about a current network in order to detect changes in the network and allow for an operator to make changes based upon the information obtained from the network. Due to the increasing size of modern networks, the complexity of monitoring is also increased, making the use of a centralized point for monitoring more appealing.

1.2 Problem Motivation

Many traditional network monitoring solutions rely on protocols like the Simple Network Management Protocol (SNMP). Although SNMP has been a heavily used protocols for multiple decades, there are many issues it introduces. SNMP is agent based, meaning that every networking device must have SNMP installed and configured for functionality. The polling process for SNMP is inefficient and cumbersome, which is undesirable in large scale networks. SNMP also relies on sending information stored in Management Information Bases (MIBs), which are often poorly defined and vendor specific. This creates even more challenges in a multi-vendor environment, as the monitoring system must keep track of which MIBs are present on various devices.

Another issue with network monitoring is the location for the collection of data. In large traditional networks, each device is required to send its data to a collection point, which can be a slow and inefficient process. With the centralization of the controller, SDN monitoring tools allow for the collection of SDN statistics from a singular point. This information can be used to quantify health and metrics of key network elements, as well as provide performance metrics. This in turn yields significant benefits in security, flexibility, cost reduction, automation, and performance.

When it comes to industry there are not many commercial solutions aimed at monitoring SDN networks. Some solutions such as Cisco ACI[1], VMWare's NSX[2], Nuage Networks' Virtualized Services Assurance Platform[3] but these solutions aren't for SDN monitoring. There are a few companies such as SevOne, which offer additional monitoring for commercial solutions such as ACI and NSX. But there is a clear deficit of monitoring solutions that support SDN monitoring.

In academia, there exists some projects that are built for OpenFlow-based SDN monitoring. The likes of OpenNetMon, FlowCover, SOFTmon, and the SDN Interactive Manager. OpenNetMon provides palpable evidence to validate network quality metrics on a per flow basis.[4] FlowCover is a low cost, high-accuracy monitoring solution, focusing on reducing communication costs.[5] SOFTmon is a Network Operating System (NOS) independent tool that operates with most available OpenFlow controllers. It serves to provide statistic and utilization charts at a flow level.[6] These charts rely on flow, switch, and port information. The SDN Interactive Manager offers users a solution they can use to monitor and manage their SDN networks, with an emphasis on how flow rule timeouts and polling intervals affect CPU and bandwidth utilization.[7]

1.3 Problem Statement

Given this new paradigm in networking, we want to explore how we can build an effictive monitoring solution for SDN. Given how different OpenFlow is from other protocols, and the centralization of the control plane, methods used in traditional network monitoring systems (NMS) are inefficient. Given the northbound interface applications can use to interact with the network, and some of the built-in features available in OpenFlow, we explore what monitoring capabilities are available to us for SDN.

1.4 Proposed Solution

SDLens is a software defined network (SDN) monitoring web application designed to operate with the OpenDaylight (ODL) controller. It provides real-time telemetry from OpenFlow switches, provides an interactive topology of the network, and many more intuitive features.

With SDLens, we can obtain information directly from the controller which allows for a single point of data collection. This is possible as the SDN controller abstracts the process of collecting monitoring data from the switches. This allows a single device the capability to hold the network state. With SDLens, the requirement is simplified by the primary interaction point which is our ODL SDN controller. From there we store telemetry into a MySQL database or poll ODL directly. Then SDLens will display this data to you regardless if the desired data is current, or historical. Using this application allows for a more simplified view of SDN monitoring and greater ease of use when compared to more traditional methods of network monitoring.

1.5 Accomplishments

SDLens was built from scratch and each team member improved their programming skills in order to achieve the final product. We focused on making SDLens as user friendly as possible to achieve scalable monitoring. In the span of eight months we were able to develop an application which provides the following key network monitoring features for OpenFlow devices:

- Dynamically generated topology page showing the active network
- Flow tracing feature providing a visualization of the path to carry traffic from any source host to a destination host

- Spanning Tree topology visualization
- Real time interface throughput
- Flow table statistics
- Port Counters
- Graphs to plot various statistics such as per switch active flow count and more

Our BIT-NET senior project was an excellent learning opportunity where our team was able to gain exposure of full stack web development with Flask, Python3, and JavaScript, and databases. We also obtained a greater understanding of network monitoring with OpenFlow and the usefulness of a single point of data collection, the ODL SDN controller. We also appreciate the fact that using OpenFlow as a lone source of network telemetry is not desirable to adequately understand network conditions. Additional sources of telemetry are needed. From the application development accomplishment standpoint, the best takeaway was creating Application Programmer Interfaces (API) and fine-tuning communication between components such as the MySQL database, agent process, and SDLens graphical user interface (GUI) at the front end. It was also useful learning how to use AJAX and jQuery to display dynamic text, images, and modifications to the topology page without reloading the entire webpage.

1.6 Overview of Remainder of Report

During the remainder of the report, we will provide some additional background information for SDN, and go in-depth into the SDLens application. We will discuss the application architecture, detail features like the dynamic topology page, statistics tables, performance graphs, startup scripts, and our flow tracing capabilities. We will then finish off by reflecting on our solution, the lessons learned throughout this project, and the suggestions we have for future projects.

2 Technical Section

2.1 Background and Terminology

SDN consists of three layers: Application Layer, Control Plane, and Data Plane. The data plane consists of networking equipment that forms the underlying network. This is where forwarding of network traffic is done, I.e. OpenFlow Switches in our Mininet test environment.

The Control layer is where the SDN controller(s) reside. The controller determines what information to fetch and maintains network state. This layer is used as an abstraction between the network application logic, and the underlying network. To bridge the application to the network, the controller provides two types of interfaces. The northbound interface enables communication between the application layer and the controller, whereas the southbound interface is used to communicate to the data plane devices.

The Application layer is where engineers can build their applications to manage the network. Our app SDLens exists in this space. Given the northbound interface provided by the controller, this simplifies the process of developing applications for the network.

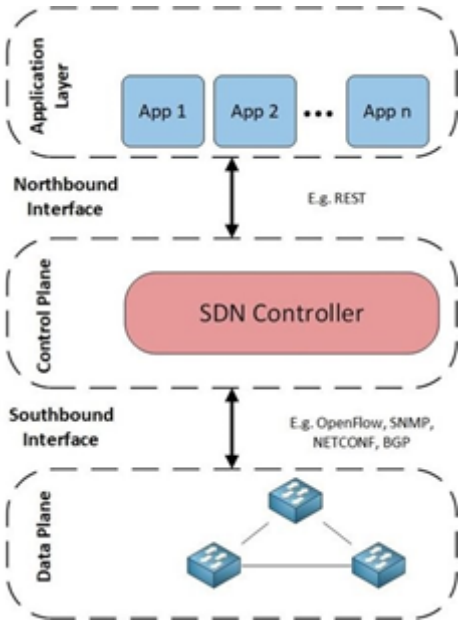


Figure 1: SDN Architecture

The SDN networks we used in our project are based on the OpenFlow protocol. OpenFlow is a protocol that defines the southbound communication between the controller and the switches, and it also specifies the forwarding logic on the switches themselves. OpenDaylight (ODL) is a popular open-source SDN controller, that offers a wide range of support for different applications and protocols. Given that OpenFlow is still the most popular SDN protocol to date, our project uses the core OpenFlow modules provided by ODL to control our networks.

2.2 Project Details

During the remainder of the report, we will provide some additional background information for SDN, and go in-depth into the SDLens application. We will discuss the application architecture, detail features like the dynamic topology page, statistics tables, performance graphs, startup scripts, and our flow tracing capabilities. We will then finish off by reflecting on our solution, the lessons learned throughout this project, and the suggestions we have for future projects.

3 Application Architecture

SDLens is designed to solely live in the application layer of the SDN stack. All interactions between the application are the network are done through the network controller's northbound interface. Having these common APIs defined provides a consistent method for SDLens

to query for network information, which lowers application complexity. SDLens is mainly written in Python, with some JavaScript used in the front-end for visualization purposes.

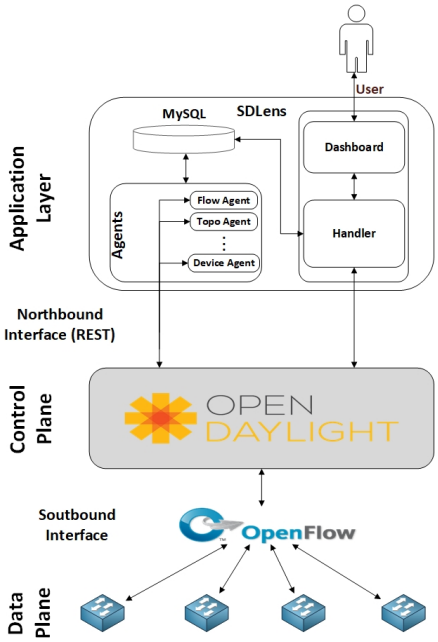


Figure 2: SDLens Architecture

SDLens can be broken down into three main components, the web application, the monitoring agents, and the database. The web application is the main component users interact with, it provides a GUI for administrators to use and the logic to visualize the monitoring functionality. The monitoring agents periodically query the network for various statistics and store them in the database. Our database contains all the state and historical data necessary to run our monitoring functions.

3.0.1 Web Application Architecture

The web application is the main selling point of SDLens, since it is the component operators would use to monitor their networks. As a result, we put a lot of emphasis in keeping the web application efficient and easy to navigate.

Since our application is written in Python, we used Flask as our web development framework. Flask is an open-sourced, minimal, Python-based framework for web development.[8] We chose Flask because its simplicity and extensibility which made it easy for us to adapt it for SDLens. Leveraging Flask, figure 2 shows a common work flow for a web app interaction.

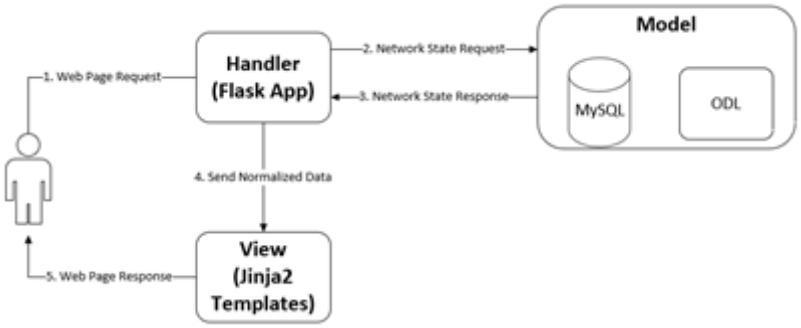


Figure 3: Web Application Work Flow

This application flow roughly follows the Model-View-Controller (MVC) design pattern. In this design pattern, our Flask application acts as the controller, receiving web requests from the user and retrieving the appropriate data from our model (the data stored in the DB and/or ODL). Once our handler normalizes the data from the model, we can dynamically generate our webpages in our views. In this case, our views are Jinja2 templates rendered by Flask. Jinja2 is a Python based templating language. This allows us to dynamically generate code, such as HTML elements based on the variables we pass to Jinja2. Once these webpages are rendered, we can send them as a response to the user.

Since many monitoring capabilities require a visualization component, such as graphs or topologies, we used JavaScript for client-side visualizations. Specifically, we leveraged the vis.js libraries for our monitoring components. The vis.js libraries are open-sourced and useful for network and graphing capabilities.[9] We were able to utilize these libraries for our dynamic topologies as well as our performance graphs.

3.0.2 Agent Architecture

Although SDLens users do not directly interact with the monitoring agents, they play a crucial role in regularly collecting monitoring data for many of our functionalities. Our application is currently composed of several agents that are responsible for collecting data on components like the network topology, port counters, and flow rules to name a few. Our agents are meant to be modular, meaning that developers should be able to contribute more agents without having to refactor the code.

When looking at the application logic for our agents, they all follow a similar pattern:

- Send an API request to the SDN controller for network data
- Parse the API response for important information
- Store this normalized data in our SQL database

Given this similarity, every agent inherits methods to execute this pattern while only having to tweak parts specific to the data they're monitoring. This minimized the amount of repeated code between agents and gave our main application a predictable method to run this pattern for every agent. Having this predictable method made it easy to introduce features like multi-threading to increase our application efficiency.

3.0.3 Database Architecture

The DB is the component that both the web application and the agents rely on for their functionality. Give that database design is outside of our team's area of expertise, our goal was to keep the database structure as simple as possible, while still giving us the support that we needed for our application. We have tables that keep tracks of components:

- switches and hosts
- links
- OpenFlow tables
- port counters
- flow summaries
- node information

These tables are intuitive, and only contain the information our application requires. Since we kept our database simple, we are not using any of the relational features of SQL. Tables that track statistics such as port counters also contain a timestamp, this is useful for looking at historical data or building performance graphs.

4 Key Application Features

4.1 Interactive Topology Page

The topology page is designed to allow users to easily peek into their SDN to gather telemetry on all their network elements. There are two main areas of interest which display statistics on the topology page, the Topology View Panel, and the Information Panel. The Topology View Panel displays the interactive topology as can be seen in the center of Figure-4 (SDLens Topology Page). The Information Panel will display important data when an event is triggered. More information pertaining to both panels will be expanded on in the following subsections.

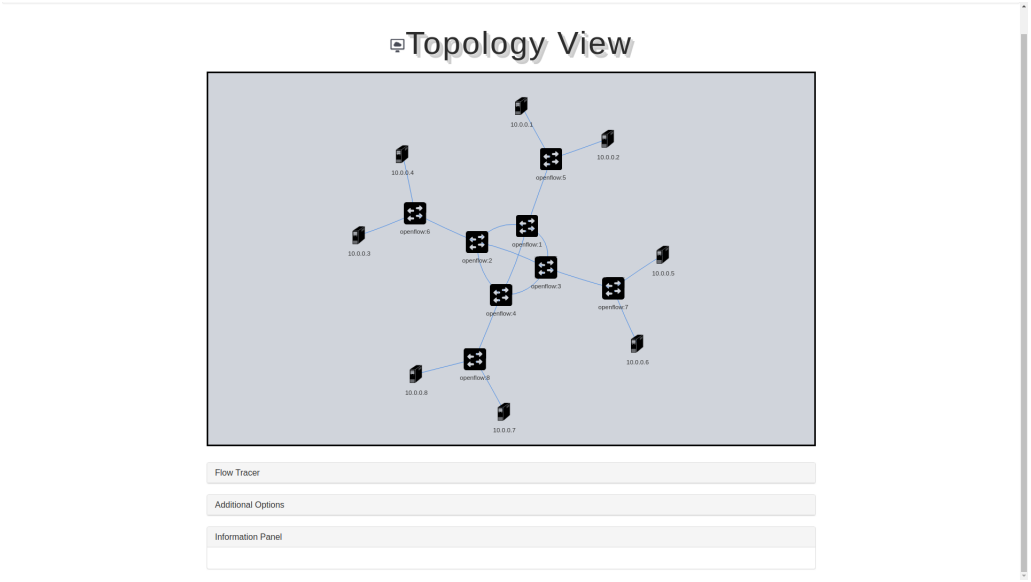


Figure 4: SDLens Topology Page

4.1.1 Topology Page Features

Within the Topology View Panel, you can hover over devices or click devices to obtain more information about them. Depending on if the user hovers over a node or clicks on a node, different events happen.

4.1.2 Throughput and Basic Flow Information

Hovering over hosts, switches or links will display information about the element of interest. For example, hovering over a switch shows throughput statistics for each interface and as well as real-time nodal flow information. Figure-5 (Throughput and Basic Flow Information) illustrates this action.

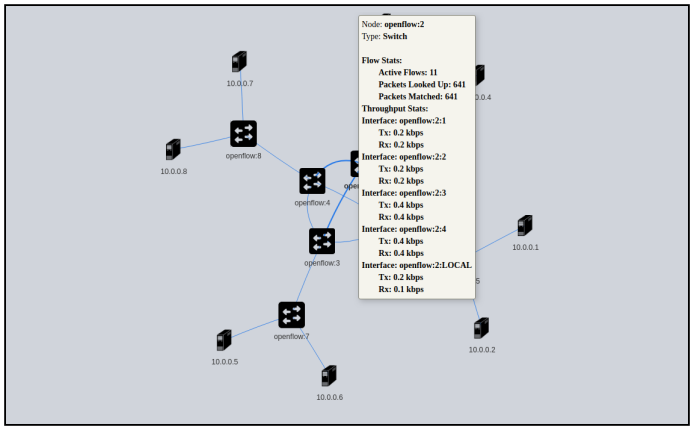
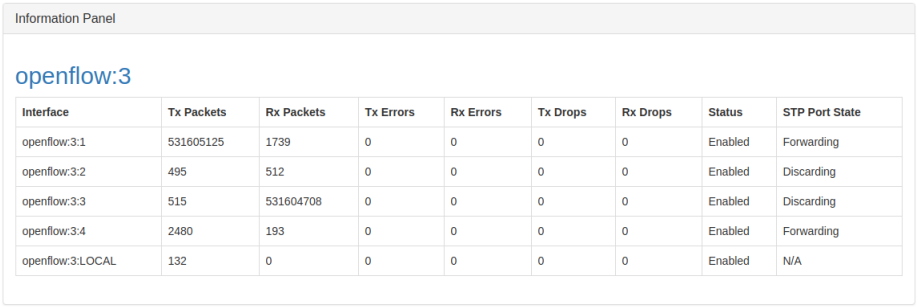


Figure 5: Throughput and Basic Flow Information

4.1.3 Switch and Host Information

Clicking on a switch will provide information such as switch port counters, errors, Spanning Tree state, port status, etc.

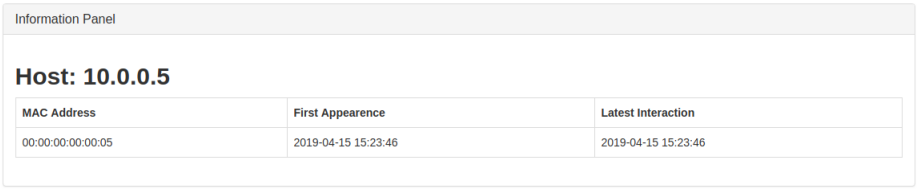

Information Panel

openflow:3

Interface	Tx Packets	Rx Packets	Tx Errors	Rx Errors	Tx Drops	Rx Drops	Status	STP Port State
openflow:3:1	531605125	1739	0	0	0	0	Enabled	Forwarding
openflow:3:2	495	512	0	0	0	0	Enabled	Discarding
openflow:3:3	515	531604708	0	0	0	0	Enabled	Discarding
openflow:3:4	2480	193	0	0	0	0	Enabled	Forwarding
openflow:3:LOCAL	132	0	0	0	0	0	Enabled	N/A

Figure 6: Switch Clicked

Clicking on a host will display a timestamp of when that joined the network and the latest interaction with the network it has had.


Information Panel

Host: 10.0.0.5

MAC Address	First Appearence	Latest Interaction
00:00:00:00:00:05	2019-04-15 15:23:46	2019-04-15 15:23:46

Figure 7: Host Clicked

4.1.4 Spanning Tree Topology

If you would like to see the current Spanning Tree Topology, then this is possible by clicking the STP Topology button.

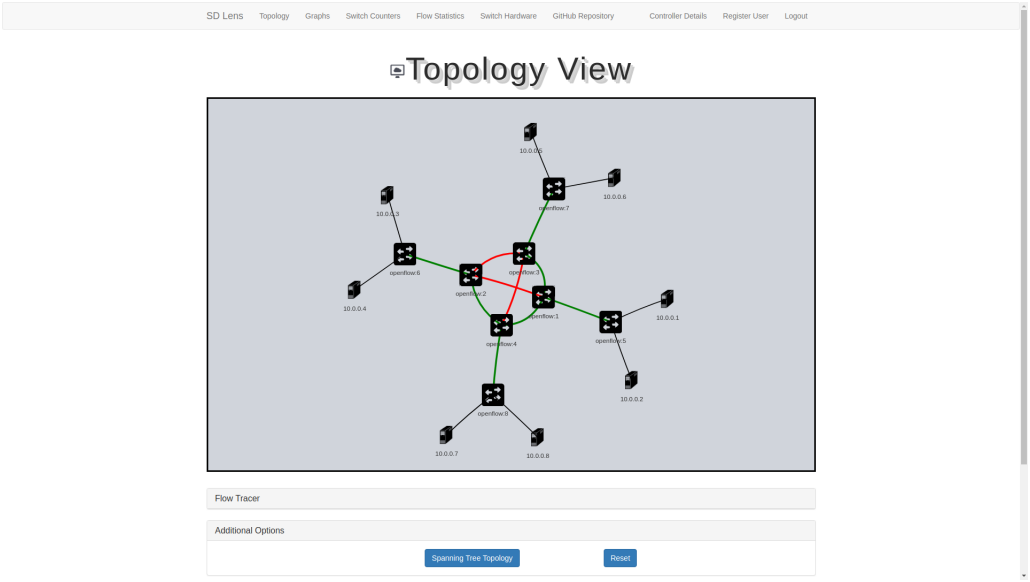


Figure 8: Spanning Tree Active

The last feature part of the topology page is called flow tracer and has its own space below dedicated to its features and design.

4.2 Topology Page Architecture and Design

As mentioned in the application architecture section, SDLens uses an MVC design pattern. The topology page makes extensive use of APIs within the Flask Handler in order to display information dynamically and asynchronously. This data is retrieved from the MySQL database, however other pages such as the port counters page retrieve data from ODL directly. Most actions requiring a button press trigger a JavaScript function calling a specific API. Depending on the nature of the data needed, a POST or GET request is made to the Flask API handle serving that request. A POST request is used for instances where data is needed on a unique element. For example, to determine the throughput and flow information while hovering a switch, a POST request is made containing the name of the switch to be queried. In some cases, an API Uniform Resource Identifier (URI) allows for variable items to be placed within itself. In the case of the flow tracer feature, this is the case and the source and destination hosts are variably included as part of the URI.

4.3 Statistics Table

Within SDLens, we have several pages that represent many statistics in tables (see Figure 9). This includes tables for the port counters, OpenFlow tables, and switch hardware information. These tables show the relevant information the users want for every device on the network.

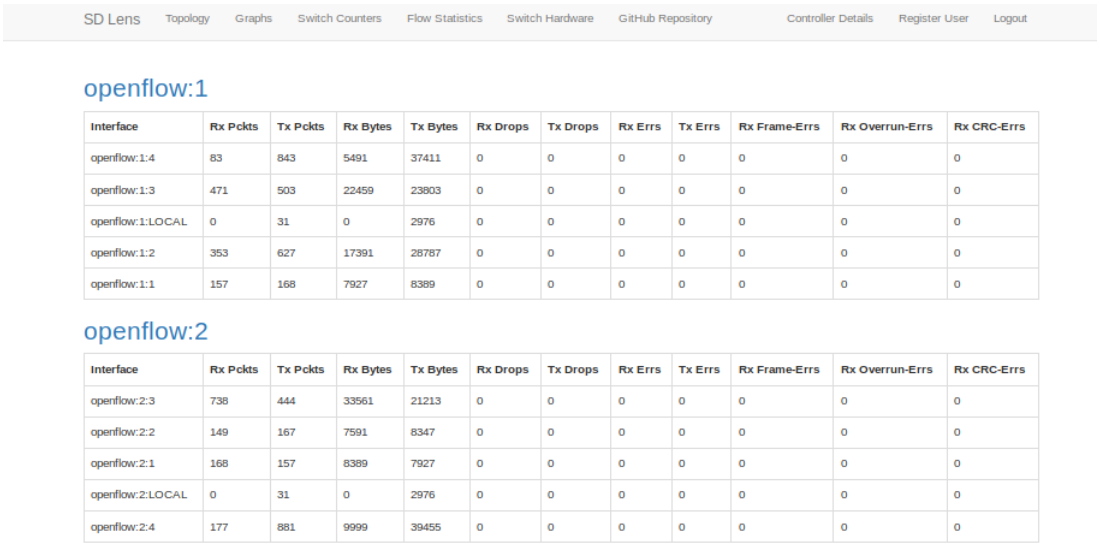


Figure 9: Port Counters Table

The process for generating these tables is very similar, so we only need to discuss the application logic once. Following the MVC logic mentioned earlier, when a user requests one of these pages, the Flask app will query the SDN controller for the appropriate data. When the controller responds, SDLens will build a data structure of this data that it can then render in Jinja2. With the Jinja2 rendering, these tables are dynamically built for every switch we have in our data structure.

The port counters table provides interface statistics on packets and bytes received/transmitted, as well as the number of drops or errors that have occurred. Our flow statistics table provides a comprehensive look into the flow rules present on a device. It provides flow-IDs, flow rule priority, packet/byte counts for the rule, how long the flow rule has been present for, its timeout values, and the corresponding match and action rules. The switch hardware page gives some information about the actual interface hardware. It provides the port ID, port number, supported speed, and the MAC address associated to that port.

4.4 Interface Graphs

The interface graphs take advantage of having a database more than any other function. These graphs display four interface statistics based upon packet count; the Received Packets, Transmitted Packets, Received Packet Drops, and Transmitted Packet Drops. In order to display information over time, it is required to maintain that information, and unlike most other features, the database is required for this. For the graphs to function properly, there are two components required, not including the database; the `port_counter_agent` and the `gen_graphs` function. These two functions collect and format the data so that it can be handed to the webpage for the JavaScript based graphing function to display it.

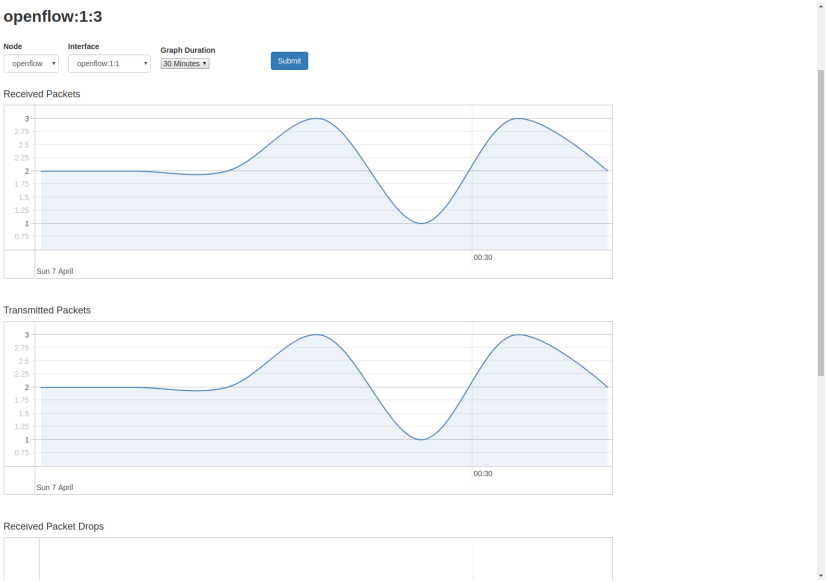


Figure 10: Received and Transmitted Packets

The `port_counter_agent`, as described above, is used to collect port information from the controller and store it in the database. This is a crucial function for the graphs, as they only display information based on a specific interface.

The main portion of the graphs comes from the `gen_graphs` function, which collects the information from the database and processes it, so that it can be outputted to the JavaScript function. The function takes in three parameters, the `node`, the `interface`, and the `time`. With these three inputs, the appropriate SQL query can be made to retrieve the interface data from the database, in the set timeframe. The data is then separated, the first data set in the group is used as the baseline for the second set, and so no operation is made upon it. Once the first set of data has been stored, after a baseline is set, the following data sets are subtracted from the current one, as this shows the change between the two sets. This prevents the graph from just being an increasing line, and this is further explained by the following pseudo code:

Data: SQL Query Response
for *Data Points in* **Data:**
do
  Store first points in `pointDictionary`;
  if *`pointDictionary` exists* **then**
    Store `PREVIOUS` as `OLD`;
    Store `DIFFERENCE` = `CURRENT` − `OLD` ;
  end
  Append `pointDictionary` to Array;
end

Algorithm 1: Generating Graph Points

At this point, the data is stored in a dictionary, which is returned when the function completes. This data is then used by the written out in a `for loop` by a `Jinja2` function which displays the timestamp along the X-axis, and the corresponding value along the Y-axis. Since the entire dataset is returned with all four value sets (Received Count,

Transmitted Count, Received Drops, Transmitted Drops), the `for loop` is used to sperate the data so that only the one of the four data sets is displayed.

4.5 Startup Script

One of the objectives of this application was to be able to start the program and its components with one file. This file became the `startup.py` file which starts both the `agent` process and the `web application` portion of the SDLens. The startup program can be broken into three different parts, the administrative information, clearing the database, and starting the application.

The first part, obtaining the information required for the database, application, and controller is done as a check to confirm that the correct credentials exist so that the application can talk to the appropriate devices. If these values are incorrect or the file doesn't exist, the script will take the user through the process of creating or changing the values.

After it has been confirmed that the credentials file exists, the application must check if the database needs to be cleared, as the program will fail if the topology has changed between runtimes. If the database does not need to be cleared, then the program proceeds normally, otherwise it begins the clearing process. To do this without destroying the entire database, a special query needed to be created which selected all the relevant OpenFlow tables and removes them. In doing so this would prevent the users table, which stored login information from being dropped, thus allowing users to maintain their credentials while the rest of the stored information is removed from the database.

The final step of the startup script is to start both the `agents` and `web application` programs at the same time. This allows both to be run in the background so that each program can be run simultaneously. The final part of this section is how the entire application shuts down. In order to stop the script and prevent it from leaving other processes running, it uses the `grep` function to find the Process ID (PID) of each application, and then kill each process. This is done to allow the application to have a clean and consistent start up, and to not leave processes running in the background.

With more time, a few changes to the script would be made. Having it so that startup takes in arguments rather than sending the user through a set of menus would speed up the startup process. Along with that, have the program check if the credentials file exists, rather than relying on the user to confirm that fact. Finally make a more efficient system to clearing the database, as the current method could be improved upon.

4.6 Flow Tracer

The flow tracer feature allows users to visualize the path used for two host elements to communicate with one another. The user is presented two dropdown menus which are loaded via Jinja2, following the MVC design pattern mentioned earlier. Within this tab, you select the source host, destination host, and press trace. JavaScript is used for front end validation. If the user has not selected a valid option, both the trace and swap path buttons will remain greyed out as is the case in Figure-11 (Flow Tracer Options).

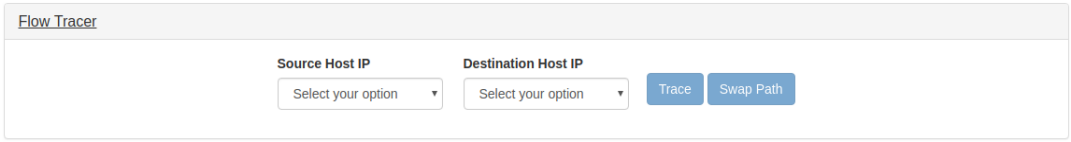


Figure 11: Flow Tracer Options

The Topology View Panel will display the path taken from the source host to destination host.

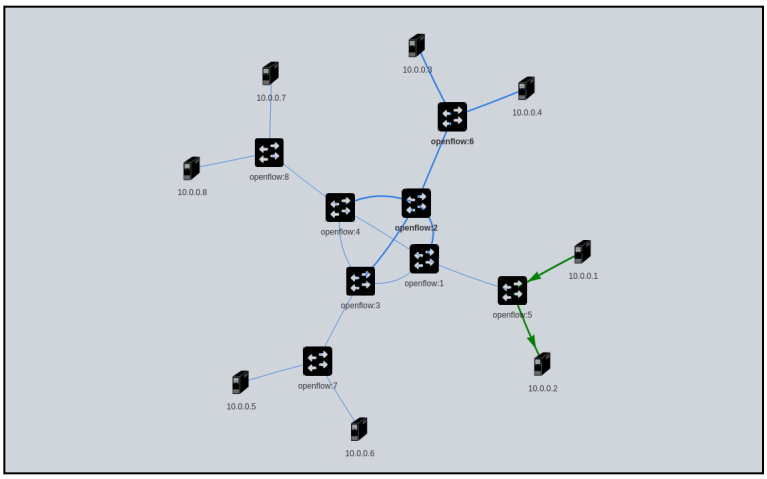


Figure 12: Flow Tracer Topology View

The information panel displays information of the switches traversed and the flow rule used along with information on that particular flow.

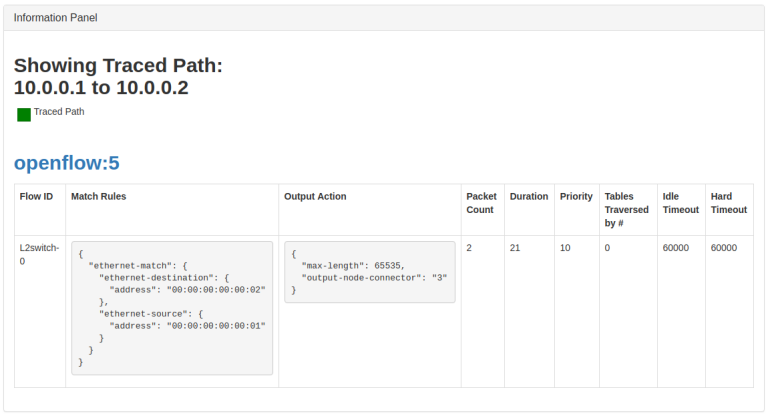


Figure 13: Flow Trace Information Panel View

4.6.1 Flow Tracer Implementation Details

The topology page gets flow trace data the same way it does for other data by using an API. The logic to get a flow path is done in the backend in Python by analyzing flow rules on transient switches. The IP addresses passed to the backend for the source and destination are first converted into the names of the hosts consistent with what ODL calls

them. The names of hosts contain their MAC address. From this information, the ingress and egress switches are found from the data known about the OpenFlow topology. From here, we can determine the exact flow used on each transient switch by the match rule for the source and destination MAC addresses. The flow rule action provides the egress interface used to reach the next location. This process is completed until no further rules can be found or the destination host is reached.

5 Conclusion and Recommendations

SDN monitoring introduces many unique opportunities and challenges compared to traditional network monitoring. Having a well-defined northbound API accelerates the application development process and it also helps scale monitoring solutions. But there is also a lack of flexibility as you are limited by the features supported on the controller and the capabilities of OpenFlow. SDLens attempts to get the most out of the monitoring capabilities available. Our application tracks topology state, port counters, and flow statistics. With this information, we allow users to monitor their network topologies, performance, interface health, flow rules, and track traffic flows through a user-friendly dashboard.

Although there is excitement around SDN, it is important to realize that new technologies are never perfect when first released. While exploring ODL as our SDN controller, we noticed several odd behaviors within ODL. Many of which were discovered through the usage and testing of SDLens. For one, ODL does not necessarily create an optimal layer 2 network. When ODL implements the spanning-tree protocol, it does not strategically pick a root switch which often leads to suboptimal paths. In addition, layer 2 ODL networks simply flood traffic by default, requiring the user to edit the controller start-up settings if they want stateful flows. Finally, ODL will sometimes return problematic data for standard API calls. We have run into instances where ODL will say a switch contains zero flow rules, when it is clear there are more than what is reported.

In practice, many tools that obtain network telemetry are used in concert to interpolate an accurate view of the network. Relying on OpenFlow for telemetry alone is insufficient. With traffic rates growing at the pace they are today; it is increasingly difficult to sample data effectively and accurately. For this reason, multiple sources of telemetry such as sFlow(Sampled Flow) or NETFLOW, and even SNMP are commonly are used together.

Presently, SDLens only monitors ODL driven networks that use OpenFlow. A logical next step for an application like this would be to introduce the ability to manage flows and the network from the dashboard. For future projects, we recommend going beyond simple OpenFlow monitoring. Other options include building a streaming telemetry solution, an in-band telemetry solution, or looking at other emerging SDN technologies such as P4.

This project was an interesting exercise that allowed us to dive deep in SDN and network monitoring. Building an application using the controller's northbound interface has showed us the potential for SDN. Encountering the various limitations of OpenFlow throughout the project has broadened our understanding of what the common roadblocks are with regards to building an SDN monitoring application. With that said, we understand why the OpenFlow has lost momentum since the specification was last updated in 2015. We are excited to see the direction of SDNs in the future and humbled to have dipped our toes into SDN monitoring research.

References

[1] "Cisco application centric infrastructure fundamentals," Mar 2019. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/aci-fundamentals/b$_ACI - Fundamentals/b_ACI - Fundamentals_chapter_0$1110.*html*

[2] "Flow monitoring." [Online]. Available: https://docs.vmware.com/en/VMware-NSX-for-vSphere/6.4/com.vmware.nsx.admin.doc/GUID-86609A0C-00DA-45CC-A5C6-068687D0937B.html

[3] "Virtualized services assurance platform." [Online]. Available: http://www.nuagenetworks.net/products/virtualized-services-assurance-platform/

[4] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–8.

[5] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Flowcover: Low-cost flow monitoring scheme in software defined networks," in *2014 IEEE Global Communications Conference*. IEEE, 2014, pp. 1956–1961.

[6] M. Hartung and M. Körner, "Softmon-traffic monitoring for sdn," *Procedia Computer Science*, vol. 110, pp. 516–523, 2017.

[7] P. H. Isolani, J. A. Wickboldt, C. B. Both, J. Rochol, and L. Z. Granville, "Interactive monitoring, visualization, and configuration of openflow-based sdn," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 207–215.

[8] "Welcome to flask." [Online]. Available: http://flask.pocoo.org/

[9] "vis.js." [Online]. Available: http://visjs.org/