POX-PLUS: An SDN Controller with Dynamic Shortest Path Routing

Muteb Alshammari
Faculty of Computing and Information Technology
Northern Border University
Rafha, Saudi Arabia
muteb.alshammari@nbu.edu.sa

Abdelmounaam Rezgui
School of Information Technology
Illinois State University
Normal, IL, USA
arezgui@ilstu.edu

2020 IEEE 9th International Conference on Cloud Networking (CloudNet) | 978-1-7281-9486-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/CLOUDNET51028.2020.9335792

***Abstract*—Routing in SDNs exploits the controller's global view and computes paths either using a single-source shortest path algorithm (e.g., Dijkstra, Bellman-Ford) or an all-pairs shortest path (APSP) algorithm (e.g., Floyd-Warshall). Existing APSP routing algorithms for SDNs have substantial performance limitations in handling changes in the routes due to link deletion (failure) and link insertion (recovery). In this paper, we present POX-PLUS, a new SDN controller based on the popular POX controller. POX-PLUS includes a new routing module called DR-APSP (Dynamic Routing based on All Pairs Shortest Paths) that computes and efficiently maintains shortest paths between nodes in the SDN.**

***Index Terms*—Software-defined networks, SDN, Dynamic shortest paths, APSP, Routing**

I. Introduction

Several routing solutions (applications or systems) for SDNs have been proposed in the literature, e.g., [1]–[10]. Each application accomplishes a number of tasks. For example, a shortest path routing application finds the shortest paths to forward packets through the network. In this paper, we present POX-PLUS, a new SDN controller based on the popular POX controller. POX-PLUS includes a new routing module called DR-APSP that computes and maintains shortest paths between nodes in the SDN. DR-APSP maintains flow rule tables in the data plane. It updates only the outdated flow rules resulting from a network link update (insertion or deletion). Instead of flushing all flow rule tables, DR-APSP detects the changes and removes only obsolete flow rules.

II. Architecture and Model

Figure 1 highlights the key elements of the architecture of POX-PLUS. It is composed of three main components:

- **DR-APSP:** this is the core routing module based on our dynamic routing APSP algorithm (explained next).
- **POX Controller** is the the original POX controller which is now also responsible to update DR-APSP about the original topology and any subsequent changes.
- **Data Structures** are used to store information about the topology and its shortest paths. These include the *Topology Graph*, *Hosts*, and a shortest paths trees *(SPT) Forest* (explained in the next section).

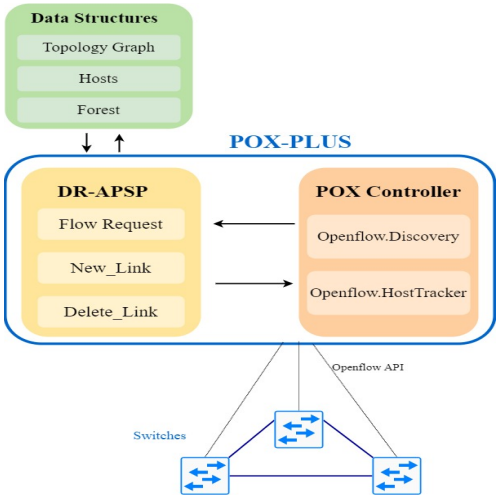

Figure 1: POX-PLUS Architecture

We represent the topology of an SDN as a dynamic directed graph $G = \{V, E, W\}$ that supports the insertion and deletion of edges and vertices. V is the set of switches and E is the set of links. We extend this model to also include the switches' ports as follow:

POX-PLUS maintains a forest T of shortest paths trees (SPT). We maintain an SPT t for every switch. We use the notation $t(s1)$ for the SPT rooted at switch $s1$. We also use *Hosts*, a set of mappings from every host to a tuple of the connected switch and port number.

Definitions:

- **Affected switch:** A switch sw is said to be *affected* if sw changes its parent in at least one SPT after a link operation[1].
- **Path:** A path is a list of switches that forms a shortest path between two switches.
- **Affected path:** An *affected path* is a path where, at least, one edge is changed because of an edge operation.
- **Affected flow:** A flow is said to be *affected* if it is a flow entry on a switch that is part of an affected path.
- **Reachability test:** This consists of sending a packet between every two hosts.

978-1-7281-9486-8/20/$31.00 ©2020 IEEE

[1]We use the terms edge and link interchangeably.

III. Approach

The core module of POX-PLUS is the algorithm DR-APSP which provides routing for dynamic SDN topologies. DR-APSP is incremental, i.e., does not recompute shortest paths from scratch when edges join or leave the topology. In the next sub-sections, we elaborate on the operation of DR-APSP.

A. Link Deletion

Link deletion (e.g., link failure) is handled as follows (Algorithm 1). The controller detects any link *event* in the network (either deletion or insertion). Upon detecting a link deletion (say (x, y)), DR-APSP starts a loop over all SPTs. The loop consists of two main phases. In the first phase, the algorithm detects the affected switches in each SPT. Then, for each affected switch, we remove affected flows that chose the affected edge (deleted edge). For each affected switch, we remove every flow rule with a source host that matches one of the hosts connected to the source of the current tree and a destination that matches one of the hosts connected to any switch in the sub-tree $t(y)$. This phase is crucial since future packets may take a different path.

The controller will issue new flow rules (new rules to replace deleted flow rules) upon future requests. We note here that we do not replace the affected flow rules. However, we delete the affected flow rules and leave the replacement for any future request. In the second phase (Line 12), we update the current SPT for future requests using our APSP approach described in [11]. We note that the algorithm updates its SPT only after it removes affected flows from the corresponding affected switches. This is due to the fact that, after deleting an edge (say (x, y)), affected switches may choose new (totally different) paths, and we need to ensure that the flow rules of affected switches are deleted. Because we always choose the shortest path, we must remove any outdated flow rules (outdated because they are no longer part of the shortest path due to an update operation).

Algorithm 1 Deleting link (x, y)

```
1: procedure DELETE_LINK(x, y)
2:     for every v ∈ V do
3:         t = T(v)                          ▷ t is a SPT rooted at vertex v
4:         if (x, y) ∈ t then
5:             path = {t.source, ... , x}    ▷ switches in the path v...x
6:             x_hosts = every host connected to x
7:             affected_switches =every switch sw s.t. sw ∈ t(y) ∪ {y}
8:             affected_hosts = every host connected to affected_switches
9:             for every u ∈ affected_switches ∪ path do
10:                remove every flow in u i.e. coming from x_hosts to affected_host
11:            Decremental_Algorithm(x, y, ∞, t)
```

B. Link Insertion

We now show how DR-APSP handles link insertions in SDN (Algorithm 2). The algorithm starts when it receives a link *event* of type insertion (say (x, y)). First, it enters a loop over all vertices. The loop consists of two main steps. The first step (Line 6) is to update the current SPT (at the current vertex) after encountering a new link using our APSP approach described in [11] . The second step is to remove affected flows from affected switches. For each affected switch, we remove every flow rule with a source host that matches one of the hosts connected to the source of the current tree and a destination that matches one of the hosts connected to any switch in the sub-tree $t(y)$. Therefore, we delete all affected flow rules that are not part of any shortest path after the update operation.

Table I: Data sets used in the evaluation.

Data set name	Figure	# of switches	# of hosts	# of links
SYNTH1	2	100	11	129
AS	3	100	2	205

Algorithm 2 Inserting link (x, y)

```
1: procedure NEW_LINK(x, y)
2:     for every v ∈ V do
3:         t = T(v)                          ▷ t is a SPT rooted at vertex v
4:         if W(x, y) + d(x) < d(y) then
5:             path = {t.source, ... , x}    ▷ switches in the path v...x
6:             Incremental_Algorithm(x, y, t)
7:             x_hosts = every host connected to x
8:             affected_switches = every switch sw s.t. sw ∈ t(y) ∪ {y}
9:             affected_hosts =every host connected to affected_switches
10:            for every u ∈ {affected_switches ∪ path } do
11:                remove every flow in u i.e. coming from x_hosts to affected_host
```

IV. Implementation and Evaluation

We implemented our routing approach as a new SDN controller (named POX-PLUS) based on the popular SDN controller POX. POX is written in Python [12] and supports the OpenFlow protocol [13]. To evaluate our implementation, we used Mininet [14] which is widely used in the literature to test SDN prototypes. We note that every reported result in this paper is an average of three runs. The code for our implementation is made available online on GitHub [15].

A. Network Topology

To evaluate routing in POX-PLUS, we consider two examples of network topology (see Table I):

1) *Synthetic data* (**SYNTH1**): This is a network that starts as a linear network of 100 switches, 11 hosts, and 129 randomly generated links. We first generated a linear topology (e.g., s1 ↔ s2, s2 ↔ s3, etc.). Then, we added the rest of the links. By doing this, we avoided cycles before we could detect all hosts and their connections.
2) *Autonomous Systems* (**AS**): In the second set of experiments, we built a network based on the graph of an actual autonomous system (AS) collected from [16]. In this scenario, switches represent autonomous systems. The network consists of 100 switches, 205 links, and 2 hosts. We added the hosts to the two ends of the network.

In the evaluation, we started an instance of our POX-PLUS controller and used a Python script that builds the desired network in Mininet.

The script starts by building the given topology without cycles and tests for reachability to populate the required data structures. Then, we added the rest of the links that introduce cycles in the topology. Next, we removed all flow rules from all switches and measured four main performance indicators:

- **init_time:** The time to initialize flow tables at switches. Here, we record the overall time it takes to send a data packet from every host to every other host in the network.
- **add_time:** The time that the considered application takes to send a data packet between every two hosts after adding links to the network.
- **delete_time:** The time that the considered application takes to send a data packet between every two hosts after deleting links from the network.
- **selected_paths_time:** The time that the considered application takes to send a data packet between every two hosts after populating the flow tables of all switches.

We compared the performance of POX-PLUS's routing module to three routing schemes provided by the POX controller, namely: *l2_learning, l2_multi, and l3_learning*. All three schemes require a topology with no cycles. Therefore, the developers of POX implemented a specially designed component called *spanning_tree*. This component prevents cycles by constructing a spanning tree for the network topology and then blocking links that are not in the spanning tree edges when forwarding to all ports. We note here that our POX-PLUS controller does not require the spanning tree. It only requires the population of the *Hosts* data structure and hosts' ARP tables before the formation of cycles. In other words, POX-PLUS needs to know the connections between hosts and switches as well as the switches' port numbers that connect to hosts. *l2_learning* and *l3_learning* use an idle timeout for installed flow rules to overcome the links update operations. Without the use of idle time, they may not recover after deleting a link. In fact, packet loss rate increases as we increase the idle time since the controller assigns new rules only after a switch requests one (which occurs after deleting affected flow rules). The idle time lets switches remove flow rules upon the expiration of the idle time. *l2_multi* removes all flows from all switches upon receiving a link operation.

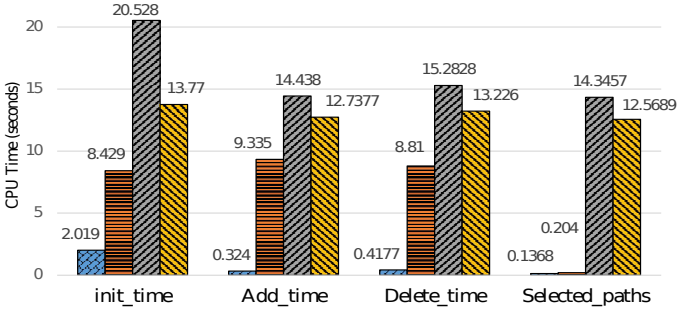

Figure 2: Experiments on the **SYNTH1** data set with 100 switches, 129 links, and 11 hosts.

B. Performance Evaluation

We now present the performance of DR-APSP compared to the three previously discussed approaches. As we explained earlier in Section IV-A, we consider two main data sets:

- **SYNTH1:** Figure 2 shows the CPU time that each controller takes after running the test script. It shows that both DR-APSP and *l2_multi* (both use shortest paths) outperform other approaches. First group of Figure 2 (i.e., init_time) shows that at initialization (that is computing from scratch), DR-APSP is 4 to 10 times faster than other approaches. *l2_multi* comes next with a factor of 1.5 to 2.4 faster than *l2_learning* and *l3_learning*
  The second and third groups of Figure 2 (i.e., Add_time and Delete_time) show the effect of links updates. When updates occur in the network, DR-APSP is the only approach that maintains shortest paths and flow rule tables instead of recomputing paths and repopulating flow tables from scratch. The cost of DR-APSP is a fraction of the cost of initialization (that is starting from scratch). We note here that the cost of other approaches is similar to the initialization since they start from scratch. The fourth group of Figure 2 (i.e., Selected_paths) shows the cost of the selected paths of each approach. We see that *l2_learning* and *l3_learning* have a high cost since they apply a short idle time for flows in flow tables. Idle time needs to be short to recover quickly.

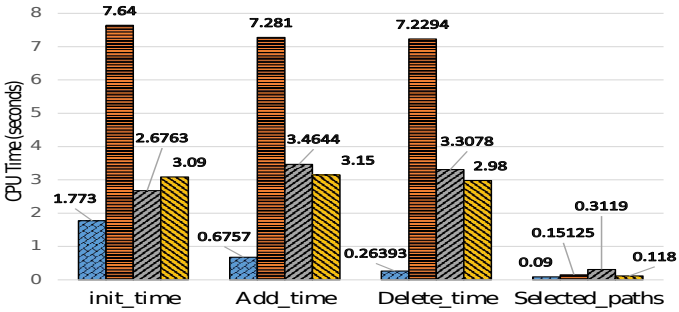

Figure 3: Experiments on the AS data set with 100 switches, 205 links, and 2 hosts.

- **AS:** Figure 3 shows the cost of the considered approaches in an autonomous system network. The same pattern repeats here with one particular difference which is that *l2_multi* has the highest cost compared to other approaches. *l2_multi* uses the Floyd-Warshall algorithm which uses a matrix of size N×N (where N is the number of switches). This led to a higher cost in terms of recomputing shortest paths.

V. Conclusion

We presented POX-PLUS, a new SDN controller based on the popular POX controller. POX-PLUS includes a new routing module called DR-APSP that computes and maintains paths between nodes in the SDN. DR-APSP provides SDN routing based on dynamic shortest paths. Our results show that DR-APSP can significantly improve the performance of SDN routing in terms of maintaining the shortest paths and flow rule tables after an update operation without recomputing paths.

REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*. USENIX Association, 2010, pp. 281–296.

[2] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks." in *NSDI*, vol. 10, 2010, pp. 249–264.

[3] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM Sigcomm Demo*, vol. 4, no. 5, p. 6, 2009.

[4] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild." *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[5] S. Waleed, M. Faizan, M. Iqbal, and M. I. Anis, "Demonstration of single link failure recovery using bellman ford and dijikstra algorithm in sdn," in *2017 International Conference on Innovations in Electrical Engineering and Computational Technologies (ICIEECT)*. IEEE, 2017, pp. 1–4.

[6] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Goncalves, R. Figueiredo, D. Morris, M. Pickavet, and P. Demeester, "Implementing quality of service for the software defined networking enabled future internet," in *The European Workshop on Software Defined Networking (EWSDN 2014)*. IEEE, 2014, pp. 49–54.

[7] K. Jeong, J. Kim, and Y.-T. Kim, "Qos-aware network operating system for software defined networking with generalized openflows," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 1167–1174.

[8] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém, "Control of multiple packet schedulers for improving qos on openflow/sdn networking," in *2013 Second European Workshop on Software Defined Networks*. IEEE, 2013, pp. 81–86.

[9] C. Zhang, X. Wang, F. Li, and M. Huang, "Nnirss: neural network-based intelligent routing scheme for sdn," *Neural Computing and Applications*, vol. 31, no. 10, pp. 6189–6205, 2019.

[10] W. Qi, Q. Song, X. Kong, and L. Guo, "A traffic-differentiated routing algorithm in flying ad hoc sensor networks with sdn cluster controllers," *Journal of the Franklin Institute*, vol. 356, no. 2, pp. 766–790, 2019.

[11] M. Alshammari and A. Rezgui, "An all pairs shortest path algorithm for dynamic graphs," *International Journal of Mathematics and Computer Science*, vol. 15, no. 1, pp. 347–365, 2020.

[12] M. McCauley, "Pox controller." [Online]. Available: https://github.com/noxrepo/

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

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

[15] "Pox-plus controller." [Online]. Available: https://github.com/muteb-nbu/POX-PLUS.git

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.