IET Wireless Sensor Systems

Review Article

IET Journals
The Institution of Engineering and Technology

Investigation of data forwarding schemes for network resiliency in POX software defined networking controller

ISSN 2043-6386
Received on 30th September 2014
Revised on 7th January 2015
Accepted on 2nd February 2015
doi: 10.1049/iet-wss.2014.0107
www.ietdl.org

Raviraj Vaghani, Chung-Horng Lung ✉

Department of Systems and Computer Engineering, Carleton University, Ottawa K1S 5B6, Canada
✉ *E-mail: chlung@sce.carleton.ca*

Abstract: Software defined networking (SDN) is an emerging networking architectural framework which aims to provide the complete separation of data forwarding plane and control plane. The two main benefits of SDN are lower cost and improved management. OpenFlow is a well-known architecture that facilitates SDN. The core idea of OpenFlow is to control the switches or routers through programming from the centralised controller. The connection reliability is a major concern for network service providers to meet quality of service requirements. Hence, an investigation of network resiliency is required for this new paradigm. The resiliency is the network's ability to survive against attacks and other component failures. This study investigates and compares different data forwarding algorithms currently supported by the POX OpenFlow controller standards for network protection and restoration. A thorough investigation of existing approaches or standards in SDN not only is essential for the research community to better understand the topic, but also plays a crucial role in realising or improving network resiliency or protection and restoration in practice. The authors also provide the extension of one of the components in POX for improvement. The restoration scheme in the current POX components as well as in the modified component is evaluated and compared.

1 Introduction

Separating the control plane and data forwarding plane allows network administrators to manage networks easily. This approach is referred to as software defined networking (SDN) [1]. In the traditional networks, both of these planes are integrated within the same network device In SDN, the controller manages the entire network state from a central vantage point or controller, which hosts features for the entire network, such as routing protocols, access control, network virtualisation, energy management and new prototype features [1]. The controller is logically centralised. However, the controller could possibly be distributed physically [2]. In SDN, the southbound application program interface (API) makes the controller able to connect with switches in the network. The well-known southbound API is the OpenFlow protocol, which was released by ONF (open network foundation). This protocol enables the administrator to control the routing table remotely.

The controller is the strategic control point in SDN which controls the switches or routers by relaying the information using the OpenFlow protocol. The controller manages the flow control to enable intelligent networking. Since the controller is centralised, SDN provides better management and security. Some critical features of SDN are summarised as follows [1]:

- It simplifies the hardware of network devices (switches or routers) and reduces the hardware and operational cost.
- It facilitates better traffic engineering and quality of service (QoS) for improved management.
- It is programmable and can be used for continuous network monitoring and quick network-wide or individual devices adjustments.

1.1 Resiliency in software defined networks

Several approaches have been proposed for protection and restoration in SDN. In [3], the proposed architecture relies on preplanned backup paths to ensure that, in case of a failure, recovery is performed by affected switch locally. This scheme can minimise the restoration time. To support the proposed scheme, several enhancements to the OpenFlow architecture have been proposed in this paper. However, switches cannot make intelligent decisions in SDN, as their primary operation is limited to forwarding of packets only. In case of a failure, if a switch directly forwards the traffic via the backup path, then the controller is bypassed and some other paths may be affected without going through the controller. The nature of network traffic is dynamic, hence while calculating the backup path, we must consider the current traffic situation and, in the context of SDN, control decision should be made in the controller and switches should be limited to data forwarding only.

When a failure occurs, the first action required is detecting the failure. The mechanism can be used here is bi-directional forwarding (BFD) mechanism [4] session to notify the failure. BFD is a network protocol used to detect the failure between the source and the destination. It facilitates low-overhead detection of failure on physical media, such as Ethernet, virtual circuit and tunnels. In SDN, data and control planes are working separately.

Data plane restoration can be performed in two possible ways [5]. One is to support the recovery mechanism provided by a protocol by implementing it with the OpenFlow. Another approach is to implement the protection or restoration mechanism in OpenFlow itself. Data plane restoration can be achieved by immediately calculating the new flow values for affected switches. Protection scheme can be implemented by pre-emptively calculating the back-up path for all the pairs and storing them in the controller.

The control plane is responsible for entire network's control and management functionalities. The control plane can be implemented in two ways. The first is in-band control in which data and control planes share the same channel for transmissions. The network resiliency scheme for in-band control is explained in [6]. The second is out-of-band control in which data and control planes are using separate communication channels. In this case, if the control plane channel fails, we can use data plane channel for restoration.

The controller failure can cause the complete service disruption in the network. For this reason, controller placement and physical

distribution of multiple controllers is crucial [2]. For controller failure, several approaches for distributed controllers or a primary and a backup model have been proposed and evaluated. The latest OpenFlow protocol [7] supports the distributed controllers approach. However, higher workload will occur [8, 9] to replicate data and to ensure consistency between controllers.

1.2 Research objective

The main objective of this paper is to explore the area of protection and restoration in SDN, as it is a crucial functionality in practice for traffic engineering and QoS, and the topic is still at the research stage. In this work, we have investigated existing forwarding algorithms in the POX controller. Various restoration approaches based on OpenFlow standards using the POX controller have been compared and evaluated. We have conducted a detailed analysis of the design and implementation of those approaches. Understanding existing forwarding approaches in POX plays a crucial role as fast restoration is essential for carrier grade services. The evaluation can help researchers better understand existing approaches in POX. In addition, we propose some modifications to one of the algorithms to improve recovery performance. This paper is an extension of our preliminary comparison of different forwarding algorithms for SDN network resilience [10]. The extension includes more detailed descriptions and discussions for various algorithms, and experiments and results with an extra performance metric and a larger network topology. In addition, this paper also presents a comparison between SDN used for wired networks and potentially for wireless sensor networks (WSNs) with an emphasis on the resilience perspective.

The rest of the paper is organised as follows: Section 2 presents existing POX forwarding algorithms. Section 3 describes the evaluation of various forwarding algorithms from the protection and restoration perspective. Section 5 depicts the conclusion and some future directions.

2 POX forwarding approaches

This section presents an overview of the main components of the forwarding functionality used in the current POX [11]. The reasons for choosing POX include POX was the first open source SDN emulator and was robust and commonly used for experiments than, and it also has an easy-to-use Python interface, which is more effective than NOX OpenFlow controller [11] at the time we conducted our experiments. Moreover, POX has the ability to run anywhere by bundling with the PyPy runtime environment. A survey of existing forwarding functionality in SDN is crucial in understanding the current standards and devising an efficient protection and restoration mechanism. Sections 2.1–2.5 explain the forwarding algorithms present in the current POX version. Section 2.6 describes our proposed modifications to the *L2_Multi* component to support the fast restoration. Sections 2.7 and 2.8 discuss two major components of OpenFlow in POX that are crucial to failure identification *Discovery* and *Spanning tree*, respectively.

2.1 L2_Learning

The *L2_Learning* component in POX acts as a layer 2 switch. It learns the different sources based on their media access control (MAC) addresses and maps them to their corresponding incoming port. In POX any component is invoked by the *Launch* function. The learning switch takes the appropriate action based on the incoming packet. It checks the parameters and destination address and forwards the packet accordingly.

Here as we can see from Fig. 1, the *Learning_switch* function will parse the packet and take the appropriate action based on the packet type. This is very straight forward switch which will eventually learn all the destinations.

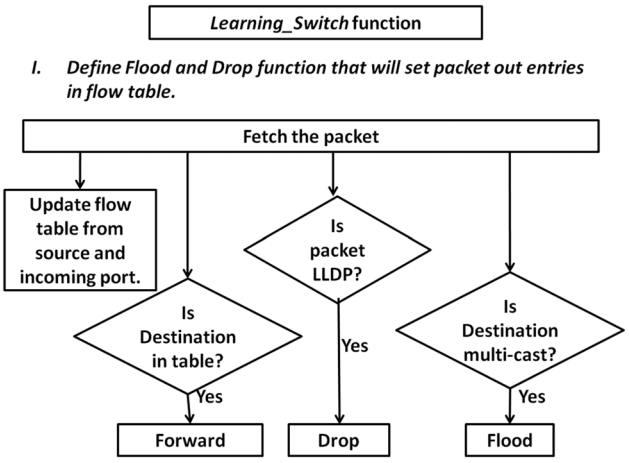

Fig. 1 *Functional view of L2_Learning*

2.2 L3_Learning

L3_Learning acts the same as *L2_Learning* for forwarding, except *L3_learning* component in POX acts as a layer 3 device. *L3_Learning* has some functionality to reply address resolution protocol (ARP) requests [11]. *L3_Learning* keeps a table that maps IP to MAC and corresponding ports. When a switch receives an ARP query, it will check the table for this entry. If the entry is found then it will answer the ARP query; otherwise, it will flood it.

2.3 L2_Pairs

L2_Pairs algorithm performs almost the same functionality as *L2_Learning* [11]. This is just another method to create the learning switch. *L2_Pairs* function is associated with *Handle_Connection_Up* event and it fetches the packet associated with the event (incoming packet). If the packet is in a flow table then it will just forward it through listed port; otherwise, it will flood the packet to all the ports except the incoming port.

As described in Fig. 2, whenever a packet comes into the switch it will search the flow table for the destination. If there is a matching entry for the destination then it will fetch the outgoing port and forward the packet. At the same time it will also install the rule for such packet. If there is no matching entry then it will simply flood the packet to all other ports except the incoming port.

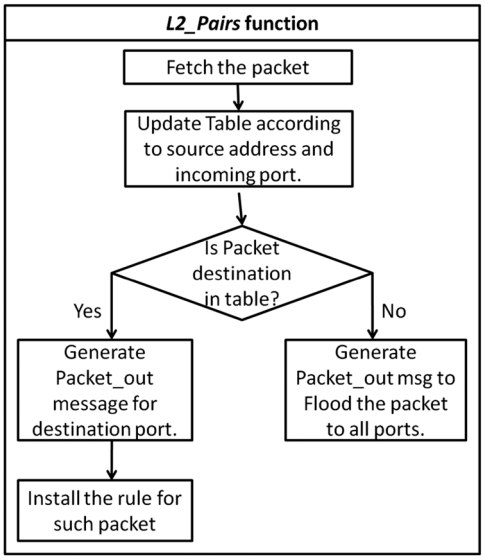

Fig. 2 *Functional view of L2_Pairs*

2.4 L2_Multi

This is used for the layer 2 switch that learns Ethernet addresses across the entire network and selects shortest path(s) between them [11]. It imports the *Discovery* module in POX and the adjacency list identified in the *Discovery* module. *L2_Multi* defines the class for switch as well as the path map between two switches with intermediate nodes and distance.

Calculate_path function is responsible for calculating the shortest path for a given set of nodes. Here this component uses the Floyd–Warshall algorithm to calculate the shortest paths between all pairs of nodes. When this function is called, it will first clear all the paths in *path_map*. It then calculates the shortest path for all the switches in the adjacency list. To calculate the shortest path, this function will identify the nodes that are directly connected. As a second step it will try to find the nodes that are one hop away from the origin. This process continues until all the nodes in the network are identified by the shortest distance. It finds the intermediate nodes such that the distance between all the source–destination pairs is minimised.

The limitation of this approach is that whenever a link event occurs, it wipes out the entire stored paths (*path_map*) and starts the process of calculating paths again. Here if we add a mechanism that only updates the path map with the affected path then there will be less computation overhead compared to the entire path map calculation adopted by the current *L2_Multi*.

2.5 L2_Flowvisor

L2_Flowvisor installs the flow entry the same way as *L2_Pairs* does [11]. This component uses the *calc_spanning_tree* function of the *spanning_tree* component in order to find and update the spanning tree. It does not set the *no_flood*; instead, it simply conducts flooding for the selected ports from the spanning tree. This component imports the *Discovery* component in POX and catches the link status events fired by the *Discovery* module. The core functionality is depicted in Fig. 3.

Every time a packet arrives, a switch checks the packet type and takes the appropriate action. If the packet is multicast, then it will forward the packet to the specific group with the ports listed in the *spanning_tree*. If the packet is unicast and there is no match in the flow table, then the switch will flood this packet to all other ports.

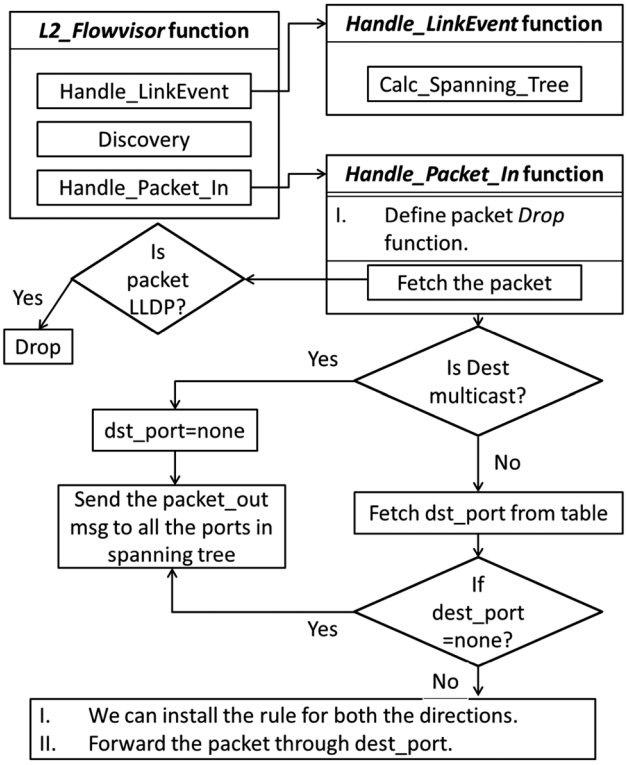

Fig. 3 *Functional view of L2_Flowvisor*

2.6 L2_LR (layer 2 link restoration)

This section describes proposed modifications to the *L2_Multi* switch component to support local link restoration. As we can see from Fig. 4, there is a solid line between S1 and S2 which represents the primary path link. In case of a link failure, the added mechanism for *L2_LR* finds the backup path between these switches. If there is a secondary link available between these two switches, then *L2_LR* will select this link as the restoration link. As depicted in Fig. 3, there is a direct secondary link which is shown by the top dotted line.

If there is no secondary link between these two end switches, then *L2_LR* will find a common switch(es) in the adjacency list of both end switches. Here both S1 and S2 have common directly connected switches S3 and S4. In this case, either S3 or S4 is selected and then the traffic is routed through the selected node. If there is no common switch, then *L2_LR* will find a pair of switches such that each end switch (S1 or S2) is connected to one switch from this identified pair and the switches present in the pair are directly connected to each other. As shown in Fig. 3, S5 and S6 are the two switches that satisfy the above condition, e.g. S5 is directed to S1 and S6 is directly connected to S6, and S5 and S6 are directly connected.

This is a temporary restoration mechanism that can reduce packet losses until the new discovery cycle is initiated and the shortest path is established. The restoration time will be similar to *L2_Multi* as this method highly depends on the discovery cycle for network recovery. The benefit is the quick response to find a temporary backup link or path to reduce packet losses. Section 3 presents an experiment and results for the forwarding schemes. A possible extension of this solution is to consider other parameters while selecting the local node as a backup path. For instance, we can consider current traffic patterns as well as the cost associated with each node.

2.7 Discovery component in POX

The *Discovery* component is a key module that is used to identify the connectivity between OpenFlow switches by sending periodically link layer discovery protocol (LLDP) packets [11]. The *Discovery* component sends out LLDP packets, and monitors the arrival of the returned LLDP packets from other switches. It maintains the adjacency list which has the information about nodes and their connections with their neighbour nodes. The component also triggers the link events when the link state is changed. However, as per the current version of POX, the *Discovery* component takes too much time (around 4–5 s) to update the failed link status, which causes the high recovery time and does not meet the carrier-grade requirements. There is a need of a mechanism that can immediately notify the failure and recover from it.

An essential component that is responsible for sending and receiving LLDP packets from the controller is called *LLDPSender*. The functionality is explained in Fig. 5. The following describes the main functionalities of *LLDPSender*:

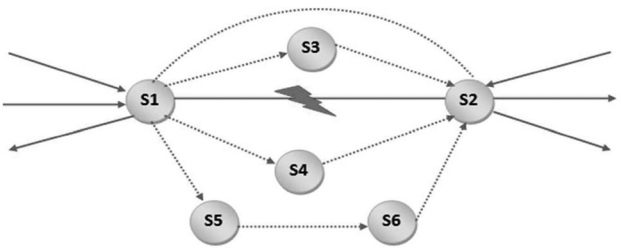

Fig. 4 *Example for L2_LR*

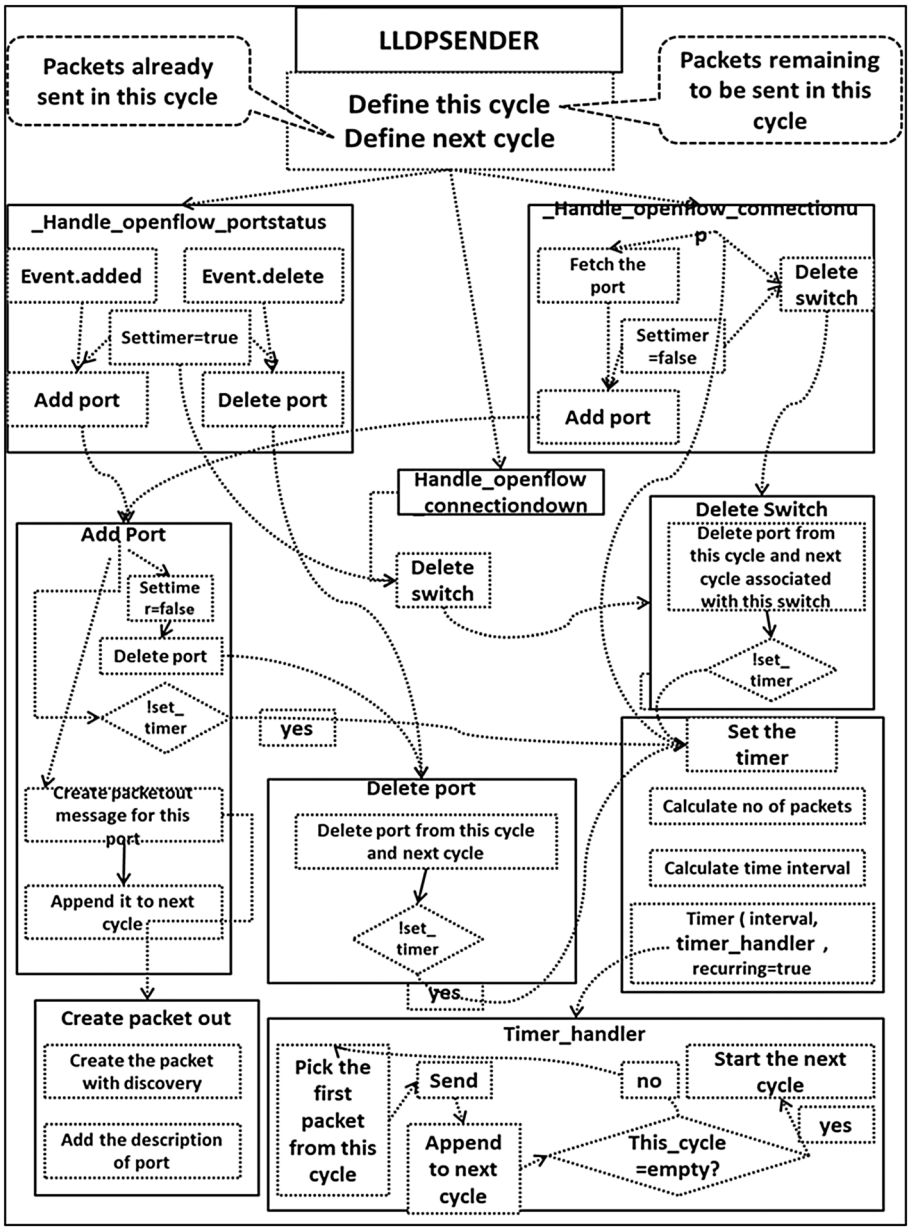

Fig. 5 *Detailed functional view of LLDPSender*

• The process of *LLDPSender* class is explained in Fig. 2. *LLDPSender* is responsible for periodically sending LLDP packets to all the switches it is connected to. *LLDPSender* defines this cycle list and the next cycle list that keeps track of packets to be sent in this cycle and already have been sent in this cycle, respectively.

• *LLDPSender* listens to the port event. If a new port is added then it will generate a packet out message for this port and append it to the next cycle list. If the port is removed, then it will remove the port from both the cycle lists.

• *LLDPSender* also listens to the connection up and connection down events when a switch connects or disconnects. When a switch is newly connected, the ports in this switch are added to this cycle list and the switch is deleted for a particular timestamp. Timer handler is responsible for starting a new cycle list after this cycle list is empty.

2.8 Spanning tree component in POX

The *Discovery* component is also used by the *Spanning Tree* [11] component to build the network topology. *L2_Flowvisor* (see Section 2.5) uses a *calc_spanning_tree* function to build a spanning tree. This component creates the spanning tree for all the nodes and then disables flooding on unused ports.

The *calc_spanning_tree* function calculates the tree for each node present in the network. The algorithm to calculate the tree is presented in the following steps:

(i) Get adjacency list from *Discovery*.
(ii) Define a set of links and switches that are present in the network.
(iii) Put switches in list and sort them by datapath identifier (dpid).
(iv) Start with the first switch in list as a root.
(v) Find all the adjacent switches and add them to the tree.
(vi) Find the switches that are connected to the root via one intermediate switch and add them to the tree with intermediate node.
(vii) Repeat the procedure until all the switches are in the tree.
(viii) Choose the next switch in list as a root and go to step v.
(ix) Finally, the entire tree set contains the spanning tree for all the switches.

In summary, Table 1 presents a comparison of the main features of the abovementioned forwarding algorithms that are available in POX.

Table 1 Summary of main features for forwarding algorithms

Feature	*L2_Learning*	*L3_Learning*	*L2_Pairs*	*L2_Flowvisor*	*L2_Multi*	*L2_LR*
failure detection	√	√	√	√	√	√
failure recovery	×	×	×	×	√	√
initial configuration computation	×	×	×	√	√	√
shortest path calculation	×	×	×	×	√	√

We have considered all the existing algorithms except *L2_Multi* which is launched with the *Discovery* and *Spanning Tree* components in POX. Failure detection is facilitated by the *Discovery* component as it triggers the link events.

However, as far as the recovery is concerned, the first four algorithms in Table 1, e.g. *L2_Learning*, *L3_Learning*, *L2_Pairs* and *L2_Flowvisor*, do not react against the failure. They simply repeat its procedure and eventually establish the new flow, but no immediate action is performed. On the other hand, both *L2_Multi* and *L2_LR* immediately react to the failure. Whenever they are notified with the link event, they initiate the path calculation process for changed link status and establish the new route from source to destination. Complex configuration computation is initially performed using approaches *L2_Flowvisor*, *L2_Multi* and *L2_LR*. *L2_Flowvisor* simulates the flooding according to the spanning tree so tree calculation is required initially. *L2_Multi* and *L2_LR* pre-calculate the path for all the sources and destinations. This is the reason why they require initial computation. The shortest path calculation is performed by *L2_Multi* and *L2_LR* only.

3 Evaluation of POX forwarding approaches

This section presents performance evaluation and comparison of different forwarding algorithms depicted in Section 2. We have adopted several evaluation criteria for the comparison. First, we investigated the paths selected by different forwarding algorithms, which helps understand different algorithms and the overhead and cost associated with different paths. The next two criteria are related to CPU usage for various algorithms, which is also related to scalability if the network size increases, as the controller has to perform many other tasks. Higher CPU demands may affect response time, throughput, or other performance metrics. The last criterion is packet losses due to a link failure. Packet losses have direct impact on QoS which becomes vital for today's network services and applications.

We have conducted experiments using Mininet 2.1.0 on POX. The network topology used for experiments is depicted in Fig. 6. The topology is an extension of network size compared to our previous study in [10]. The computing machine used for this experiment is *Intel core i7* processor with 3.2 GHz clock speed. For virtualisation purpose, *Virtual* Box 4.2.18 is used. There are 26 switches (S1–S26) and a controller residing on S1 in the hybrid mesh network. There is a host connected to each switch (e.g. hosts H1–H26 are connected to switches S1–S26, respectively). The hosts are not shown in Fig. 6 for brevity. All the 43 links have 10 Mbps of bandwidth. For larger network sizes, the results will be similar, because all the switches have a direct control plane connection with the controller based on the SDN principle. When a failure is detected, any switch can communicate directly with the controller, which is independent of the network size.

All the forwarding algorithms presented in Section 2 have been evaluated with the *Discovery* and *Spanning_tree* components, except the *L2_Flowvisor*. The *L2_Flowvisor* is not compatible with the *Spanning Tree* component, hence it is directly launched with the *Discovery* module.

3.1 Path selection

In the experiments, hosts H1 and H26 are considered as the source and destination and they are connected to S1 and S26, respectively. The paths identified by different forwarding algorithms are illustrated in Fig. 6 and listed in Table 2 when the *ping* messages were sent from H1 to H26. As we can see in Table 2, *L2_Multi* and *L2_LR* select the path with the least number of intermediate nodes, because both of them are based on the shortest path algorithm. The first three algorithms in the table randomly select the path from H1 to H26. Specifically, this path selection is based on the first entry made in the flow table as well as the port status changed by the *Spanning Tree* component. *L2_Flowvisor*, on the other hand, simply forwards the packet to the ports, which allows it to find an optimal path. However, this is not always the case, because it highly depends on the tree calculation procedure.

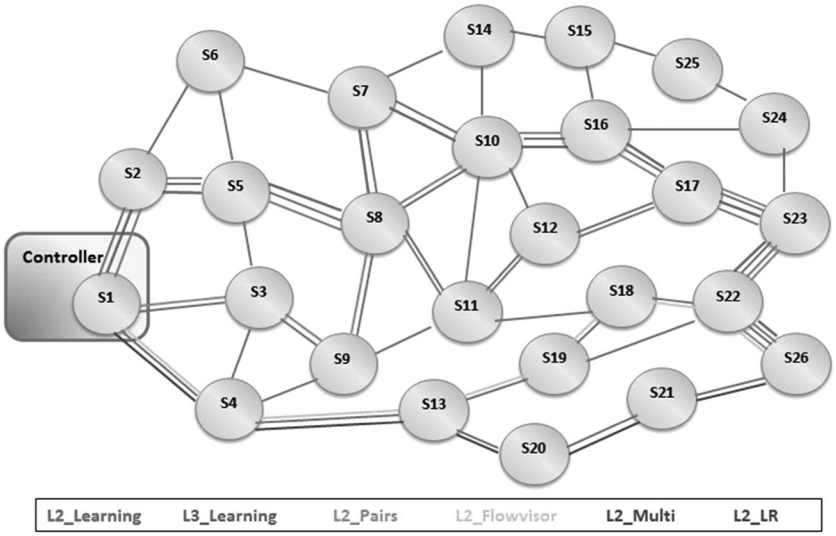

Fig. 6 *Network topology*

Table 2 Path selected by different forwarding algorithms

L2_Learning	S1→S2→S5→S8→S7→S10→S16→S17→S23→S22→S26
L3_Learning	S1→S2→S5→S8→S11→S12→S17→S23→S22→S26
L2_Pairs	S1→S3→S9→S8→S10→S16→S17→S23→S22→S26
L2_Flowvisor	S1→S4→S13→S19→S18→S22→S26
L2_Multi	S1→S4→S13→S20→S21→S26
L2_LR	S1→S4→S13→S20→S21→S26

3.2 CPU usage of forwarding algorithms

The aggregated CPU usage is measured using Linux *top* command for the entire forwarding algorithm to find the computational overhead of different algorithms. Computational overhead is an important factor to consider for scalability, as the SDN controller may need to perform various functionalities for many switches. As we can see in Fig. 7, *L2_Flowviso*, *L2_Multi* and *L2_LR* have slightly higher overhead than that of the other three algorithms. This deviation is caused by route calculation in these algorithms. When each of these algorithms is initially launched, it has higher overhead due to more initial operations. After the process of flow establishment is finished, they have neutral overhead of around 3%.

3.3 CPU usage for flow setup

When a switch receives a packet from any source for the first time it will install the flow entry in the flow table. When a switch learns all the nodes from the network, the complete flow establishment has taken place. The *pingall* command in Mininet is used to check the connectivity of the entire network. The *pingall* command is sent from every host to all other hosts. The CPU usage is measured for this initial flow establishment. Fig. 8 shows that *L2_Learning*, *L2_Multi* and *L3_Learning* have higher CPU usage than the other three algorithms.

The main reason is that *L2_Multi*, *L2_Flowvisor* and *L2_LR* calculate the paths once the links and nodes are identified. When the *pingall* command is fired these three algorithms require less number of packets to be flooded into the network and that is the reason for low CPU usage.

3.4 Packet loss in presence of failure

To evaluate the effect of the failure on different forwarding algorithms, packet losses per 100 packets have been measured. In spite of the failure, the round trip time results for different algorithms are almost the same (0.027 ms), thus we present only the packet losses. After running the traffic with a specified forwarding algorithm, we failed one of the links along the primary path to generate the link failure and packet loss scenario. The failed links for the algorithms were different according to the path chosen by different algorithms, as the path generated for each algorithm may be different, see Fig. 6 or Table 2 for the path for each algorithm. The failed link was chosen arbitrarily and the selection of the failed link does not make any difference, as each link is directly connected to the controller and the same mechanism is used for each switch and the controller for each forwarding scheme.

To measure the packet loss, 100 ping messages were sent. The results are shown in Fig. 9. In case of first three algorithms (*L2_learning*, *L3_learning* and *L2_Paris*), if we fail the link from the primary path while traffic is running, then each of them will start flooding again and try to find the appropriate port that leads the packet to the destination. Another reason for high packet losses for the first three algorithms is that they do not store the entire path from the source to the destination, whereas the last two algorithms (*L2_Multi* and *L2_LR*) store the entire path as well as whenever the link failure is detected they invalidate all the flows and try to find the optimum path using the *Discovery* module. There is only a minor difference between the *L2_Multi* and *L2_LR* algorithms based on the experiment, because they both depend upon the discovery cycle adopted in POX. However, as the traffic rate increases, the difference of packet losses is expected to become larger between *L2_Multi* and *L2_LR*, because *L2_LR* transmits traffic over a tentative link/path before the Discovery module recalculated the shortest path.

4 Conclusions and future work

SDN is an emerging research topic and it has drawn a great deal of attention. On the other hand, network resiliency is a critical

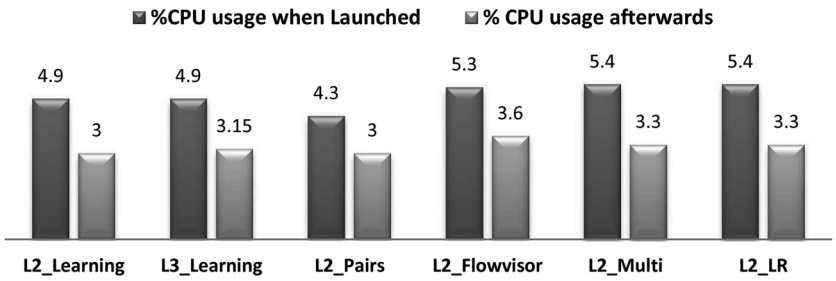

Fig. 7 *CPU usage of different forwarding algorithms*

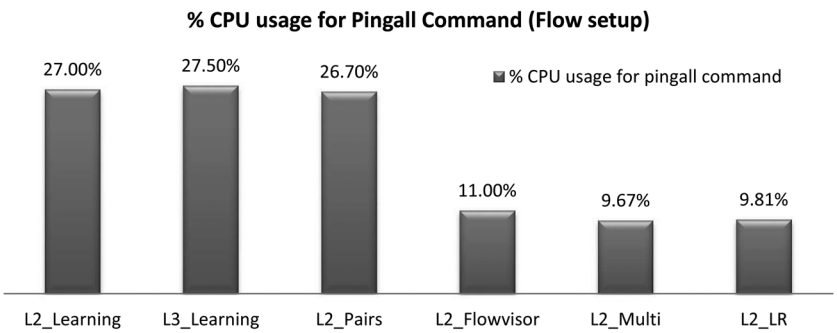

Fig. 8 *CPU usage for pingall operations (flow setup)*

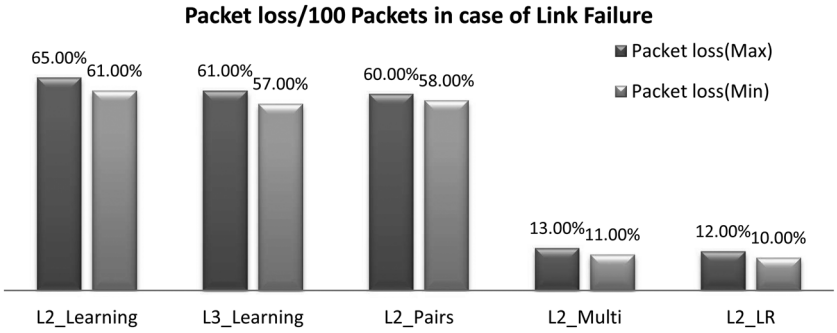

Fig. 9 *Packet loss in case of link failure*

requirement for network services these days. It is crucial to understand how existing SDN approaches measure up against the reliability and QoS requirements [2]. The paper presented a detailed comparison of existing forwarding approaches on POX for SDN resiliency. The comparison helps better understanding of the crucial topic and for future improvement.

We conducted experiments using available forwarding schemes on POX and also proposed *L2_LR* for improvement. Experimental results showed that *L2_Multi* and *L2_LR* require less time for recovery from the failure in comparison to other forwarding schemes. *L2_LR* also generates less packet losses than that of *L2_Multi*, as using *L2_LR* traffic can be forwarded via a backup link/path identified temporarily before the Discovery module starts. The experimental results are consistent with that obtained from a smaller network [10]. For larger network sizes, the results will be similar, because all the switches have a direct connection with the controller based on the SDN principle. When a failure is detected, the corresponding switches can communicate directly with the controller, which is independent of the network size.

Our experience with this investigation has pointed to two major research areas of future research:

4.1 Improving network resilience for reliability and QoS requirements

The restoration time for the existing POX is high due to the long discovery cycle. Reducing the discovery cycle to a very small value would generate a high volume of traffic and may not be able to meet the 50 ms recovery time of network reliability requirement. If an SDN is deployed in a geographically large network, the delay will become even higher due to longer propagation delay for messages transmitted between switches and the controller. The high restoration time needs to be reduced to meet the network reliability requirements and ever increasingly essential services and applications.

In other words, there is a need of a protection scheme for fast recovery which does not require controller-switch communications when a failure occurs. Different potential modifications to the POX design can be considered to increase network resilience efficiency. Protection mechanisms that pre-establish backup paths are more efficient and have been used in today's high-speed networks, e.g. Multiprotocol Label Switching (MPLS) fast reroute [12]. For SDN, if a backup entry has been pre-established and stored in a flow table when a failure is detected, the backup entry can be used immediately. Protection can be divided into path protection and link protection. For path protection, the following steps warrant further research for validation:

(i) After the discovery procedure of links and switches is completed, the controller can find primary as well as backup paths for all possible source and destination pairs. If we consider the *L2_Multi* algorithm, then there will be two *path_map* lists, one for the primary paths and the second for backup paths. These calculations should be updated periodically because of dynamic traffic conditions and to avoid the congestion.

(ii) Installing two paths for the same destination which can be achieved by two ways. First, flow table entries can be stored with different priority levels. The second option is to reserve a buffer in switches, which allows switches to store and retrieve additional control information. In this buffer, we can store the backup path.

(iii) If a component fails, then the nodes that are using this component for their primary path are notified to alter their path entries to the backup path. The nodes that are using this component for their backup path are also notified to calculate their backup path again. The controller will re-calculate and pre-establish the new backup path and send it to node to update its buffer.

The possible link protection solution based on OpenFlow could be

(i) The controller will calculate the backup path set for all the links in the network.

(ii) Flow tables in the switches are installed with primary and secondary forwarding ports. The assumption is that the secondary forwarding port is pre-calculated such that it will lead to the destination from the point of failure.

4.2 Network resiliency for software-defined WSNs

Although SDN was not originally proposed for WSNs, many WSNs share a similarity with the concept of SDN – a centralised base station or sink for WSNs and a centralised controller for SDN. In WSNs, though, the sink(s) may only be used for data collection and aggregation instead of for managing or controlling of the traffic of sensor nodes.

With the advancement of sensor and microcontroller technologies, WSNs have drawn a great deal of attention and there are more applications using WSNs. Typically, WSNs are specific to applications and environments. However, there are some general problems for WSNs, as described in [13]. Some of the problems are (i) counter-productivity: different vendors develop WSNs in isolation without considering common features; (ii) inflexible to policy changes: changes in business need are difficult to handle algorithmically; and (iii) difficult to manage: development of a network management system for distributed WSNs is non-trivial and difficult to deal with as technologies advance quickly.

A tremendous amount of efforts has been spent on various performance-related issues for WSNs, including clustering of sensor nodes for data aggregation and network lifetime prolongment [14, 15], routing calculation for energy harvesting [16], sensor node localisation [17] and so on. Although the performance issues, e.g. network lifetime, are crucial to WSNs, other issues, such as the aforementioned problems described in [13] may outweigh some of the performance issues for specific applications in the long run. In addition, the cost issue may become prevailing for some applications, as many research efforts on WSNs are based on fully fledged sensor nodes with functionalities of all layers, e.g. from physical layer to application layer, which increases both the capital expenditures and operational expenditures.

Luo *et al.* [13] proposed an approach to enabling software-defined WSNs to address the aforementioned problems existing in existing WSNs. With SDN technologies, sensor nodes could focus on simplified data sensing and forwarding tasks without carrying extra overhead of different functionalities. The management of sensor nodes also becomes simpler, since updates mostly can be performed at the controller(s) or sink(s) as opposed to all the sensor nodes in a WSN.

From the network resilience perspective, node or link failures typically can happen more frequently in WSNs than in wired networks. With POX, if a failure is detected due to a loss of connectivity, a switch will react by notifying the controller. The controller will then take an action: either using a pre-established path or re-calculating a new path.

For WSNs, pre-established protection schemes may generate a lot of control traffic, which consumes a large amount of energy. Further, pre-established paths may not be reliable, as more frequent node or link failures in a WSN may affect the protected paths. In other words, the protected paths may not guarantee connectivity, especially for sensor nodes that are deployed in a harsh environment. As a result, a reactive approach to restoration may still be needed or appropriate for WSNs after a failure is detected, unless the sensor nodes and connections are reliable. The existing POX OpenFlow approach to restoration is based on the reactive method, which could be investigated further for adaptation and improvement for WSNs.

There is another evident difference between the wired SDN and software-defined WSNs: the control traffic for WSNs will typically be realised with an in-band channel, whereas the Mininet or SDN uses a separate control channel for control plane by default. Most sensor nodes are equipped with only one communication channel. If multiple channels are available or a dedicated control channel is used for a WSN, the cost will be high for many applications and the lifetime will also be affected. For some applications, some sensor nodes may be far away from the sink (a potential controller). In this case, multi-hop communications may be needed [11] using an in-band channel for restoration, which could cause high delay and energy consumption.

Protection and restoration for WSNs is still an open area. WSNs share some similarity with SDN. More research needs to be conducted for this area. In addition, a tradeoff evaluation should be considered in combination with other factors, such as performance, energy consumption, management and cost.

5 References

1 Bob, L., Heller, B., McKeown, N.: 'A network in a laptop: rapid prototyping for software-defined networks'. Proc. Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, 2010

2 Ros, F.J., Ruiz, P.M.: 'Five nines of southbound reliability in software-defined networks'. Proc. Third Workshop on Hot Topics in Software Defined Networking (HotSDN), August 2014, pp. 31–36

3 Sgambelluri, A., Giorgetti, A., Cugini, F., *et al.*: 'OpenFlow-based segment protection in Ethernet networks', *IEEE/OSA J. Opt. Commun. Netw.*, 2013, **5**, (9), pp. 1066–1075

4 Katz, D., Ward, D.: 'Bidirectional forwarding detection', IETF RFC 5881, 2010

5 Staessens, D., Sharma, S., Colle, D., *et al.*: 'Software defined networking: meeting carrier grade requirements'. Proc. 18th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN), 2011

6 Sachin, S., Staessens, D., Colle, D., *et al.*: 'Fast failure recovery for in-band OpenFlow networks'. Proc. of Ninth IEEE Int. Conf. on Design of Reliable Communication Networks (DRCN), 2013

7 Openflow Switch Specification: Version 1.3.3. Open Networking Foundation. June 2012

8 Koponen, T., Casado, M., Gude, N., *et al.*: 'Onix: a distributed control platform for large-scale production networks'. In OSDI, 2010, vol. 10, pp. 1–6

9 Cai, Z., Cox, A.L., Ng, T.S.E.: 'Maestro: a system for scalable OpenFlow control'. Technical Report TR10-08, Rice University, 2010

10 Vaghani, R., Lung, C.-H.: 'A comparison of data forwarding schemes for network resiliency in software defined networking'. Proc. of Int. Workshop on Software Defined Networks for a New Generation of Applications and Services (SDN-NGAS), August 2014

11 NOX and POX, http://www.noxrepo.org/

12 Pan, P., Swallow, G., Atlas, A.: 'Fast reroute extensions to RSVP-TE for LSP tunnels'. Internet Engineering Task Force (IETF) Request for Comments (RFC) 4090, May 2005

13 Luo, T., Tan, H.-P., Quek, T.Q.S.: 'Sensor OpenFlow: enabling software-defined wireless sensor networks', *IEEE Commun. Lett.*, 2012, **16**, (11), pp. 1896–1899

14 Hu, S., Han, J.: 'Power control strategy for clustering wireless sensor networks based on multi-packet reception', *IET Wirel. Sens. Syst.*, 2014, **4**, (3), pp. 122–129

15 Kumar, D.: 'Performance analysis of energy efficient clustering protocols for maximising lifetime of wireless sensor networks', *IET Wirel. Sens. Syst.*, 2014, **4**, (1), pp. 9–16

16 Wu, Y., Liu, W.: 'Routing protocol based on generic algorithm for energy harvesting-wireless sensor networks', *IET Wirel. Sens. Syst.*, 2013, **3**, (2), pp. 112–118

17 Wang, G., Yang, K.: 'A new approach to sensor node localization using RSS measurements in wireless sensor networks', *IEEE Trans. Wirel. Commun.*, 2011, **10**, (5), pp. 1389–1395