Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

A subtree-based approach to failure detection and protection for multicast in SDN#

Vignesh RENGANATHAN RAJA[1], Chung-Horng LUNG‡[1], Abhishek PANDEY[1], Guo-ming WEI[2], Anand SRINIVASAN[3]

([1]*Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario K1S 5B6, Canada*)

([2]*School of Computer Science, Carleton University, Ottawa, Ontario K1S 5B6, Canada*)

([3]*Eion Inc., Ottawa, Ontario K2K 2E3, Canada*)

E-mail: vigneshrenganathanra@cmail.carleton.ca; chlung@sce.carleton.ca; abhishekpandey@cmail.carleton.ca; guomingwei@gmail.com; anand@eion.com

Received Apr. 4, 2016; Revision accepted Apr. 24, 2016; Crosschecked June 19, 2016

Abstract: Software-defined networking (SDN) has received tremendous attention from both industry and academia. The centralized control plane in SDN has a global view of the network and can be used to provide more effective solutions for complex problems, such as traffic engineering. This study is motivated by recent advancement in SDN and increasing popularity of multicasting applications. We propose a technique to increase the resiliency of multicasting in SDN based on the subtree protection mechanism. Multicasting is a group communication technology, which uses the network infrastructure efficiently by sending the data only once from one or multiple sources to a group of receivers that share a common path. Multicasting applications, e.g., live video streaming and video conferencing, become popular, but they are delay-sensitive applications. Failures in an ongoing multicast session can cause packet losses and delay, which can significantly affect quality of service (QoS). In this study, we adapt a subtree-based technique to protect a multicast tree constructed for OpenFlow switches in SDN. The proposed algorithm can detect link or node failures from a multicast tree and then determines which part of the multicast tree requires changes in the flow table to recover from the failure. With a centralized controller in SDN, the backup paths can be created much more effectively in comparison to the signaling approach used in traditional multiprotocol label switching (MPLS) networks for backup paths, which makes the subtree-based protection mechanism feasible. We also implement a prototype of the algorithm in the POX controller and measure its performance by emulating failures in different tree topologies in Mininet.

Key words: Software-defined networks (SDNs), OpenFlow, Multicast tree, Protection, POX controller, Mininet, Multiprotocol label switching (MPLS)

http://dx.doi.org/10.1631/FITEE.1601135 **CLC number:** TP393

1 Introduction

Network failures are inevitable due to various reasons, such as administrative errors, system bugs, and fiber cuts. When a failure occurs (e.g., a link or node failure), the network layer of the Internet Protocol (IP) stack at each node will recalculate the routing paths to every other node. The failure recovery time can be high, e.g., tens of seconds or even minutes (Pan *et al*., 2005), using the traditional IP recovery scheme at the network layer. The delay can cause tremendous destruction to end users and businesses for today's high-speed networks, which is unacceptable for highly popular video or multimedia applications and business demands. Multiprotocol label switching (MPLS) has become widely adopted by Internet service providers (ISPs) for their backbone networks (Osborne and Simha, 2002). One of the benefits of MPLS networks is the feature of fast reroute (FRR) (Pan *et al*., 2005), which facilitates

‡ Corresponding author

A preliminary version was presented at the 13th Wireless and Wired International Conference, Malaga, Spain, May 25–27, 2015

ORCID: Chung-Horng LUNG, http://orcid.org/0000-0002-5662-490X

rapid failure recovery. FRR has been practically used by ISPs to protect label switched paths (LSPs) from link or node failures. FRR uses a local repair mechanism for the protected LSPs at the configured point of failure. FRR relies on a pre-established backup LSP for either link or node protection, which can significantly shorten the failure recovery time. FRR can even support bandwidth protection for the protected LSPs using resource reservation protocol-traffic engineering (RSVP-TE) (Osborne and Simha, 2002; Pan *et al*., 2005).

Multiple backup LSPs may be required to protect a primary LSP if bandwidth protection is needed for quality-of-service (QoS) requirements. Pointurier (2002) presented an approach to reducing the number of backup paths and the bandwidth resources required to meet the same level of QoS constraints. The approach divides a long primary LSP into segments with an aim to protect each segment and also to meet the QoS constraints at the same time. Dividing a primary LSP into segments is flexible, depending on the network topology and QoS requirements. A tighter requirement implies that more segments and more backup LSPs are needed. In addition, the reservation for more backup bandwidth is essential for more segments.

Multicasting has become more popular due to the increasing popularity of video and multimedia applications. Multicasting is a group communication technology, which makes the network infrastructure usage more efficient by transferring the same data only once from a sender to a selected group of receivers that share a common path. Multicasting is bandwidth efficient due to shared paths. Some applications for multicast include video conferencing, video streaming, corporate communications, etc.

Multicasting usually requires the pre-establishment of a tree structure, a multicast tree, based on the interested members and the network topology. Multicasting enables the sender to deliver the same traffic flow to a group of receivers that share a common path without duplicating the packets. Packets are duplicated only when necessary at certain nodes, e.g., branch nodes in a multicast tree. The mechanism can greatly reduce redundant traffic flowing in the network and bandwidth usages, especially when the multicast tree is large.

The network devices, i.e., routers, involved in a multicast session are capable of forming multicast trees dynamically according to the network topology and members joining and leaving the multicasting group (Cain *et al.*, 2002). In the most common multicast tree, the sender or source is the root of the tree, and the receivers, e.g., end users, are connected to the leaf nodes of the tree. The structure of a multicast tree may change dynamically when receivers join or leave the corresponding multicasting session (Cain *et al.*, 2002). Multicasting applications like real-time video conferencing and live video streaming are becoming more common. Further, the performance of such applications relies heavily on the resiliency of the multicast tree technology (Xu *et al*., 1997).

One of the key concerns of real-time multicast traffic is the delay and packet losses due to failures, which can drastically affect QoS. To ensure that the QoS demands can be satisfied, it is central to protect the traffic from the link and node failures. Various mechanisms have been proposed to protect multicasting traffic from link failures for traditional networks (Saidi *et al*., 2006; Zhou and Zhang, 2009; Wei *et al*., 2010; Kotani *et al*., 2012; Farhady *et al.*, 2015).

In these methods, failure notifications are sent via other nodes involved in the multicast session to initiate the protection process. In practice, routing protocols and message flooding mechanisms are used for topology synchronization. During the convergence period, packet losses and higher delay are inevitable and can be significant, depending on the network topology and multicast tree. Among the techniques (Saidi *et al*., 2006; Zhou and Zhang, 2009; Wei *et al*., 2010; Kotani *et al.*, 2012; Farhady *et al.*, 2015), the algorithm presented in Wei *et al*. (2010) explicitly considers QoS constraints and bandwidth protection. However, the algorithm requires the creation of many backup LSPs, which becomes highly complex in practice.

Software-defined networking (SDN) is a fast evolving paradigm that has received tremendous attention recently. SDN separates the network forwarding plane and the control plane, which is often coupled with the physical network devices in traditional networks. The SDN control plane is run in a logically centralized location (Open Networking Foundation, 2012). The separation and centralization of the control plane provides a global view of the entire network for the SDN controller, which can be efficiently used to monitor and control the changes in

the network dynamically. The proposed OpenFlow protocol (McKewon *et al*., 2008) enables the interaction between the controller and the forwarding elements or switches. Using OpenFlow, the controller can install the flow entries to each of the network switches according to the topology information that the controller receives from each switch and the services running on the controller.

SDN also has the potential to simplify the creation of traffic engineering, protection, and multicast trees.

When a failure occurs, notifications can be immediately sent by a switch that detects the failure directly to the SDN controller instead of flooding the network by sending notification messages to every other node or as many nodes as possible. This means that the network can be automatically configured according to the way by which the control plane has been programmed, which significantly reduces the network complexity of traditional networks.

To meet the 50 ms recovery time requirement with SDN, Kempf *et al*. (2012) presented an OpenFlow fault management technique based on the MPLS path protection concept. A secondary backup path can be programmed and installed to flow entries using OpenFlow. In van Adrichem *et al.* (2014), experiment results showed that a path restoration time of 3.3 ms can be achieved using bidirectional forwarding detection (BFD) (Katz and Ward, 2010) to detect link failures. Kitsuwan *et al*. (2015) proposed an approach for failure protection, which reduces the required forwarding table size. Instead of storing backup paths in flow entries, the mechanism makes use of two separate planes, working and transient planes, for data forwarding. The approach considers only link failures. The working plane is used in normal conditions, whereas the transient plane provides routing information for failures. The approach creates link-failure trees; each link-failure tree is calculated to consider possible link failures. They showed that the number of flow entries can be significantly reduced.

However, these approaches do not address protection for multicast applications. For SDN multicast, when a failure occurs, the devices (or device) that detect (or detects) the failure can transmit failure notification message to the controller directly and the multicast recovery process can be started immediately at the controller. The controller can then send the updated multicast forwarding information to switches.

Further, in traditional networks, it is difficult for a network device to efficiently distinguish between a link and a node failure using the routing protocols (Osborne and Simha, 2002). To identify if a node failure occurs, routers have to identify if all links of a particular node have failed. The process is time consuming and will result in high delay and packet losses and degrade the QoS. With the combination of the programmable control plane and the global view of the network topology in SDN, this study is motivated to develop and analyze how fast the control plane can detect, protect, and restore a multicast tree from either link or node failures in SDN.

To meet the objective, we adopt and tailor the subtree-based protection and recovery mechanism proposed in Wei *et al*. (2010) to SDN. The complexity of setting up a number of backup LSPs (Wei *et al*., 2010) that is needed for traditional networks can be significantly simplified using SDN instead of the RSVP signaling protocol for FRR (Osborne and Simha, 2002; Pan *et al*., 2005).

Analyzing and understanding failure restoration for real-time multicasting is crucial, as multicasting becomes popular in practice. We have extended the preliminary work (Renganathan Raja *et al.*, 2015) with more algorithms and detailed analysis. The main contribution of the paper is to investigate the control plane architecture for fast multicast restoration. The controller in our proposed approach responds to a failure by installing updated flow tables to the corresponding switches for the link or node failure in a multicast tree. Our proposed approach to multicast failure protection and restoration can distinguish a link and a node failure in the multicasting session tree constructed by OpenFlow switches. The proposed scheme also responds to the failure by installing or modifying flow entries to the OpenFlow switches for fast recovery. To demonstrate the proposed scheme, we design a prototype in a POX controller and measure failure detection and controller response time by emulating SDN using Mininet (Lantz *et al*., 2010).

2 Related work

In the literature, multicast tree protection and restoration schemes have been advocated for the

optical layer or network layer. This study focuses on protection at the network layer. This section first presents some multicast protection schemes that have been reported in the literature for traditional networks. Next, we describe the key components used in the current POX controller, which plays a key role in failure detection. Following that, we briefly describe an existing approach for multicast protection using fast tree switching in SDN.

2.1 Multicast tree protection

Multicast routing can be realized in the traditional network using protocols such as the Internet Group Management Protocol (IGMP) (Cain *et al.*, 2002), Protocol Independent Multicast Spare Mode (PIM-SM) (Fenner *et al*., 2006), and Multicast Open Shortest Path First (MOSPF) (Moy, 1994). These traditional protocols are operated based on distributed techniques; each node maintains a local view of the entire network topology for path calculations. There are two main issues with this approach, high complexity and delay due to convergence of link state updates.

Some recent work on multicast using SDN has been reported in the literature. Marcondes *et al*. (2012) proposed a clean-slate approach in support of SDN multicast. The approach was proposed to replace existing IGMP (Cain *et al.*, 2002) used in traditional IP networks. Bondan *et al*. (2012) presented another approach which uses IGMP for group management and the calculation of multicast routes. Craig *et al*. (2015) presented an approach to load balancing for multicast traffic in the SDN domain. However, these approaches do not address multicast protection.

In traditional IP networks, multicast protection can be supported by proactive and reactive methods (Pointurier, 2002). In general, reactive methods are considered to be inefficient due to the increase in recovery time. The reason is that the backup paths or trees will be calculated only when a failure occurs and is detected, which will result in high delay. For proactive approaches, the backup paths or trees are precalculated and pre-configured before the failure occurs. FRR (Pan *et al*., 2005) is a proactive approach and has been widely adopted by ISPs in practice to support either link or node failures. However, the number of backup LSPs can be large with FRR if more protection is needed and the subsequent configuration complexity becomes high. Instead of protecting each link or node along an LSP, Pointurier (2002) proposed an algorithm to divide an LSP into segments and apply protections to each segment. Each segment must be well chosen so that it can be protected. As a result, an LSP can be protected if all segments can be protected. In addition, the protection must satisfy the QoS requirements if needed.

A few proactive tree protection methods for multicasting sessions have been discussed (Fei *et al.*, 2001; Saidi *et al*., 2006). Fei *et al.* (2001) proposed the dual-tree algorithm to support protection by switching over the entire primary tree to a preconfigured backup tree when a link failure occurs. The main limitation of the dual-tree algorithm is that only a single link failure is protected in a multicast tree. To overcome the limitation and to accommodate protection for either a link or a node failure, a dual-forest algorithm was proposed by Saidi *et al*. (2006). The node protection scheme is performed by preconfiguring backup paths covering each link involved in the primary tree. The dual-forest algorithm is efficient only when the network topology can provide alternate paths from a node to each leaf of a multicast tree.

Another solution to multicast protection is to use redundant tree protection (Medard *et al.*, 1999; Huang and Guo, 2009). In Medard *et al.* (1999), two disjoint trees were computed at once. Both trees are built in such a way that any node is connected to the common root of the trees by at least one of the trees in the case of a node or link failure. The requirement on the underlying network topology of this scheme is stronger than those of the aforementioned schemes. When a receiver joins or leaves a multicast session, a recalculation is required. This limits the scalability and flexibility of this approach. However, this approach can provide protection against more than one concurrent failure in the tree.

A subtree-based failure protection mechanism was presented by Wei *et al*. (2010). The approach divides a multicast tree into subtrees. Protection is provided for each subtree. As each subtree is protected, the entire multicast tree will be protected. Protecting a subtree can be realized more effectively. The approach has been adapted in the context of SDN. Section 3 presents a more detailed description for the approach.

2.2 SDN topology discovery

Based on the OpenFlow protocol, a topology discovery mechanism is used in the controller to make switches aware of their neighboring nodes. The discovery mechanism has been adopted to detect link or node failures in our proposed approach. Specifically, the topology discovery module in the POX controller (POXTD, 2014) is used for the discovery of a network topology administered by a central controller. The topology discovery module uses the Link Layer Discovery Protocol (LLDP) (Congdon, 2002) to identify the connectivity among network switches. The controller triggers the OpenFlow switches to send LLDP packets between each other via the topology discovery module. When a switch receives an LLDP packet from its neighbor, the switch in turn sends an LLDP packet encapsulated in a Packet-In message to the controller. The Packet-In message consists of both the datapath ID (DPID) and the port number of the sending and receiving switches. Upon receiving a Packet-In message, the controller stores the information embedded in the Packet-In message to establish a link for these two switches. In this way, the controller eventually learns the topology of the entire network in the same administrative domain. In current POX implementation, the LLDP sending operation is triggered at a particular interval of time, which is known as the send cycle time, defined as

$$\text{Send cycle time}=(\text{Link timeout})/2. \quad (1)$$

By default, the link timeout in the current POX implementation is set to 10 s. When a failure occurs, the topology discovery module in the current POX controller will not be triggered until the send cycle time interval expires. In other words, failure detection in the POX controller may be delayed by an entire send cycle time in the worst-case scenario.

Fig. 1 illustrates the scenario. Assume the send cycle time is the difference between t_1 and t_0 (Fig. 1), where t_0 and t_1 are the beginning and end times of a send cycle period, respectively. If a failure occurs at time t_{f1}, then the switches that detect the failure have to wait until t_1, e.g., after the expiration of the current cycle, to send the notification to the controller. This may affect the failure detection time by several seconds, depending on the time at which the failure occurs. The detection mechanism has a considerable negative impact on the recovery time, which results in substantial packet losses and degradation of QoS. The exact value of the delay varies as the failure time, t_{f1}, can be any value that is between zero and the send cycle time. Overall, the failure detection time can be much larger than the actual failure time. Based on measurements for the current version of POX, the topology discovery module takes around 4–5 s on average to update the failed link status. Consequently, the failure recovery time will be significantly larger than the required time, e.g., 50 ms used for existing carrier grade networks. Practically, faster failure detection is often supported using lower layer (e.g., physical layer) techniques (Osborne and Simha, 2002).

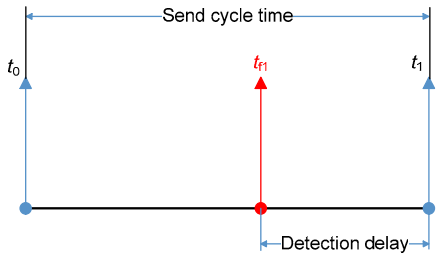

Fig. 1 Detection delay in existing topology

2.3 Existing multicast mechanisms in SDN

Multicasting in SDN has also been discussed in the literature. The traditional IP multicast is highly coupled with IGMP (Cain *et al.*, 2002), which is not scalable due to the large amount of resources needed (Kreutz *et al.*, 2015). An approach was proposed by Nakagawa *et al.* (2012) to manage IP multicast with overlay networks based on OpenFlow. The approach is more scalable as it eliminates periodic join/leave messages. Marcondes *et al.* (2012) presented CastFlow, a clean slate approach to SDN multicast in replacement of IGMP. Bondan *et al.* (2012) proposed an approach called Multiflow for SDN multicast. In their study, however, the IGMP protocol was used for group management and the calculation of multicast routes. Lee *et al.* (2014) described an approach that uses multiple paths to support multicast to achieve robustness, load balancing, and adaptiveness of the applications. The authors reported an improvement in QoS performance for their algorithm using SDN OpenFlow in comparison to the traditional IP multicast. The approach, however, uses multiple paths for video applications for each sink at the same time. In

other words, each sink receives each video stream from multiple multicast trees to improve robustness and load balancing. As a result, network bandwidth usage increases.

Rückert *et al*. (2015) presented Dynamic Software-Defined Multicast (DynSDM) for SDN. The approach can be used for multicast planning and management using a network-layer multi-tree mechanism. The approach also supports load balancing on links inside the ISP domain with multiple multicast trees. Further, DynSDM considers dynamic changes to network conditions, such as congestion or link failures. The failure recovery approach adopted by the approach is reactive; i.e., new subtrees are calculated when a failure is detected and the affected trees are identified. Reactive approaches could cause packet losses, which may not be acceptable for high-speed networks or high QoS demands.

Pfeiffenberger *et al*. (2015) evaluated reliable multicast communications using OpenFlow 1.1. The approach uses the fast-failover feature specified in OpenFlow 1.1 to provide single link fault tolerance. The authors proposed to use virtual local area network (VLAN) tags to distinguish multicast trees for packet forwarding. The approach uses a local failover recovery mechanism supported by OpenFlow 1.1, but it does not address multiple link failures or node failure(s). For node failure(s), the controller needs to ensure that node failures indeed occur. Further, only qualitative results are illustrated in the verification and a complete redundant multicast tree may not be available.

In Kotani *et al*. (2012), multicast protection was performed by fast tree switching. A redundant tree is calculated as soon as the primary tree is calculated and flow entries for both primary and redundant trees are installed in the switches. To avoid duplication of the flow tables for the same destination, they are differentiated by using unique IDs. If a failure occurs in a link in the primary tree, the whole tree will be switched to the pre-calculated redundant tree. This limits the approach from supporting more than one link failure. In addition, a complete diverse redundant tree may not be available for some topologies.

2.4 GroupFlow, multicasting for software-defined networks

GroupFlow (Craig, 2014) implements the Internet Group Management Protocol (IGMP, version 3) on a POX controller to support multicasting for SDN. This project is one of the few available resources involved in implementing multicasting for SDN. There are three main components in GroupFlow: GroupFlow module, IGMP manager module, and topology discovery module (an existing component in POX). These three components react to the dynamic changes in the network, such as changes in network topology and incoming multicasting traffic, handling and managing the IGMP v3 packets and installing flow tables to the OpenFlow switches to route the multicasting traffic.

The GroupFlow module is the component that routes the multicasting traffic by installing flow tables to OpenFlow switches. The component also triggers path calculation when it receives updates for topology changes or group membership changes.

3 Subtree-based failure protection for a multicast tree

Our study is based on the proactive subtree-based protection scheme for a multicast tree (Wei *et al*., 2010). To tackle the problem of switching the whole multicast tree in case of a link failure, a subtree-based protection scheme for multicast session was presented for traditional networks. Based on the approach, when a multicast tree is built from the source to destinations, the multicast tree will be divided into multiple subtrees.

Note that a multicast tree is a subset of a network topology; it consists of all the nodes that are involved in multicast session, but only a subset of the links of the original network. A multicast tree is generated based on a network topology, the source of the multicast session, and typically a spanning tree algorithm. Wei *et al*. (2010) assumed that a multicast is generated. The subtree-based protection increases flexibility in meeting the QoS requirements and localizes the protection in a relatively small physical area of the multicast tree.

A subtree is defined as a subset of the (multicast) tree that itself is also a tree. Any node in a tree T, together with all the nodes below T, comprises a subtree of T. Note that in this study, the term ‘subtree’ refers only to a tree whose root has more than one direct child, unless otherwise explicitly noted.

The following example shows how the division into subtree works. The tree (Fig. 2) can be divided into four subtrees: subtree 1, rooted at node 3; subtree 2, rooted at node 9; subtree 3, rooted at node 7; subtree 4, rooted at node 1.

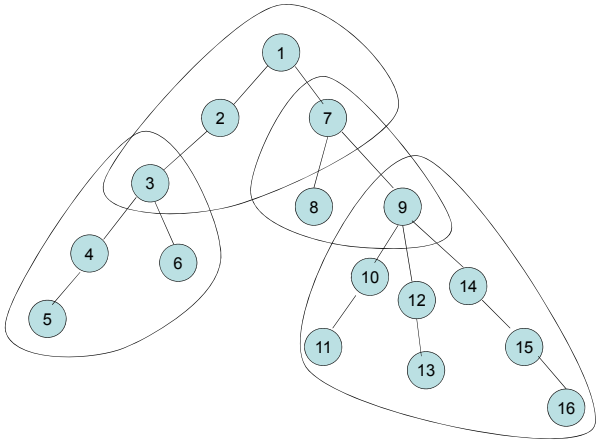

Fig. 2 Subtree division: an example

These subtrees are referred to as subtrees 3, 9, 7, and 1, respectively. Subtree 7 consists of node 9, node 8, and subtree 9; subtree 9 is also a subtree of subtree 7; subtree 3 does not contain any other subtree, neither does subtree 9. Note that the tree *T* rooted at node 1 can also be viewed as a subtree according to the definition. Subtree 1 consists of node 1, subtree 3, node 2, and subtree 7 (which, in turn, consists of subtree 9 and node 8).

After dividing a large tree into smaller subtrees, the next step is to find an algorithm to protect each of the subtrees. In analyzing the structure of the tree as shown in Fig. 2, it is noticed that the large tree *T* rooted at node 1 can be protected once subtree 3, subtree 7, and node 2 are protected. Subtree 7 can be subsequently protected by protecting subtree 9 and node 8. A recursive bottom-up approach has been proposed in the subtree-based protection method to protect the entire tree *T* rooted at node 1.

There are advantages using the subtree-based approach:

1. It minimizes the failure detection time by avoiding the notification to be sent all the way to the root of the multicast tree for failure recovery.

2. It makes the protection scheme efficient by dividing the tree into smaller ones and providing backup paths from the root of each subtree to its leaf nodes.

3. When a failure occurs, the changes are made only to the corresponding subtree instead of the entire tree.

The goals of protecting a large tree and meeting the QoS requirements can be achieved if each of its subtrees is protected and meets the requirements. Since the smaller subtree is more localized and the topology of the subtree is simpler than that of the large tree, finding better protection and setting up protection paths become easier than protecting the entire tree.

The protection process of the tree as shown in Fig. 2 is described as follows: subtree 9 is the first subtree to be protected and subtree 3 the second, as both of them are at the bottom of the tree *T* and contain no other subtrees. The next step is to move up one level, i.e., subtree 7 for subtree 9. Subtree 7 contains subtree 9, which has been protected in the previous step. Subtree 9 will be considered as a protected node in subtree 7 when considering the protection for subtree 7. It now becomes simple to protect subtree 7 by protecting links <7, 8> and <7, 9> without having to consider the links in subtree 9, as they are protected when subtree 9 is protected. Tree *T* now consists of the already protected subtree 3, subtree 7 (containing subtree 9), and the unprotected links <1, 2>, <2, 3>, and <1, 7>. After protecting these three unprotected links, *T* is entirely protected.

The protection approach can be looked at in another way. Protecting subtree 7 equals protecting subtree 9, plus protecting the nodes (nodes 7 and 8) belonging to subtree 7 but not to subtree 9. When protecting subtree 9, subtree 9 contains no subtrees. Subtree 7 has one and only one subtree, which is subtree 9. Because subtree 9 was already protected in the previous step, subtree 7 has no unprotected subtrees. If a protected subtree (subtree 9) is considered to be a protected node while the protection is applied to its parent tree (subtree 7), it is found that protecting a tree with no subtrees (e.g., subtree 9) and protecting a tree with no unprotected subtrees (e.g., subtree 7) are essentially the same. Thus, a tree with no unprotected subtrees is defined as the smallest protection unit in this study, and the goal becomes finding an approach to protecting each of the units.

By dividing a large tree into subtrees and protecting the subtrees using a bottom-up approach, the large problem is not only divided into small pieces,

but it also becomes clear that only one protection algorithm is needed to protect a large tree. Moreover, each protecting unit is more physically localized than the other approaches to multicast protection. The path protection, dual-tree protection, and redundant tree protection described earlier in this section all protect a tree without dividing the tree in any way; they are theoretically correct but practically difficult to adopt. Finding a backup path for a pair of nodes that are far away might not be possible or economic. Also, these approaches do not consider the structure of the tree in a QoS-aware manner.

The subtree protection scheme is based on the assumption that each path in a subtree is protected (Wei *et al*., 2010). In a subtree, each branch can be viewed as a unicast label switched path (LSP) (Pan *et al*., 2005). There are approaches to protecting unicast LSPs, one of which is the segment-based approach (Pointurier, 2002) which can provide a bounded switchover time, but it needs to be modified for multicast protection. Once each of the branches is protected and satisfies the requirements, a subtree is protected. If all subtrees of a node are protected, the parent node is protected. The process can be repeated until all subtrees of the root are protected.

MPLS-based LSPs can be set up explicitly with QoS constraints, providing more flexibility and leverage in comparison to IP-based traffic engineering techniques. However, control-plan signaling protocols, e.g., RSVP-TE, are needed to establish backup LSPs to support FFR (Pan *et al*., 2005). An LSP used for FRR is typically signaled using RSVP by the head-end router to downstream routers hop by hop along an explicit path. Each hop has to perform admission control and call setup for the request. Further, the state of each LSP needs to be periodically refreshed using the RSVP protocol to keep the LSP alive, as RSVP is a soft-state protocol. For instance, consider a scenario for a path setup for *A->B->C->D*, where *A* is the head-end router. Node *A* sends a request message to *B*, node *B* sends a subsequent request to node *C*, and node *C* does the same thing to node *D*. If successful, node *D* then sends a reply message back to *C*, and *C* sends a reply message to *B* until *A* receives a reply message. Consequently, an LSP *A->B->C->D* is set up successfully. After a path is set up, neighboring routers have to periodically exchange hello messages to keep the path alive.

Wei *et al*. (2010) presented how backup paths with bandwidth protection could be established, even with bandwidth optimization. The readers are referred to Wei *et al*. (2010) for detailed discussion on backup bandwidth optimization in a traditional network.

However, the path setup process is complicated from the network management and control overhead perspectives (Sharafat *et al.*, 2011; Das, 2012; Akyildiz *et al*., 2014). The complexity becomes higher if many backup LSPs need to be created for a multicast tree protection, as required by the subtree-based protection scheme. Therefore, the subtree-based protection scheme for multicast applications has limited benefits in the traditional IP/MPLS networks.

On the other hand, SDN separates data and control planes. The simplicity of SDN control can significantly mitigate the complexities of the MPLS control plane (Das, 2012; Akyildiz *et al*., 2014). With a global view of the entire network at the SDN controller and using the simplified OpenFlow control plan, it becomes much easier to establish LSPs (Sharafat *et al.*, 2011; Das, 2012; Akyildiz *et al*., 2014), even multiple LSPs for subtree protection. For the aforementioned example, the controller can perform a path calculation and send the flow entry information to nodes *A*, *B*, *C*, and *D* to set up an LSP *A->B->C->D* without having to use RSVP to set up a path hop by hop. Hence, the complexity of the proposed subtree protection mechanism can be considerably reduced.

In summary, the failure detection and restoration time are critical, and the protection scheme must support protection for both link and node failures for a single multicast session tree. However, the algorithm proposed by Wei *et al*. (2010) has several limitations: (1) it was proposed for the traditional network; (2) it considers only a single link failure; and (3) the control overhead of establishing and maintaining backup LSPs can be considerably high in practice. We have tailored the subtree-based protection mechanism and applied it to the SDN domain to support both link and node failures.

4 Multicast tree protection and restoration for software-defined networks

This section presents our proposed subtree-based protection and restoration scheme for SDN

based on the scheme discussed in Wei *et al*. (2010). We have tailored the algorithm for SDN and extended the scope of the algorithm from a single link failure to link failures or node failures.

The main component that has been added is the failure localization and protection module (Fig. 3). Fig. 3 demonstrates the high-level structure of SDN with the OpenFlow POX controller for multicast tree protection and restoration. The topology discovery module is already available in the POX controller, playing a central role in identifying the connectivity for OpenFlow switches. The topology discovery module periodically sends Link Layer Discovery Protocol (LLDP) packets to OpenFlow switches and maintains the adjacency list for the switches. If the topology discovery module detects a failure, the module will send the failure information to the proposed failure localization and protection module, which in turn will send the flow tables to the OpenFlow switches according to our proposed scheme.

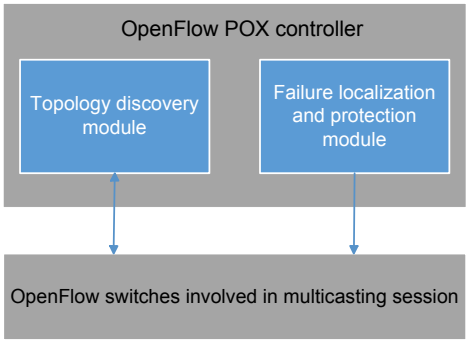

Fig. 3 Proposed OpenFlow controller architecture for failure detection and protection

Several assumptions have been adopted for our proposed approach, which will be described next. Following that, the proposed subtree-based protection and restoration scheme for SDN is presented in Section 4.2.

4.1 Assumptions

Four assumptions have been made for our proposed scheme. The first two basic assumptions adopted are inherited from the subtree-based approach:

Assumption 1 A multicast tree has already been established.

Some recent works on multicast with SDN have discussed this topic, e.g., Marcondes *et al*. (2012) and Bondan *et al*. (2012). Marcondes *et al*. (2012) presented a clean-slate approach to handling multicast with SDN. The approach was proposed to replace existing IGMP (Cain *et al.*, 2002) used in traditional IP networks. Bondan *et al.* (2012) proposed another approach which uses IGMP for group management and the calculation of multicast routes. Rückert *et al*. (2015) proposed DynSDM for multicast group management using multiple multicast trees. For our proposed approach, a multicast tree is assumed available using any technique, since our focus is on the protection of a multicast tree.

Assumption 2 A backup LSP has been established from the root of a subtree to each of the leaf nodes.

The backup paths, however, are not used in normal operation; i.e., they are used only when needed for a failure recovery. As stated in Section 3, using SDN, the creation of backup LSPs can be considerably simpler than that for traditional MPLS/IP networks. For this reason, our approach is not suitable for a very sparse topology where diverse backup paths are difficult to create due to the limited alternative options.

Another two assumptions are adopted for our proposed scheme for SDN, as highlighted in the following:

Assumption 3 The entire network has a single OpenFlow controller.

The motivation of our research is to validate the behavior and the performance of the protection and restoration algorithm in SDN. A single controller is also common to many approaches in SDN in the literature.

Assumption 4 All nodes used in our investigation are OpenFlow switches and each switch has a direct connection to the central SDN controller.

Hence, we have avoided representing the connection between each switch and the controller in the figures.

The assumption on direct connection from a switch to the central controller can be relaxed to direct logical connection instead of physical connection without affecting our approach, except that the communication delay will become slightly larger. In addition, multiple controllers may be available. The main concept of our proposed approach is still applicable in this scenario, as each controller is responsible for a subset of nodes. Using multiple

controllers can reduce the average propagation time, as each switch will generally become geographically closer to a controller.

Nodes can dynamically join and leave a group in a multicasting session. In this study we do not address this issue directly. To protect a dynamic multicast tree, backup paths for the newly joined members need to be created and backup paths for the members that have left also need to be removed. Maintenance of the tree due to joining/leaving can be supported using either a global or a local approach. A global backup approach refers to the case where backup protection is needed for the entire primary tree. However, even though this issue is not addressed explicitly, based on the proposed subtree protection scheme, a global backup maintenance is not necessary. Instead, backup maintenance is required only for the subtree whose structure has changed during the primary tree reconstruction phase.

4.2 Multicast tree protection and restoration method for the OpenFlow controller

This section describes the five major operations of our proposed multicast tree protection and restoration algorithm for the OpenFlow controller, i.e., subtree division, failure detection, failure localization, failure protection, and flow modification.

4.2.1 Subtree division

This is the initial stage of the entire protection and restoration algorithm. In this stage, the unstructured tree information from the emulated topology is sorted and stored in a structured and organized manner. The need for tree sorting in this operation is due to the structure and the order of information that the controller receives from the POX topology discovery module. This is performed by using hash tables, where keys and values are used to identify parent and children nodes of the tree.

When the switches are added to the network, the topology discovery module creates link information between the switches according to the way by which they are connected with each other. The link information is created based on the LLDP packets sent by the OpenFlow switches to the controller. Each switch sends this information to the controller using the Packet-In messages. The controller on receiving the Packet-In message from the switches creates a table with the link information between the two switches. The link information sent is as shown below:

Link [“DPID1”, “port1”, “DPID2”, “port2”].

DPID1 and DPID2 are the data path IDs of the switches which share the link, port 1 is the port for switch DPID1, and port 2 is the port for switch DPID 2. This information is again generated by the controller in a reversed manner when the switch with DPID 2 sends the Packet-In message to the controller. The data is stored in a hash table with switch DPIDs as its keys and the links as its values. Hence, the first step of the tree sorting process is to delete the duplicate link information for a single link between two switches (POXST, 2016). As a result of this step, we form a two-dimensional hash table which has single link information for each pair of switches.

The next step is to reform the sorted link information to parent and child relationship. This step begins with the isolation of the sorted tree from the topology discovery module. The isolation is necessary to sustain the tree information for later stages when a link failure occurs. After the tree is isolated, an iteration through the tree is initiated. By iterating through the tree, the parent and children nodes are separated. The parent nodes are saved in a list and the children nodes are saved in a hash table with their respective parent DPID as the key. The parent and children information is then passed to the subtree division module.

Pseudo code for subtree division

Objective: to divide an existing multicast tree into subtrees.

Components:

Parent_nodes: a list of DPIDs of all the parent nodes in the tree. The first element in this list is the root of the whole multicast tree.

Child_nodes: a hash table which has parent DPID as the key and children DPIDs as its values.

Subtree: a hash table which has the subtree root DPID as the key and DPIDs of members of the subtree as its values.

Subtree_key: a list of root nodes of divided subtrees.

Subtree_search(): a recursive function which starts searching from the root of the tree and divides the current tree into subtrees. First it is invoked with the root node and later it is recursively invoked with

the current node and the root of the current subtree.

Root_node: the root node of the whole tree, i.e., the first element of the Parent_nodes list.

Cur_node: the current node being iterated.

Algorithm 1 Subtree division

```
Input: Parent_nodes, Child_nodes.
Output: Subtree, Subtree_key.
  Root_node=Parent_nodes[0]
  Subtree_search(Root_node, Root_node)
  Subtree_search(Cur_node, root)
  while (Child of Cur_node is not null) do
    if (Child_nodes of Cur_node>1) then
      Subtree[Cur_node].add(Child_nodes of
       Cur_node)
      Subtree_search(Child_nodes of Cur_node,
       Cur_node)
    else
       Subtree[root].add(Child of Cur_node)
       Subtree_search(Child of Cur_node, root)
    end if
  end while
```

The subtree algorithm starts searching the tree from its root. It stores the divided subtree in a hash table where the root of the subtree is the key and its members are the values. The root of a subtree is defined when a node has more than one child. The members of a subtree are added to the subtree until the search algorithm identifies a node which has more than one child. It then saves all the roots of the subtree in a list. This is to make the search process efficient when a link failure occurs. The operation of the subtree algorithm is described in the aforementioned pseudo code.

We have implemented the algorithm using lists and dictionaries in Python on a POX controller. The roots of subtrees are stored in a list and each subtree is stored in a form of Python dictionary, which again has a list as its values. The time complexity for a hash table lookup is $O(1)$ for either the average case or the worst case. The time complexity for a lookup in a Python dictionary is $O(1)$ for the average case, but $O(n)$ for the worst case (Python, 2015).

For a multicasting tree as shown in Fig. 4, the subtree search algorithm starts from the root of the multicasting tree, i.e., node S1. It checks if S1 has more than one child node. If yes, it creates subtree 1 and adds S1 as the root of the subtrees and S2 and S3 as the members of subtree 1. Then the search process continues by checking S2's children. Here S2 has one child and hence the search algorithm adds the child of S2, i.e., S4, to the member of subtree 1. Since this process is recursive, the search process hits the leaf of the tree before it continues searching the other side of the root. Hence, the search algorithm traverses to node S4 and checks S4's children. Node S4 has more than one child, i.e., S7 and S8. Therefore, in this condition subtree 2 has been created with S4 as its root and S7 and S8 as its members. Since S7 and S8 have no children, they are not considered as the root and the search process continues with the other side of S1 by continuing searching S3. In this manner, the whole subtree has been divided. The divided subtree is stored in the form of hash table, with the root node as the key and members as values. For instance, subtree 1 has S1 as the key and S2, S3, and S4 as its values.

4.2.2 Failure detection

The dynamic changes in SDN are monitored by the control system running in the controller. Link and node failures are among the most important reasons that cause changes to the network. Hence, both link and node failures have to be dealt with efficiently. The global view of the controller makes failure detection more efficient, as switches do not have to flood the network with messages for topology synchronization. The convergence period could be long using the flooding mechanism, which results in packet losses and long delay. Our approach is for concept demonstration for subtree protection and we use existing functionalities in POX. Failure detection is conducted using the POX topology discovery module (POXTD, 2014).

The topology discovery module plays a key role in identifying the connectivity between OpenFlow switches by periodically sending LLDP packets. After the discovery module sends out LLDP packets, it monitors the arrival of the returned LLDP packets from other switches. The topology discovery module maintains the adjacency list, which has the information about switches and their connections with their neighbor nodes. The topology discovery module raises a LinkEvent whenever there is a change in the status of the link associated with the two OpenFlow switches. The links here are layer 2 Ethernet interfaces of the OpenFlow switches. Whenever a switch does not send the LLDP packet associated with the port connected to its neighbor, the controller will

consider that the link is timed out and will fire the LinkEvent for link removal.

The failure detection time starts from the time at which the topology discovery module in the controller receives the failure notification, not the actual time at which a failure occurs. This is due to the limitation existing in the current POX topology discovery module, as explained in Section 2.2. The current version of POX is designed for emulation purpose and the discovery module could take around 4–5 s to update the failed link status based on measurements. For real products, a mechanism that can immediately notify the failure is often used, such as lower layer techniques (Osborne and Simha, 2002).

4.2.3 Failure localization

Failure localization begins after the failure is detected. It is the process of identifying where the failure exactly occurs and what kind of protection should be provided for the failure. Our proposed tree localization and protection module listens to the topology discovery module by registering itself to the core of the POX controller. The localization module handles the LinkEvent by capturing the events from the topology discovery module. The link failure is captured by using the event.removed part of the LinkEvent. This is triggered when the link is timed out. By capturing the event.removed, the controller gets the link information of the switches that share the link. The link information contains the DPID and the port number of the two switches that share the link. Using the DPIDs from the link information received, the algorithm searches the failed switches in the divided subtree to find to which subtree the failure belongs. The reason to do this is to identify the subtree for the changes. This search process avoids unnecessary changes to the whole tree upon a failure.

The subtree search algorithm considers three scenarios to exactly identify the location of a failure in a subtree. The three scenarios are explained below:

1. When either one of the DPIDs is the root of a subtree.

2. When each of the two DPIDs is a root of a subtree.

3. When each of the two DPIDs is a child/descendant of a subtree.

These three scenarios are important to decide where to provide protection and what kind of protection to provide for a failure event.

Scenario 1 Either one of the DPIDs is the root of a subtree.

There are a couple of cases in which either of the two switches is the root of a subtree. Consider the tree shown in Fig. 4 where a failure occurs during S3–S5 or S2–S4.

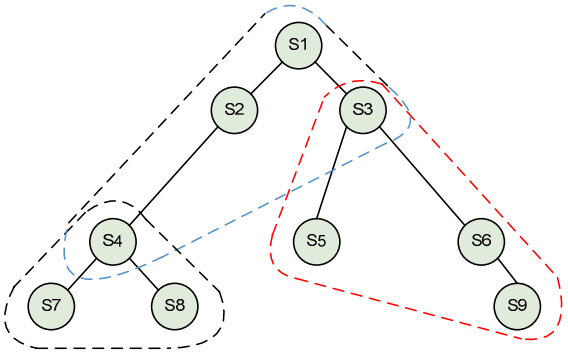

Fig. 4 A sub-divided tree of OpenFlow switches

Case i: A node is the root of a subtree, e.g., S3, and the root is also the parent of the other node, e.g., S5. If a failure occurs between S3 and S5, the controller receives link failure notification between them and will extract the DPIDs of the two switches. With the extracted DPIDs, the localization module searches if either of the DPIDs is in the root list of the subtree. In this case, S3 is the root but S5 is not. Then it checks if the node which is not the root of a subtree, e.g., S5, is also a member of the root node, e.g., S3. Thus, it checks if S5 belongs to S3. If yes, the localization module forwards the information to the protection module with root node S3 as its starting point.

Case ii: A node is a root of a subtree, e.g., S4, and the root node is also a child node of the other node, e.g., S2. If a failure occurs between S2 and S4, S4 is the root of a subtree. However, the localization module also checks if S2 belongs to subtree S4. In this case, S2 does not belong to S4, so the localization module sends the root ID of S2 (in this case it is S1) to the protection module.

Scenario 2 Each of the nodes is a root of a subtree.

Consider that a failure occurs between S1 and S3, which are both root nodes. When the localization module identifies that each of them is a root of a subtree, the localization module identifies the relationship between these two nodes; i.e., it searches which node belongs to which subtree. Specifically, it searches whether S1 belongs to S3 subtree or S3 belongs to S1 subtree. For this example, S3 belongs to

S1 and hence the localization module initiates the protection module with S1 as the starting point.

Scenario 3 Each of the nodes is a child/descendant of the same subtree.

If the localization module is not satisfied with the above two scenarios, it considers that the DPIDs of the switches are just members of a subtree. The localization module then verifies to which subtree one of the nodes belongs, and initiates the protection module with the root of the subtree where the failure occurred as the starting point. For instance, assume that a failure occurs between S6 and S9 as depicted in Fig. 4. S6 and S9 belong to the same subtree rooted at S3. The search process will then result in sending S3, the root of the subtree, as the starting point for the protection module.

The next step is to analyze the time complexity. From the algorithm analysis perspective, to locate a failure which involves a root node, it takes $O(1)+O(1)+O(1)$. The reason is that the root nodes are stored in a list and hence the time to get an item from the list is $O(1)$ using a hashing function. After getting the node ID (which serves as the key of the hash table), the time complexity to obtain the item from the Python dictionary is again $O(1)$ (Python, 2015). However, when a failure occurs between the non-root nodes, the algorithm does not know the key of the hash table in the first place and hence it has to traverse through the whole dictionary to find the key value pair. So, the time complexity in the average case for that search is $O(1)+O(1)$ and in the worst case it is $O(1)+O(n)$, as the time complexity for a lookup in a Python dictionary is $O(1)$ for the average case, but $O(n)$ for the worst case, where n is the number of nodes stored in the associated Python dictionary (Python, 2015). From the practical perspective, the number of routers in the range of a typical open shortest path first (OSPF) area is around 50, as recommended by Cisco (Bondan *et al*., 2012), which means that each path in a multicast tree does not have a large number of nodes. Therefore, the algorithm is considered efficient even though the current algorithm focuses mainly on concept demonstration.

Pseudo code for failure localization

Objective: to identify at which part of the subtree the failure has occurred and to initialize the protection process according to the switches involved in the failure.

Components:

Failed_switches: a list of DPIDs of the switches involved in a failure event.

Failed_links: a list of the links involved in the failure. A link has DPID and the port number of two switches involved in the failure.

Subtree_root: a set of root nodes of the divided subtrees.

S1 and S2: switches that are directly connected to the failed link.

Algorithm 2 Failure localization

```
Input: Failed_link(s).
Output: invoking protection function with the exact switch where the flow table has to be installed.
for all switches in Failed_switches do
  if (S1 or S2 is in Subtree_root) then
  // check if either is a subtree root
    if (S1, but not S2, is in Subtree_root) then
    // S1, but not S2, is a root
      execute Failure_protection(S1, S1, S2)
    else if (S2, but not S1, is in Subtree_root) then
    // S2, but not S1, is a root
      execute Failure_protection(S2, S1, S2)
    else
  // both S1 and S2 are in Subtree_root
  // need to determine parent / child
      if (S1 is the subtree root of S2) then
        execute Failure_protection(S1, S1, S2)
      else if (S2 is the subtree root of S1) then
        execute Failure_protection(S2, S1, S2)
      end if
    end if
  else
  // neither S1 nor S2 is the root
    find the Subtree_root of S1 and S2
    execute Failure_protection(Subtree_root, S1, S2)
  end if
end for
```

There are multiple failure scenarios. The nested 'if' statements presented above are used to identify each scenario and trigger the failure protection process accordingly. All the steps to localize the failure explained earlier require tree isolation. The reason is that when a failure occurs, the link information is deleted from the topology discovery module and hence will change all the information to hash tables associated with it. Without isolation, the entry in the hash tables will also be removed and the search process cannot identify the location of the failure in the subtree.

4.2.4 Failure protection

Failure protection is initiated after the failure is localized. This module performs two major functions. One is to determine whether the failure is a link or a node failure and the other is to send flow table modifications to the switches responsible for protecting the switches from failure. For traditional networks, network nodes cannot efficiently distinguish link or node (neighbor) failures with the routing protocols. However, with SDN, the controller can distinguish these two failures, as switches can send notification directly connected to the controller. The difference is described below.

Whenever the failure is detected by the LinkEvent handler, the ports associated with each switch involved in the failure are added to failed_ports. The failed_ports is later used to count if the number of ports the switches involve in the failure is equal to the total number of ports in that switch; i.e., all ports of a switch have failed. When the result is true, we initiate the node failure function. If the count is different, we consider it as a general link failure.

1. Node failure

This function performs two quick searches. One is to identify if the node which failed is a root node. If the node is a root, then the controller initiates the flow installation function with the root of the higher level subtree to which the failed node belongs. If the node is not a root, the controller initiates the flow installation function with the root of its own subtree. For example, in Fig. 4, if node S4 has a node failure, then the controller will respond to S1, the root of the subtree. If S6 fails, the controller will respond to node S3.

2. Link failure

A link failure is initiated if the condition for a node failure is false. For a link failure, the controller just initiates the flow installation function with respect to the node DPID that it gets from the flow localization module. For example, in Fig. 4, when a link failure occurs between S3 and S5, the link failure function will make changes to S3, as it is the root node of the subtree where the failure occurred.

4.2.5 Flow modification

Flow modification is the final step of the multicast tree protection and restoration algorithm. It is invoked once the failure has been decided as either a link or node failure. The flow modification function installs new flows to switches based on the DPIDs of the failed link or node it receives from the failure protection function. Since we are emulating only the tree topology in Mininet, we install flow tables to the responsible switch to show that the controller notifies it successfully.

When the flow modification function receives the DPID of the switch where the new flow has to be installed, the controller creates the OpenFlow flow modification message with a unique cookie ID. According to OpenFlow, cookie is an identifier for a flow table installed in an OpenFlow switch. Each flow table installed in an OpenFlow switch will have a unique cookie ID, and the protection module sets the DPID of the switch involved in the failure as the cookie ID of the flow table installed to recover the failure. This is done to remove the flows when the primary link is up again. Then the OpenFlow flow modification messages with the cookie ID and action messages are sent to the corresponding OpenFlow switches.

Pseudo code for failure protection and restoration

Assumption: Backup paths are available from the root of each subtree to its leaf nodes.

Objective: to determine if it is a link or node failure and to modify or install flow tables to the corresponding switches.

Components:

Node_failure(): a method which determines the switch where the flow table should be modified in case of node failure. It triggers the flow installation operation.

Link_failure(): a method which determines the switch to which the flow table should be modified in case of link failure. It triggers the flow installation operation.

Failure_protection(): a method which determines if the failure is a link or node failure and triggers the corresponding action for the failure.

Flow_installation(): a method which installs or modifies the flow tables. It takes the root node as an argument to install flows to recover from the failure.

failed_ports: it tracks the number of ports that fail in each switch.

fs, fs1, fs2: DPIDs of the switches involved in the failure. This argument is passed by the Failure_ Localization module.

failed_root: the root node identified by the Failure_localization module in which flows are to be installed.

max: the maximum number of ports in a switch.

Algorithm 3 Failure protection

```
Input: failed_root, fs1, fs2.
Output: invoking link failure or node failure recovery.
  if (failed_ports of fs1 or fs2==max) then
    if (failed_ports of fs1==max) then
      Node_failure(failed_root, fs1)
    else
      Node_failure(failed_root, fs2)
    end if
  else
    Link_failure(failed_root)
  end if
```

Algorithm 4 Node failure

```
Input: failed_root, fs (DPID of the failed switch).
Output: invoking flow installation.
  if (fs is not in Subtree[root]) then
    Flow_installation(failed_root)
  else
    for (all roots in subtree)
      if (failed_root in Subtree[root]) then
        Flow_installation(root)
      end if
    end for
  end if
```

Algorithm 5 Link failure

```
Input: failed_root.
Output: invoking flow installation.
  Flow_installation(failed_root)
```

Algorithm 6 Flow installation

```
Input: root.
Output: sending flow modification messages to the switches.
  target=root
  message=OpenFlow_mod()
  message.command=Add flow table
  connection=OpenFlow.getconnection(target)
  connection.send(message)
```

5 Experimental results

This section presents the experimental results and analysis of the proposed multicasting failure localization and protection algorithms with respect to the failure localization time and the failure recovery time using the Mininet environment.

5.1 Experiment setup

The experiments are focused on failure localization and protection algorithms, which are implemented in the OpenFlow POX controller. Failure detection is realized by obtaining the information from the existing topology discovery module in POX.

The real instances of OpenFlow switches representing a multicast tree session and the POX controller are emulated in Mininet (Lantz *et al*., 2010). Mininet is a network emulation framework which emulates real instances of OpenVSwitches and an OpenFlow controller. We run Mininet on a virtual machine (VirtualBox 4.2.16) which is running a Windows 7 64-bit operating system with 3.40 GHz Intel i7-3770 CPU and 16 GB RAM.

Fig. 4 shows the network topology, with nine OpenFlowVSwitches connected to the POX controller. Note that although the network size is not large, the concept can be applied to large networks. The reason is that the proposed approach deals with subtrees. Therefore, the performance of the algorithm is closely related to the subtree size, not the network size. Further, a multicast tree can typically be divided into smaller subtrees. In addition, each node is either directly connected to the controller or can reach the controller with a small delay in an SDN environment in principle. If a failure occurs, only the corresponding subtree needs to perform the recovery tasks. Nodes in other parts of the network are not affected.

All switches and the controller are running in the same virtual machine. The controller is running the topology discovery module, multicasting failure localization module, and protection module. The performance of the controller is evaluated by observing the failure localization time and the failure recovery time after the failure has been detected. We create failures by randomly making the link between two switches down using Mininet's Command Line Interface. We disconnect each link between each pair of switches and repeat the experiment 10 times per pair. We calculate the average failure localization time and failure recovery time. The results of the experiments are discussed in detail in the following.

5.2 Experimental results

There are three main components for failure recovery: failure detection time, failure localization time, and the subsequent failure response time that the

controller takes to send the flow table update to corresponding switches.

Failure detection time starts from the time at which the topology discovery module in the controller receives the failure notification. Failure detection is based on the implementation of the existing topology discovery module in POX. The current POX is primarily used for emulation and functionality demonstration; performance is not a main concern for POX. In practice, failure detection is more efficient using lower layer techniques as opposed to using a higher level software module (Osborne and Simha, 2002). For the POX emulator, failure detection time could take 4–5 s based on measurements, which is much higher than industry standards. Therefore, for our experiments, failure detection time is excluded. Instead, we focus on the performance of the subtree protection algorithm, i.e., the time at which the controller starts to identify the failure location in the multicast tree to the time at which the recovery is completed.

The other two components and results are presented as follows:

1. Failure localization time

Failure localization time is the time interval from the time when the failure localization and protection module, as illustrated in Fig. 3, gets link failure information from the topology discovery module, to the time when it finds the exact failure location in a multicast tree. Specifically, the task is to search through the hash tables where the subtrees are stored and identify the failure location of the multicast tree.

Table 1 depicts the results of failure localization time for failures between different pairs of switches. The results are average of multiple runs. The variation is small among different experiments. As depicted in Table 1, the average failure localization time is small, mostly less than 40 μs.

Based on the algorithm, the delay for failures that involve a root node generally is higher than that of other failures, because the search process is performed on both the hash table and subtree root lists to identify which node is the root and which node is a member of the root. Specifically, the cases based on the scenarios and cases depicted in Section 4.2 are described in the following.

Scenario 1, Case i: A failure occurs between a root and a non-root node and the root node is the parent, e.g., S1–S2, S3–S5, S3–S6, S4–S7, and S4–S8. For this case, the controller receives the failure notification and will extract the DPIDs of the two switches. With the extracted DPIDs, the localization module searches if either of the DPIDs is in the root list of the subtree. Then it checks if the other node is a member of the root node. In this case, the result is positive; therefore, the localization module forwards the information to the protection module with the root node as its starting point.

Table 1 Average failure localization time (T_L)

Link	Average failure localization time (μs)	Standard deviation (ms)
S1–S2	40.5	0.042
S1–S3	33.6	0.051
S2–S4	65.1	0.048
S3–S5	44.2	0.013
S3–S6	15.9	0.004
S4–S7	23.8	0.022
S4–S8	26.6	0.024
S6–S9	18.5	0.042

Scenario 1, Case ii: A failure occurs between a root and a non-root node and the root is the child, e.g., S2–S4. For a failure between S2 and S4, S4 is first identified as it is in the root list. Next, the algorithm checks if the other node, S2, belongs to the subtree rooted at S4. In this case, S2 does not belong to S4; hence, the localization module sends the root ID of S4, i.e., S1, to the protection module.

Scenario 2: A failure occurs between two nodes that are both root nodes, e.g., S1–S3. In this scenario, the localization module identifies each of them as a root of a subtree. It then identifies the relationship between these two nodes; i.e., it searches which node belongs to which subtree. For this example, S3 belongs to S1 and hence the localization module initiates the protection module with S1 as the starting point.

Scenario 3: Each node is a member node of the same subtree, e.g., S6–S9. For this scenario, the algorithm does not know the key of the hash table and hence has to traverse through the whole Python dictionary to find the key value pair. Hence, the result is dependent on the size of the dictionary. As described in Section 4.2, the time complexity in the average case for that search is $O(1)+O(1)$ and in the worst case

it is $O(1)+O(n)$. However, the number of nodes in a practical OSPF domain is not large (Tiso, 2011).

Overall, the computation of the proposed SDN subtree algorithm is fast using hash tables and the algorithm is scalable, as hash table calculations can be efficiently realized for large sizes.

2. Failure recovery time

Failure recovery time is the total time taken from the time at which the controller receives a failure notification from the topology discovery module to the time at which the new flow tables are installed in the OpenFlow switches. This also includes the failure localization time. Let T_L be the failure localization time (as discussed in the previous subsection) and T_C the time taken by the controller to respond to the failure after failure localization. Specifically, the task for T_C includes the time to determine if it is a link or node failure and to modify or install flow tables to the corresponding switches. Hence, the failure recovery time T_R is depicted as

$$T_R=T_L+T_C. \quad (2)$$

Note that the total recovery time used in the calculation does not include the propagation delay, as the experiments were carried out in POX. The actual recovery time should include the propagation delay, which can be calculated based on the distance and added to the total recovery time. The propagation delay, however, would not be significant if the network topology is not very large or multiple controllers are used for the SDN.

Table 2 shows the results for the average failure recovery time for the failures between different pairs of switches. Fig. 5 illustrates the distribution of failure localization time and controller response time after a failure in the multicast tree is located till the time flow table entries in corresponding switches have been updated. The result is the average based on our experimental measurements. As shown in Fig. 5, after a failure in the multicast tree is located (T_L), the time required to determine if it is a link or node failure and to modify or install flow tables to the corresponding switches (T_C) dominates the failure recovery process. As a whole, however, the total recovery time falls below 0.5 ms on average (even if POX is not a carrier grade system) after a failure is detected and under an assumption that a backup path has been established. This indicates that the subtree-based protection algorithm is efficient in reacting to failures in a multicasting session in SDN.

Table 2 Average failure recovery time

Link	Average failure recovery time (ms)	Standard deviation (ms)
S1–S2	0.20	0.154
S1–S3	0.14	0.057
S2–S4	0.44	0.245
S3–S5	0.48	0.254
S3–S6	0.14	0.058
S4–S7	0.32	0.222
S4–S8	0.15	0.037
S6–S9	0.31	0.230

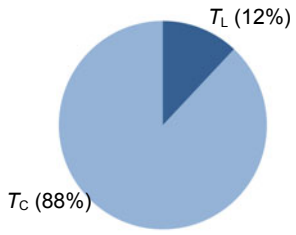

Fig. 5 Recovery time distribution

6 Conclusions and future work

Multicasting becomes more important in practical applications. This paper extended a subtree-based protection scheme (Wei *et al*., 2010) for a multicast tree proposed for the traditional network to the SDN domain. The proposed approach overcame some shortcomings of its predecessors, i.e., lower complexity in dealing with subtrees instead of the entire tree. However, the original subtree-based protection scheme assumed the existence of backup LSPs with each subtree. In practice, establishing backup LSPs using RSVP and FRR mechanisms requires a significant amount of configuration efforts, which makes the approach impractical in the traditional network.

On the other hand, in SDN LSPs can be set up much more efficiently using the controller rather than the RSVP signaling protocol which sets up an LSP in a hop-by-hop fashion. Hence, we adapted the subtree-based protection mechanism proposed for traditional networks and applied it to the SDN domain. Our proposed approach focuses mainly on protecting and

restoring failure at the network level of an ongoing multicasting session. Similar to Wei *et al*. (2010), our proposed scheme is more efficient in protecting subtrees as opposed to building an entire redundant backup tree as used in other approaches. In addition, the high complexity and control overhead of RSVP and FRR can be avoided. A number of experiments have been performed using Mininet. The results showed that the restoration time was short from the viewpoint of failure detection, even at the presence of the limitation of the existing topology discovery module in the POX controller (POXTD, 2014).

There are still open issues related to multicast. Some of the key areas in which this work can be extended are described here.

As described in Section 2, the existing topology discovery module (POXTD, 2014) in the POX controller has a main limitation on detecting the failure quickly. The reason is that the module is not event triggered; instead, it checks the connection between the OpenFlow switches periodically. We are modifying the existing discovery module event so that an event can be triggered in a much shorter time to reduce failure detection time. The Bidirectional Forwarding Detection (BFD) protocol (Katz and Ward, 2010) has been used for fast failure detection for traditional IP networks and SDN (van Adrichem *et al.*, 2014). Another direction is to integrate BFD for our proposed multicast subtree protection scheme.

The algorithm assumes a central controller. For large networks, multiple distributed controllers can be deployed. One direction is to modify the algorithm to a distributed algorithm for multiple controllers. On the other hand, the propagation delay will become smaller using distributed SDN controllers than that of a centralized controller.

Currently, the proposed technique is considered for a single multicast tree. If multiple multicast trees exist at the same time, the proposed protection scheme needs to be applied to each multicast tree separately, as different trees may have different topologies, source nodes, participating members, etc. In this case, it would be more efficient to have backup LSPs to different leaf nodes. Establishing backup LSPs using the SDN controller can greatly simplify the signaling overhead. Dynamic joining or leaving of members also warrants further investigation, even though subtree-based protection needs only a local maintenance scheme for a subtree as opposed to a global maintenance approach for the entire multicast tree.

We are also working on a couple of more directions. First, we are conducting experiments using the Floodlight (Floodlight, 2015) environment. OpenDaylight (https://www.opendaylight.org) is another option for experimental evaluation, as both environments have become popular and evolved quickly. Also, we consider to integrate our method with GroupFlow (Craig, 2014) for some multicast applications.

References

Akyildiz, I.F., Lee, A., Wang, P., *et al.*, 2014. A roadmap for traffic engineering in SDN-OpenFlow networks. *Comput. Netw.*, **71**:1-30. http://dx.doi.org/10.1016/j.comnet.2014.06.002

Bondan, L., Müller, L.F., Kist, M., 2012. Multiflow: multicast clean-slate with anticipated route calculation on OpenFlow programmable networks. *J. Appl. Comput. Res.*, **2**(2):68-74. http://dx.doi.org/10.4013/jacr.2012.22.02

Cain, B., Deering, S., Kouvelas, I., *et al.*, 2002. Internet Group Management Protocol, Version 3. RFC 3376. Internet Engineering Task Force, Fermont. Available from http://www.ietf.org/rfc/rfc3376.txt.

Congdon, P., 2002. Link Layer Discovery Protocol. RFC 2922. Available from https://tools.ietf.org/html/rfc2922.

Craig, A., 2014. GroupFlow. Available from https://github.com/alexcraig/GroupFlow.

Craig, A., Nandy, B., Lambadaris, I., *et al.*, 2015. Load balancing for multicast traffic in SDN using real-time link cost modification. IEEE Int. Conf. on Communications, p.5789-5795. http://dx.doi.org/10.1109/ICC.2015.7249245

Das, S., 2012. PAC.C: a Unified Control Architecture for Packet and Circuit Network Convergence. PhD Thesis, Stanford University, USA.

Farhady, H., Lee, H., Nakao, A., 2015. Software-defined networking: a survey. *Comput. Netw.*, **81**:79-95. http://dx.doi.org/10.1016/j.comnet.2015.02.014

Fei, A.G., Cui, J.H., Gerla, M., *et al.*, 2001. A "dual-tree" scheme for fault-tolerant multicast. IEEE Int. Conf. on Communications, p.690-694. http://dx.doi.org/10.1109/ICC.2001.937328

Fenner, B., Handley, M., Holrook, H., *et al.*, 2006. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 4601. Internet Engineering Task Force, Fremont. Available from https://tools.ietf.org/html/rfc4601.

Floodlight, 2015. Project Floodlight—Open Source Software for Building Software-Defined Networks. Available from http://www.projectfloodlight.org/floodlight/.

Huang, W.L., Guo, H.Y., 2009. A fault-tolerant strategy for multicasting in MPLS networks. Proc. Int. Conf. on Computer Engineering and Technology, p.432-435. http://dx.doi.org/10.1109/ICCET.2009.138

Katz, D., Ward, D., 2010. Bidirectional Forwarding Detection (BFD). RFC 5880. Internet Engineering Task Force, Fremont. Available from https://tools.ietf.org/html/rfc5880.

Kempf, J., Bellagamba, E., Kern, A., *et al.*, 2012. Scalable fault management for OpenFlow. Proc. IEEE Int. Conf. on Communications, p.6606-6610. http://dx.doi.org/10.1109/ICC.2012.6364688

Kitsuwan, N., McGettrick, S., Slyne, F., *et al.*, 2015. Independent transient plane design for protection in OpenFlow-based networks. *J. Opt. Commun. Netw.*, **7**(4):264-275. http://dx.doi.org/10.1364/JOCN.7.000264

Kotani, D., Suzuki, K., Shimonishi, H., 2012. A design and implementation of OpenFlow controller handling IP multicast with fast tree switching. IEEE/IPSJ 12th Int. Symp. on Applications and the Internet, p.60-67. http://dx.doi.org/10.1109/SAINT.2012.17

Kreutz, D., Ramos, F.M.V., Veríssimo, P.E., *et al.*, 2015. Software-defined networking: a comprehensive survey. *Proc. IEEE*, **103**(1):14-76. http://dx.doi.org/10.1109/JPROC.2014.2371999

Lantz, B., Heller, B., McKeown, N., 2010. A network in a laptop: rapid prototyping for software-defined networks. Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p.19. http://dx.doi.org/10.1145/1868447.1868466

Lee, M.W., Li, Y.S., Huang, X., *et al.*, 2014. Robust multipath multicast routing algorithms for videos in software-defined networks. Proc. IEEE 22nd Int Symp. of Quality of Service, p.218-227. http://dx.doi.org/10.1109/IWQoS.2014.6914322

Marcondes, C.A.C., Santos, T.P.C., Godoy, A.P., *et al.*, 2012. CastFlow: clean-slate multicast approach using in-advance path processing in programmable networks. IEEE Symp. on Computers and Communications, p.94-101. http://dx.doi.org/10.1109/ISCC.2012.6249274

McKewon, N., Anderson, T., Balakrishnan, H., 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.*, **38**(2):69-74. http://dx.doi.org/10.1145/1355734.1355746

Medard, M., Finn, S.G., Barry, R.A., *et al.*, 1999. Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs. *IEEE/ACM Trans. Netw.*, **7**(5):641-652. http://dx.doi.org/10.1109/90.803380

Moy, J., 1994. MOSPF: Analysis and Experience. RFC 1585. Internet Engineering Task Force, Fremont. Available from https://tools.ietf.org/html/rfc1585.

Nakagawa, Y., Hyoudou, K., Shimizu, T., 2012. A management method of IP multicast in overlay networks using OpenFlow. Proc. 1st Workshop on Hot Topics in Software Defined Networks, p.91-96. http://dx.doi.org/10.1145/2342441.2342460

Open Networking Foundation (ONF), 2012. Software-Defined Networking: the New Norm for Networks. ONF White Paper. Available from https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

Osborne, E., Simha, A., 2002. Traffic Engineering with MPLS. Cisco Press, Indianapolis, USA.

Pan, P., Swallow, G., Atlas, A., 2005. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090. Internet Engineering Task Force, Fremont. Available from https://tools.ietf.org/html/rfc4090.

Pfeiffenberger, T., Du, J.L., Arruda, P.B., *et al.*, 2015. Reliable and flexible communications for power systems: fault-tolerant multicast with SDN/OpenFlow. 7th IFIP Int. Conf. on New Technologies, Mobility, and Security, p.1-6. http://dx.doi.org/10.1109/NTMS.2015.7266517

Pointurier, Y., 2002. Link Failure Recovery for MPLS Networks with Multicasting. MS Thesis, University of Virginia, Charlottesville, USA.

POXST, 2016. POX Spanning Tree. Available from https://github.com/noxrepo/pox/blob/carp/pox/openflow/spanning_tree.py.

POXTD, 2014. POX Topology Discovery. Available from https://github.com/noxrepo/pox/blob/carp/pox/openflow/discovery.py.

Python, 2015. Python Time Complexity. Available from https://wiki.python.org/moin/TimeComplexity.

Renganathan Raja, V., Pandey, A., Lung, C.H., 2015. An OpenFlow-based approach to failure detection and protection for a multicasting tree. *LNCS*, **9071**:211-224. http://dx.doi.org/10.1007/978-3-319-22572-2_15

Rückert, J., Blendin, J., Hark, R., *et al.*, 2015. An Extended Study of DynSdm: Software-Defined Multicast Using Multi-trees. Technical Report, No. RS-TR-2015-01. Technische Universität Darmstadt, Darmstadt, Germany.

Saidi, M.Y., Cousin, B., Molnar, M., 2006. Improved dual-forest for multicast protection. 2nd Conf. on Next Generation Internet Design and Engineering, p.371-378. http://dx.doi.org/10.1109/NGI.2006.1678265

Sharafat, A.R., Das, S., Parulkar, G., *et al.*, 2011. MPLS-TE and MPLS VPNs with OpenFlow. Proc. ACM SIGCOMM, p.452-453. http://dx.doi.org/10.1145/2018436.2018516

Tiso, J., 2011. Designing Cisco Network Service Architectures (ARCH): Developing an Optimum Design for Layer 3 (CCDP). Cisco Press, Indianapolis, USA.

van Adrichem, N.L.M., van Asten, B.J., Kuipers, F., 2014. Fast recovery in software-defined networks. Proc. 3rd European Workshop Software Defined Networking, p.61-66. http://dx.doi.org/10.1109/EWSDN.2014.13

Wei, G.M., Lung, C.H., Srinivasan, A., 2010. Protecting a MPLS multicast session tree with bounded switchover time. Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems, p.236-243.

Xu, X.R., Myres, A.C., Zhang, H., *et al.*, 1997. Resilient multicast support for continuous-media applications, IEEE 7th Int. Workshop on Network and Operating System Support for Digital Audio and Video, p.183-194. http://dx.doi.org/10.1109/NOSDAV.1997.629385

Zhou, Y.L., Zhang, Y.S., 2009. An aggregated multicast fault tolerant approach based on sibling node backup in MPLS. Int. Conf. on Information Engineering and Computer Science, p.1-4. http://dx.doi.org/10.1109/ICIECS.2009.5364380