An Openflow-Based Approach to Failure Detection and Protection for a Multicasting Tree

Vignesh Renganathan Raja, Abhishek Pandey, Chung-Horng Lung

An Openflow-based Approach to Failure Detection and Protection for a Multicasting Tree

Vignesh Renganathan Raja, Abhishek Pandey, Chung-Horng Lung

Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada
{vigneshrenganathanra, abhishekpandey}@cmail.carleton.ca, chlung@sce.carleton.ca

Abstract – Software Defined Networking (SDN) has received considerable attention for both experimental and real networks. The programmability of the centralized control plane utilizes the global view of the network to provide better solutions for complex problems in SDN. This results in an increase in robustness and reliability of network functions running in SDN. This paper is motivated by recent advancement in SDN and increasing popularity of multicasting applications by proposing a technique to increase the resiliency of multicasting in SDN. 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. Multicasting applications, e.g., live video streaming and video conferencing, are popular and delay sensitive applications in the Internet. Failures in the ongoing multicast session can cause packet losses and delay and hence affect quality of service (QoS). In this paper, we present a technique to protect a multicasting tree constructed by Openflow switches in SDN. The proposed algorithm can detect link or node failures from the multicasting tree and then determines which part of the multicasting tree requires changes in the flow table to recover from the failure. 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.

Keywords: Software Defined Networks, Openflow, Multicasting Tree Protection, POX controller, Mininet

1 Introduction

Multicasting is a group communication technology, which uses the network infrastructure efficiently by transferring the same data only once from a sender to a group of receivers [5]. The routers involved in the multicasting session are capable of forming multicasting trees dynamically according to the members joining and leaving the multicasting group [11]. In the most common multicasting tree, the sender or source is always connected to the root of the tree and the receivers are connected to the leaf nodes of the tree. The structure of the tree may change dynamically when receivers join or leave the multicasting session [11]. Multicasting applications like real time video conferencing and live video streaming are getting more popular and the perfor-

mance of such applications relies critically on the resiliency of the multicasting tree architecture [13]. One key concern of real time multicast traffic is the delay and packet losses due to failures. To decrease the delay due to failures, it is central to protect the traffic involved from the link and node failures. There are various mechanisms proposed to protect multicasting sessions from link failures in the traditional networks [1-5]. In those methods, the failure notifications are sent via other nodes involved in the session to initiate the protection process. In practice, routing protocols and message flooding mechanism are used for topology synchronization. During the convergence period, packet losses and higher delay are inevitable and can be significant.

SDN separates the forwarding plane and the control plane from the physical networking element and runs the control plane in a logically centralized location [12]. This separation and centralization of control plane gives SDN controller the global view of the network, which can be efficiently utilized to monitor and control the network due to dynamic changes in the network. The introduction of the Openflow protocol enables the interaction between the controller and the forwarding elements or switches [10]. Using Openflow, the controller can install the flow entries to the switches according to the control applications running in the controller. Upon a failure, notifications can be sent by a switch that detects the failure directly to the SDN controller instead of flooding in the network to notify each node or as many nodes as possible. This means that the network can be automatically configured according to the way the control plane has been programmed, which reduces tremendous network complexity used in the traditional networks. Specifically, in a multicasting scenario, when a failure happens, the failure notification message can be sent to the controller rapidly and the multicast recovery process can be started immediately.

Further, in traditional networks, it is difficult for a network element to efficiently distinguish between a link and a node failure using the routing protocols [15]. To find out a node failure, routers have to identify if all links of a particular node are down, in which case the node is considered failed, which will result in high delay and packet losses and low QoS. With the combination of programmable control plane and the global view of the network topology in SDN, we can detect, protect, and restore *either link or node failures* in the network efficiently. In the traditional networks,

This paper is motivated to analyze how fast the control plane can react to protect a multicasting tree from either link or node failures in SDN. Analyzing and understanding failure restoration for real time multicasting is crucial, as multicasting becomes popular in practice. Hence, the main objective of this paper is to design the control plane architecture which responds by installing updated flow tables to the corresponding switches for link or node failures in a multicasting tree. The main contribution of this paper is to present a multicast failure protection and restoration scheme which distinguishes 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 we measure failure detection and controller response time by emulating SDN using Mininet [9].

The rest of this paper is organized as follows. Section II describes the related works. Section III describes the controller design and the proposed algorithm for mul-

ticasting tree protection and restoration. Section IV demonstrates the evaluation results for the failure localization and protection algorithm proposed in this paper. Finally, Section V presents the conclusions and future work.

2 Related Work

In the literature, multicasting tree protection and restoration schemes have been advocated for the optical layer or the network layer. This paper focuses on the network layer. This section describes the key component used in the current POX controller, which plays a significant role in failure detection. Following that, an existing approach for multicast protection using fast tree switching in SDN is briefly described.

In IP networks, multicast protection can be supported by proactive and reactive methods [2]. In general, the 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 after the event of failure. In the proactive method, the backup paths or trees are preconfigured before the failure happens. A few approaches of proactive tree protection for multicasting sessions are discussed in [1-3]. In Dual tree algorithm [1], the protection is performed by switching over the entire primary tree to a preconfigured backup tree whenever a link failure happens. This limits the protection algorithm for a single link failure in the multicasting tree. To accommodate protection for a link or a node failure, a dual forest algorithm is proposed in [3]. The node protection scheme is performed by pre-configuring backup paths covering each link involved in the primary tree. This approach is efficient only when the network topology is capable of providing alternate paths from each node to the leaf of the multicasting tree.

To tackle the problem of switching the whole multicast tree in case of multiple link failures and to make the protection scheme proactive, a subtree based protection scheme for multicast session using MPLS was presented in [5]. According to this approach, when a multicasting tree is built from the source to destinations, it has been divided into several subtrees. A subtree is a subset of the tree that represents a complete tree by itself [5]. There are advantages using the subtree based approach:

- It minimizes the failure detection time by avoiding the notification to be sent all the way to the root of the tree.
- It makes the protection scheme efficient by providing backup paths from root of each subtree to its leaf nodes.
- When a failure happens, the changes are made only to the corresponding subtree instead of the entire tree.

In summary, the failure detection and restoration time are critical, and the protection scheme must support protection for both the link and node failure for a single multicast session tree. However, the algorithm proposed in [5] does not address protection of multicasting session from a node failure.

2.1 SDN Topology Discovery

A topology discovery mechanism is used in the Openflow controller to make switches aware of their neighboring nodes. It is mainly used to detect link or node

failures in our approach. The *Topology Discovery* module included in the POX controller [6] is used for the discovery of any network topologies under its control. The *Topology Discovery* module uses the Link Layer Discovery Protocol (LLDP) [7] to detect the connections between the Openflow switches. The controller which executes the discovery module triggers the Openflow switches to send LLDP packets between each other. When a switch receives the LLDP packet from its neighbor, it sends an LLDP packet encapsulated in a Packet-In message to the controller. The Packet-In message has both the datapath id (DPID) and the port number of the sending and receiving switches. The controller then stores this information in the form of links. This stored information will act as a link between two Openflow switches. In this way, the controller learns the topology of the entire network under its domain. In the current POX design, the LLDP sending operation is triggered at a particular interval of time which is known as the *send cycle time* and is defined as follows:

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

By default the *link timeout* is set to 10. Based on the current POX controller design, when a failure happens, the *Topology Discovery* module in the controller will not be triggered until the *send cycle time* interval expires. In other words, failure detection by the controller may be delayed by an entire *send cycle time* in the worst case scenario.

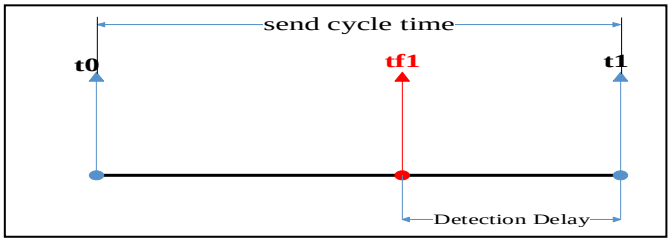

Fig. 1. Detection delay in existing Topology Discovery

As shown in Fig. 1, assume the *send cycle time* is the difference between t1 and t0. If the failure happens at time tf1, then the switches that detect the failure have to wait until t1 to send the notification to the controller. This may affect the failure detection time by several seconds, which will have a significant negative impact on packet losses, delay, and QoS. The exact value of the delay varies as the tf1 can be anywhere between the *send cycle time*. As a result, the failure detection can be much slower than the requirement, e.g., 50 ms, used for the existing carrier grade networks.

2.2 Existing failure protection mechanisms in SDN

In [4], multicast protection is performed by fast tree switching. The redundant tree is calculated as soon as the primary tree is calculated and flows for both primary and redundant tree 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 happens 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 available for some topologies.

3 Multicasting tree protection and restoration for Software Defined Networks

This section discusses the proposed multicasting tree protection and restoration scheme. Fig. 2 shows the high level view of SDN with Openflow POX controller for multicasting tree protection and restoration and the POX *Topology Discovery* modules. The existing *Topology Discovery* module in POX detects the failure and sends the information to the proposed *Failure Localization and Protection* module, and the *Failure Localization and Protection* module sends the flow tables to the Openflow switches according to the scheme.

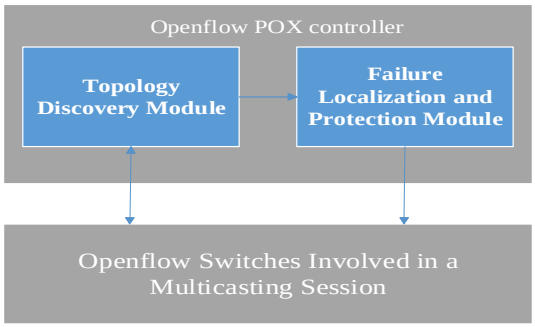

Fig. 2. Proposed Openflow Controller architecture for failure detection and protection

3.1 Assumptions

A few assumptions have been made for our proposed scheme:

- The network has a single Openflow controller. The reason is to validate the behavior and the performance of the protection and restoration algorithm. A single SDN controller is also common to many approaches in the literature.
- The failure detection time is from the time that the *Topology Discovery* module receives the notification. This is due to the limitations existing in the current POX *Topology Discovery* module as explained in Section 2.
- A backup path is available in a subtree from the root to the leaf nodes.
- The nodes of all the tree diagrams shown in this paper are Openflow switches and they are all directly connected to the controller. On this understanding we have avoided representing the connection between the switches and the controller in the figures.

3.2 Multicasting Tree Protection and Restoration Method for Openflow Controller

This section describes the four major operations of the multicasting tree protection and restoration algorithm for the Openflow controller. They are listed below:

- Subtree division
- Failure detection
- Failure localization

- Controller response

Subtree Division

This is the initial stage of the whole algorithm where 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 the controller receives from the POX *Topology Discovery* module. This is performed by using hash tables, where keys and values are used to identify parents 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 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”]

The DPID1 and DPID2 are the data path ids of the switches which share the link and port 1 is the port for 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. These data are stored in a two dimensional hash table with switch DPIDs as its keys and the links as its values. So the first step of the tree sorting process is to delete the duplicated link information for a single link between two switches [8]. 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 children 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 the link failure happens. After the tree is isolated, iteration through the tree is initiated. By iterating through the tree, the parents and children nodes are separated. The parent nodes are saved in a list and the children nodes are saved in a hash table with its parent DPID as its key. The parent and children information is then passed to the subtree division module.

Pseudo Code: Subtree Division

Objective: To divide the existing multicasting 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 multicasting tree.

Chlid_nodes: A hash table which has parent DPID as the key and Children DPID as its value.

Subtree: A hash table which has the subtree root DPID as the key and DPID of members of the subtree as its value.

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 into subtrees. Firstly 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: Root node of the whole tree. The first element of the *Parent_nodes* list.
Cur_node: Current node being iterated.
Input: *Parent_nodes, Children_nodes.*
Output: *Subtree, Subtree_key*

Algorithm:

```
Root_node = Parent_nodes[0]
Subtree_search (Root_node, Root_node)
Subtree_search(Cur_node,root):
  While (Child of Cur_node is not None)
    If (Child_nodes of Cur_node is > 1)
        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)
```

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 as the key and its members are the values. The root of a subtree is defined when a node has more than one children. The members of a subtree are added to the subtree until the search algorithm reaches 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 the link failure happens. The operation of the subtree algorithm is described in the pseudo code mentioned earlier.

Failure Detection

The dynamic changes in SDN are monitored by the control applications running in the controller. Link and node failures are among the most important changes in the network that have to be dealt with efficiently. The global view of the controller makes the 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 and we make use of existing functionalities in POX. Failure detection is conducted using the POX *Topology Discovery* module [7].

The *Topology Discovery* module raises a *LinkEvent* whenever there is a change in the status of the links associated with the Openflow switches. The links here are Layer 2 Ethernet ports of the Openflow switches. Whenever the 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.

Failure Localization.

The failure localization begins after the failure is detected. It is the process of identifying where exactly the failure happened 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 which share the link. The link information contains the DPID and the port number of the two switches which share the link. Using the DPIDs from the link information received, the algorithm searches the failed switches in the divided subtree to find which subtree the failure belongs to. The reason to do this is to identify the subtree for the changes. This search process avoids unnecessary changes to be made to the whole tree upon a failure.

The subtree search algorithm considers three scenarios to exactly find the location of the 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 both of the DPIDs are a root of a subtree.
3. When both of the DPIDs are the children of a subtree.

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

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

Consider the tree shown in Fig. 4. It has 3 subtrees as shown. If a failure happens between S3 and S5, the controller receives the link failure between them and will extract the DPIDs of the two switches. With the extracted DPIDs, the localization module searches if either one of the DPDIs is in the root list of the subtree. In this case S3 is the root and S5 is not. Then it checks if the node which is not the root of a subtree is a member of the other node, which is the root of a subtree. So it checks if S5 belongs to S3. If it is, the localization module forwards the information to the protection module with root node S3 as its starting point. If the failure is between S2 and S4, the localization module checks if S2 belongs to subtree S4. But it does not, so the localization module sends the root of the subtree where S2 belongs (in this case it is S1) to the protection module.

Scenario 2: Both of the nodes are a root of a subtree

In Fig. 3, if the failure happens between S1 and S3. When the localization module is satisfied that both of them are a root of a subtree, the localization module searches which node belongs to whose subtree. Thus, it searches whether S1 belongs to S3 subtree or S3 belongs to S1 subtree. Here, S3 belongs to S1 and hence the localization module initiates the protection module with S1 as the starting point.

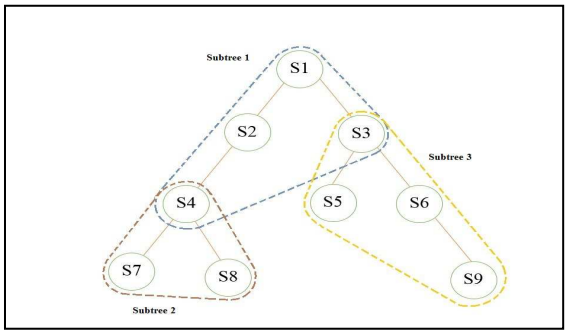

Fig. 3. A sub-divided tree of Openflow switches

Scenario 3: Both of the nodes are a child 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 just checks what subtree does one of the nodes belong to and initiates the protection module with the root of the subtree where the failure happened as the starting point for the protection module. In Fig. 3, if the failure happens between S6 and S9, the search process will result in sending S3, the root of the subtree, as the starting point for the protection module.

Pseudo-Code: Failure Localization

Objective: To identify at which part of the subtree the failure has happened 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 port number of two switches involved in the failure.

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

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

Input: *Failed_link(s)*

Output: Invoking Protection function with the exact switch where the flow table has to be installed.

Algorithm:

```
For all switches in Failed_switches
    If (S1 or S2 is in Subtree_root)
        If (S1 is in Subtree_root but not S2)
            Execute Failure_protection (S1, S1, S2)
        Else if (S2 is in Subtree_root but not S1)
            Execute Failure_protection(S2, S1, S2)
        Else
            If (S1 is the Subtree root of S2)
                Execute the Failure_protection (S1, S1, S2)
            Else if (S2 is the Subtree root of S1)
                Execute the Failure_protection (S2. S1, S2)
    Else
        Find the Subtree_root of S1 and S2 and execute Failure_protection (Subtree_root, S1, S2).
```

All the steps to localize the failure explained earlier require tree isolation. The reason is that when a failure happens, the link information gets 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.

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 one is to send flow table modifications to the switches responsible for

protecting the switches from failure. For the traditional networks, network nodes cannot efficiently distinguish link or node (neighbor) failures with the routing protocols. However, with SDN, the controller can distinguish them as switches can send notification directly connected to the controller. The difference is described below:

Determination of Link and Node Failure

Whenever the failure is detected by the *LinkEvent* handler, the ports associated with each switches involved in the failure is added to a *failed_ports*. The *failed_ports* is later used to count whether the number of ports belongs to the switches involved 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 it is not we consider it as a general link failure.

Node Failure

This function performs two quick searches. One is to identify if the node which undergone failure 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 where the failed node belongs to. 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. 3 if the node S4 has a node failure, then the controller will respond to S1. But if S6 failed, the controller will still respond to node S3.

Link Failure

Link failure is initiated if the condition for the node failure is false. For link failure, the controller just initiates the flow installation function with respect to the node DPID it gets from the flow localization module. For example, in Fig. 3, when a link failure happens between S3 and S5, then the Link Failure function will make changes to S3, as it is the root node of the subtree where the failure happened.

Flow Modification

The flow modification function installs new flows to switches based on the DPIDs it receives. 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: Failure Protection

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. Triggers flow installation.
- *Link_failure()*: A method which determines the switch to which the flow table should be modified in case of link failure. Triggers flow installation.
- *Failure_protection()*: A method which determines if it 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 *root* node as an argument to install flows to recover from the failure.
- *failed_ports*: Tracks the number of ports failed in each switch.
- *fs1, fs2*: DPIDs of the Switches involved in the failure. This argument is passed by the *Failure_Localization* module.
- *failed_root*: Root node identified by the *Failure_localization* module to which flows are to be installed.
- *max*: Maximum number of ports in a switch

Algorithm:

```
Failure_protection (failed_root, fs1, fs2):
        If (failed_ports of fs1 or fs2 == max)
          If (failed_ports of fs1 == max)
            Node_failure (failed_root, fs1)
          Else
            Node_failure (failed_root, fs2)
        Else
            Link_failure(failed_root)

Node_failure (failed_root, fs):
        If (fs not in Subtree [root])
            Flow_installation (failed_root)
        Else
            For (all root in Subtree)
              If (failed_root in Subtree [root])
                  Flow_installation (root)

Link_failure (failed_root):
      Flow_installation (failed_root)

Flow_installation (root):
        target = root
        message = Openflow_mod()
        message.command = Add flow table
        connection = Openflow.getconnection(target)
        connection.send(message)
```

4 Experimental Results

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

4.1 Experiment Setup

The focus of the experiments is failure localization and protection algorithms which are implemented in the Openflow POX controller. Failure detection is realized using the *Topology Discovery* module in POX which can be replaced with other standards. The real instances of Openflow switches representing a multicasting tree session and the POX controller are emulated in Mininet [9]. 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 on top of the Windows 7 64-bit operating system and Intel i7-3770 CPU with 3.40GHz processing power and has 16 GB of RAM installed.

Fig. 3 shows the network topology, which has 9 OpenflowVSwitches connected to the POX controller. Both switches and the controller are running in the same virtual machine. The controller is running the *Topology Discovery* module, the 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 by using Mininet's Command Line Interface. We failed each link between each pair of switches and repeated the experiment 10 times per pair. We calculated the average failure localization time and failure recovery time. The results of the experiments are discussed in detail in the following section.

4.2 Experimental Results

Failure Localization Time

Failure Localization time is the time interval from which the multicast localization and protection module gets the link failure information from the *Topology Discovery* module to the time it finds the exact failure location. This shows the time taken by the controller to search through the hash-tables where the subtrees are stored.

Table 1 shows the average failure localization time for failure between different pairs of switches. As depicted in Table I, the average failure localization time is small, mostly less than 40 μs. The results for the failure between nodes S3-S5 and S2-S4 is higher than others because of the Scenario 1 discussed earlier. The reason is that the search process is performed on both the subtree root list and the hash-table to make sure which node is the root and which is a member of the root. The failure between the non-root nodes takes more time when compared with others due to the increase in the number of searches to find the location of the failure in a subtree.

Table 1. Average Failure Localization Time

Failure between nodes	Average Failure Localization Time (μs)	Standard Deviation
S1-S3	33.6	0.051
S1-S2	40.5	0.042
S3-S6	15.9	0.004
S3-S5	44.2	0.013
S2-S4	65.1	0.048
S6-S9	18.5	0.042
S4-S7	23.8	0.022
S4-S8	26.6	0.024

Failure Recovery Time

Failure Recovery time is the total time taken from which the controller receives the 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 be the time taken by the controller to respond to the failure after the failure localization. Then the Failure Recovery Time T_R is,

$$T_R = T_L + T_C \tag{2}$$

Table 2. Average Failure Recovery Time

Failure Between Nodes	Average Failure Recovery Time (ms)	Standard Deviation
S1-S3	0.14	0.057
S1-S2	0.20	0.154
S3-S6	0.14	0.058
S3-S5	0.48	0.254
S2-S4	0.44	0.245
S6-S9	0.31	0.230
S4-S7	0.32	0.222
S4-S8	0.15	0.037

Table 2 shows the results for the average failure recovery time for the failures between different pairs of switches. As explained earlier, the response time for the failure between S3-S5 and S2-S4 is higher because of the delay in failure localization. But as a whole the total response time falls below 0.5 milliseconds. This indicates that the algorithm is efficient in reacting to failures in a multicasting session in SDN.

5 Conclusions and Future Work

Multicasting becomes more important in practical applications. This paper presented an approach to multicast tree protection and restoration for SDN. The proposed approach was designed based on subtree protection and restoration and mainly fo-

cused on protecting and restoring failure at the network level of the ongoing multicasting session. The main benefit of the proposed scheme is that it is more efficient to identify subtrees as opposed to build an entire redundant backup tree as used in other approaches. A number of experiments have been performed using the Mininet. The results showed that the restoration time was short from the point of failure detection.

Some of the key areas in which this work can be extended are described as follows:

The existing standard *Topology Discovery* module [7] in the POX controller has several limitations 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 the event can be triggered in a much shorter time to reduce failure detection time. In traditional networks, Bidirectional Forwarding Detection (BFD) protocol [14] has been used for fast failure detection. One direction is to integrate BFD into POX or SDN Openflow.

The algorithm assumes a central controller. For large networks, multiple distributed controllers can be deployed. One direction is to covert the algorithm into a distributed algorithm for multiple controllers.

References

[1] A. Fei, J. Cui, M. Gerla and D. Cavendish, "A "Dual-Tree" Scheme for Fault-Tolerant Multicast" *Proc. of ICC*, pp. 690-694, Jun 2001.

[2] Y. Zhou and Y. Zhang, "An Aggregated Multicast Fault Tolerant Approach based on Sibling Node Backup in MPLS", *Proc. of ICIECS*, pp. 1-4, Dec. 2009.

[3] M. Y. Saidi , B. Cousin and M. Molnar, "Improved Dual-Forest for Multicast Protection", *Proc. of NGI*, 2006.

[4] D. Kotani, K. Suzuki, H. Shimonishi, "A Design and Implementation of Openflow Controller Handling IP Muticast with Fast Tree Switching," *Proc. of SAINT*, pp. 60-67, 2012.

[5] G. Wei, C.-H. Lung, A. Srinivasan, "Protecting a MPLS Multicast Session Tree with Bounded Switchover Time," *Proc. of SPECTS*, pp. 236-243, July 2010.

[6] P. Congdon, *Link Layer Discovery Protocol*, RFC 2922, July 2002.

[7] POX Topology Discovery, available online: "https://github.com/noxrepo/pox/blob/carp/pox/openflow/discovery.py", last accessed in June 2014.

[8] POX Spanning Tree, available online: https://github.com/noxrepo/pox/blob/carp/pox/openflow/spanning_tree.py, last accessed in June 2014.

[9] B. Lantz, B. Heller, N. McKeown. "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks". *Proc. of Workshop on Hot Topics in Networks*, pp. 20-21, 2010.

[10] N. McKewon, T. Anderson, G. Peterson, J. Rexford, S. Shenker, J. Tuner, "OpenFlow: Enabling Innovation in Campus Networks", *SIGCOMM Rev*. 38(2), 69-74, 2008.

[11] B. Cain et al., *Internet Group Management Protocol*, Version 3. RFC 3376, Oct.2002.

[12] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks", White Paper, April 13, 2012.

[13] X. R. Xu, A. C. Myres, H. Zhang and R. Yavatkar, "Resilient Multicast Support for Continuous-Media Applications", *Proc. of NOSSDAV*, May 1997.

[14] D. Katz, D. Ward, *Bidirectional Forwarding Detection*, IETF RFCs 5880, June 2010.

[15] E. Osborne and A. Simha, *Traffic Engineering with MPLS*, Cisco Press, 2002.