Electronic Preprint for Journal of Information Processing Vol.24 No.2

Regular Paper

A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks

Daisuke Kotani[1,a)] Kazuya Suzuki[2,3] Hideyuki Shimonishi[2]

Received: May 6, 2015, Accepted: December 7, 2015

Abstract: We propose a multicast tree management method in an OpenFlow controller that handles both fast failure recovery and dynamic multicast group membership changes. Multicast communication is an efficient tool to distribute data to many hosts in various services such as live video streaming. To use multicast in such services, multicast communication must be reliable, which means multicast communication should be restored quickly after failures, and multicast tree management mechanism should support frequent group membership changes. A conventional approach, Point to Multipoint (P2MP) MPLS, only supports fast failure recovery for reliability, and is not very effective in terms of group membership changes. A new approach using OpenFlow supports dynamic group membership changes, but does not consider fast failure recovery in physical switches whose flow entry modification is slow. Our proposed method is to control multicast trees centrally, and it uses a precomputation and pre-installation approach for tree management. A controller calculates and keeps multiple trees that cover all switches where receivers are potentially connected and that have less common nodes and edges, and installs their sub-trees covering switches where receivers are actually connected. The controller calculates the difference per tree between sub-trees before and after membership changes, and reflects them into the network. At the time of failure, the controller checks and finds a pre-installed tree that is unaffected by the failure, and installs a new rule only to a root switch to send packets through the pre-installed alternate tree. Our experiments using switches and our prototype controller show that our proposed method can restore packet delivery quickly after a failure, as well as that our proposed method can handle tree modifications faster than a method of recalculating or reinstalling a tree every time that group memberships are changed.

Keywords: centralized management, multicast tree management, Fast Reroute, OpenFlow

1. Introduction

Multicast is an important communication tool to distribute data to a number of hosts at the same time with lower bandwidth. Examples of multicast applications are live video streaming, large data replication, and news headline distribution to many hosts.

As various devices are connected to IP networks, making multicast reliable becomes important in private networks such as campus and enterprise networks, as well as in carrier networks where conventional approaches is targeted. For example, multicast is an efficient way to distribute data to many monitors installed inside buildings and public areas, and to send video to many receivers for teleconferences and remote lectures.

To use multicast in these scenarios, multicast communication must be highly reliable and data loss should be minimized while delivering data in the network. In the network layer, less packet-loss packet delivery is important even at the time of failures, because packet loss imposes much overhead on a sender and many receivers for data recovery, or degrades quality of services provided by applications. For example in video streaming, when frames are lost due to packet loss, video and audio are stopped or disturbed for a longer time than the duration when frames are lost, and the distortion gets worse when frames are lost bursty [16] such as by failures. Such distortion causes obstruction in teleconferences and remote lectures.

Private networks should handle many multicast group membership changes because receivers are frequently added or removed when devices are such as monitors are turned on and off every day, and a set of receivers is not fixed until a teleconference or a remote lecture starts. In such networks, it is hard to allocate network resources to deliver multicast packets in advance, such as packet forwarding rules, and networks should quickly modify multicast trees according to multicast group membership changes. Point to Multipoint (P2MP) MPLS, which is used for reliable multicast packet delivery in carrier networks, is not designed for use in such frequent group membership changes [32], and has much overhead on routers by the tree setup procedures in RSVP-TE [2]. A tree manager in P2MP MPLS is proposed to use more flexible tree management policies by computing multicast trees centrally [4]. Since the tree manager recalculates and replaces the trees every time that group membership is changed, the manager cannot handle many group membership changes be-

[1] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[2] Cloud System Research Laboratories, NEC Corporation, Kawasaki, Kanagawa 211–8666, Japan
[3] Graduate School of Information Systems, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[a)] kotani@net.ist.i.kyoto-u.ac.jp

© 2016 Information Processing Society of Japan

cause the tree calculation takes much time and the tree setup by RSVP-TE imposes much overhead on routers.

Local repair based mechanisms such as MPLS Fast Reroute [23] and OpenFlow Fast Failover Group [9] can reduce packet loss at the time of a failure by rerouting packets at a switch where a failure is detected, but such mechanisms sometimes force a network to use non-optimal trees as backup. The private networks often have a three layered tree topology (core - aggregation - edge), and redundant routers and switches are deployed with redundant links. For example, edge switches provide links to each room in a floor, and have links to two or more aggregation switches that connect edge switches in a building. Each aggregation switch is connected to two or more core switches that connect aggregation switches on campus. Thus, several optimal multicast trees that can cover different core and aggregation switches are often available, and the use of such trees as backup trees is preferred in terms of bandwidth utilization, latency, etc.

OpenFlow [18] based multicast control schemes have been proposed [17], [33], [34] to support both frequent group membership changes and fast failure recovery. A basic concept of these works is to calculate a path or a tree in advance or using powerful computing power in OpenFlow controllers, and to modify flow entries in OpenFlow switches. In terms of fast failure recovery, the controller should take account of the time to modify flow entries in switches in addition to tree computations, because flow entry modification performance is slow, especially in physical switches. Huang et al. [12] reported that a flow entry setup took around 25 msec. When one failure affects several multicast groups, it may require several hundred milliseconds until all multicast trees are recovered due to slow flow entry modification, and large packet loss may be observed at receivers. This results in dropping tens of frames for video, for example. The previous work for multicast control schemes using OpenFlow does not consider this performance problem.

In this paper, we propose a multicast tree management method in an OpenFlow controller (a multicast controller hereafter), which provides both fast failure recovery for reliable multicast packet delivery and quick group membership changes. A multicast controller runs outside switches like a tree manager in P2MP MPLS, and directly controls flow entries regarding multicast packet forwarding installed in switches. The multicast controller is responsible for group membership collection from switches, tree management including computation, setup and modification, and failure recovery.

To shorten the time to process tree modifications by group membership changes, the multicast controller adopts a preplanned approach. When a multicast group appears in the network, which means that the controller obtains a sender's IP address and a root switch where a sender is connected and that one or more receivers join in the group, the controller computes and stores two or more trees. The trees cover all switches to which a receiver is potentially connected, and the trees share less common edges and nodes.

When the controller handles a group membership change, the controller lists switches whose flow entries must be updated by following each stored tree in the group from the leaf switch where the group membership change is occured, instead of recalculating trees. Then, the controller modifies flow entries in the switches listed. The controller stores per-tree states of nodes and edges to show whether flow entries to forward packets through corresponding switches and links are installed. The controller changes these states while following the trees if necessary, stops following the tree when the controller finds the node whose state does not have to be changed, and updates flow entries in switches that corresponds to nodes the controller has visited while following the trees.

To shorten the time for failure recovery, the controller installs multiple trees per group in switches at the same time so that the controller can also set backup trees in advance. To avoid duplicate packet delivery, the controller assigns a unique ID within a group to each tree and embeds it into packets, like MPLS labels. The tree ID for packet delivery is embedded at the root switch of the tree. The switches other than the root switch have one flow entry per tree, and forward packets according to the tree ID. When a failure is detected, in each group, the controller checks whether the failed switch or link is included in the tree used for packet delivery. If yes, the controller finds a tree unaffected by the failure from trees that the controller has already installed in switches, and changes the tree ID embedded at the root switch to the new one. In this way, the controller needs to modify only one flow entry in one switch per group, instead of modifying flow entries in many switches to replace trees.

We have implemented our prototype system using C and Trema [27]. We use two trees per group, and the trees share fewer edges. We embed a tree ID into a source MAC address. Our evaluation shows that our proposed method can handle the addition or the removal of receivers in a short time, and restore packet delivery faster than without our proposed method at the time of link failures even if we consider the use of a physical switch.

This paper is organized as follows. We describe related work in Section 2. We explain our proposed method in Section 3 and our prototype controller in Section 4. In Section 5, we present the results of evaluation experiments. We discuss the evaluation results and our proposed method in Section 6 and give our conclusion in Section 7.

2. Related Work

In current IP multicast, a host multicasts or broadcasts IGMP [8] messages to join in or leave from multicast groups, and IP routers find multicast receivers by monitoring IGMP messages. Multicast routing is controlled by PIM-SM [6], PIM-DM [1], DVMRP [28] or MOSPF [21]. PIM-SM, DVMRP, and PIM-DM cannot provide fast failure recovery. PIM-SM and DVMRP construct multicast trees using unicast routes, and they cannot reconstruct multicast trees until unicast routes are stabilized when a failure occurs. PIM-DM periodically update multicast trees by flooding multicast packets and pruning the trees, and the trees are not reconstructed until periodic flooding occurs. MOSPF distributes group membership information to all multicast routers to compute multicast trees. This behavior is not scalable to group membership changes.

© 2016 Information Processing Society of Japan

Point to Multipoint (P2MP) MPLS [32] provides reliable multicast packet delivery with fast reroute and bandwidth guarantee using MPLS mechanisms, and P2MP MPLS is designed for carrier networks. When a failure occurs, the fast reroute function in P2MP MPLS reroutes the traffic at a router where a failure is detected, which is called local repair, to reduce packet loss. Li et al. [14] proposed methods to compute efficient P2MP backup trees in MPLS networks. Cui et al. [4] proposed to use P2MP MPLS to implement Aggregated Multicast [7] in IP networks, and introduced a tree manager to manage multicast trees such as mapping trees to groups.

P2MP MPLS based solutions are unsuitable when multicast group membership is frequently changed. P2MP MPLS assumes that group membership is rarely changed [32]. To modify multicast trees in P2MP MPLS, RSVP-TE [2], which is used to setup multicast trees in P2MP MPLS, requires the root router of the tree to send a path setup messages to the leaf routers, and routers other than the root router must send a response to their upstream routers. Thus, all routers should process many messages as the multicast group membership is frequently changed. Multicast extension to LDP (mLDP) [30] may also be used for P2MP MPLS, but mLDP does not provide fast reroute functionality.

In terms of tree computation algorithms, a shortest path tree is used in many unicast and multicast routing, and usually computed by Dijkstra's [5] algorithm. Médard et al. [19] showed an algorithm to compute redundant trees in arbitrary vertex-redundant or edge redundant graphs. Xue et al. [31] extended Médard's algorithms to compute redundant trees considering costs. Mochizuki et al. [20] showed an algorithm to compute a tree minimizing the number of links included in the tree. Our proposed method can suppress the execution of tree computation algorithms and the number of flow entries the multicast controller should modify in given trees, and the multicast controller can use any tree computation algorithms including above.

Some tree pre-computation or caching algorithms are also proposed to compute multicast trees faster. Siew et al. [26] proposed a tree computation algorithm by connecting cached trees. Siachalou et al. [25] showed an algorithm to compute a tree constrained by bandwidth. These proposals only consider the computation algorithms, but the execution time for tree modification, including flow entry modification, is also an important metric in our work.

There is a great deal of research on reducing data loss in an application layer. RFC3048 [29] recommends some mechanisms like NACK based packet loss detection and recovery, and FEC coding to recover data lost by a few packets. Hasegawa et al. [11] proposed a system to multicast HD quality videos with non-stop service availability in carrier networks, which uses buffers in backup servers near receivers and backup trees controlled by separate processes from main trees.

There are several works to manage multicast trees using OpenFlow. OFM [33] proposed a mechanism to manage multicast in OpenFlow networks using multiple controllers that control different parts of a network. CastFlow [17] proposed to precompute multicast trees from possible sources to receivers, and to store a list of links included in the trees for fast processing of group membership changes. Zou et al. [34] used OpenFlow to implement authentication of receivers when a receiver joins in a group. These schemes provide how to manage multicast trees in the controller, but they do not consider how to setup multicast trees to switches in a short time, which is an important aspect for failure recovery in OpenFlow networks because of slow flow entry setup performance in physical switches. Huang et al. [12] reported that one physical switch took 25 msec to modify a flow entry. If a single failure affects multicast trees of several groups, it may take hundreds of milliseconds until the trees used by all groups are recovered, which is longer than conventional approaches like MPLS fast reroute.

Li et al. [15] proposed to manage multicast groups separately in data centers, but their work can be applied only to multi-rooted tree networks. Capone et al. [3] proposed an algorithm to reroute multicast traffic with zero packet loss by their own extension called OpenState. Although their mechanism cannot work without modification of switches to OpenState capable ones, their algorithm can be used to compute multicast trees in our proposed mechanism. Gyllstrom et al. [10] proposed a similar multicast tree recovery scheme with our proposed mechanism, but their work cannot directly applied to private networks where group membership is frequently changed because they implicitly assumed that a set of receivers is static. Our previous work [13] measured the packet loss and the tree switching time using our prototype multicast controller, but its evaluation was done with a small network, and did not show how our proposed mechanism improved the processing time in the controller.

In terms of failure recovery in OpenFlow networks, OpenFlow 1.1 [9] and later define a Fast Failover Group feature to reroute traffic at a switch where a link failure is detected (local repair). This feature allows controllers to specify actions applied to packets when a specific port or group is down, such as outputting to another port. The fast failover group feature is used for fast rerouting of paths [24], but local repair using the fast failover group feature is not suitable for failure recovery of multicast trees. Controllers should prepare for backup multicast trees per switch, and switches should have flow entries for each backup trees. This dramatically increases the number of flow entries in switches. In addition, backup multicast trees created by local repair are sometimes inefficient in terms of bandwidth utilization or latency in networks. For example, if one link between an aggregation switch and an edge switch in a three layered tree network is down, and multicast traffic gets through a core switch, it would be better to reroute traffic to another core switch near the root of the tree, otherwise multicast traffic may go through multiple core switches because of a reroute at the aggregation switch where the failed link is connected.

3. Tree Management Method

In this section, we explain the relationship between a multicast controller and switches, and how to manage multicast trees in a multicast controller.

Figure 1 shows a relationship between switches and our multicast controller. Each host is attached to one of the switches, and the controller manages flow entries regarding multicast in

© 2016 Information Processing Society of Japan

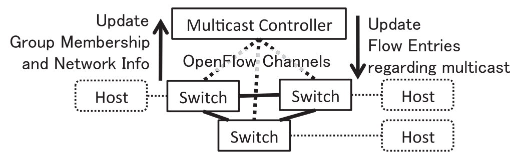

Fig. 1 Relationship between switches and multicast controller.

switches through OpenFlow channels.

To calculate and manage multicast trees, the controller must collect network topology, and find where senders and receivers are connected. The controller has a database to store network topology as a graph, a state of multicast trees, and multicast group membership including senders and receivers. Switches send packets that are related to group membership changes to the controller, such as IGMP messages that receivers send, and packets sent to unknown multicast groups. The switches also send other events to the controller, like port down or up. When the controller receives such events, the controller updates its database, and modifies flow entries in switches if needed.

When a group membership is frequently changed, the controller should be overloaded due to calculation of new trees that reflect changes, and some switches should also be busy updating their flow entries. To reduce the loads, we propose that the controller computes and stores trees covering all switches that receivers may be connected to, and uses them to extract switches whose flow entries should be updated to reflect changes. We describe the details in Section 3.1.

When a failure is notified to the controller, the controller should recalculate the trees that do not include failed links or switches, remove the existing trees from switches to avoid duplicate delivery of multicast packets, and install the new trees in switches. There are several time-consuming tasks such as tree computation, installing, and uninstalling trees in switches. These tasks make the packet loss duration longer. We propose that the controller calculates and installs redundant trees when a group membership is changed, and the controller can switch the tree to deliver packets at the time of failure. The details are explained in Section 3.2.

In the following, we use the terms "switch" and "link" to point out physical entities that construct a network, and "node" and "edge" as internal data structure of switch and link in the controller. Each node and edge corresponds to each switch and link. "Root node" or "root switch" represents a node or a switch where a sender is connected, and "leaf node" or "leaf switch" is a node or a switch where a receiver is connected.

3.1 Tree Update when Group Membership Changes

Processing group membership changes quickly is a key to handle many group membership changes in the controller. This process includes three steps: (1) updating the group membership database in the controller, including switches and ports where senders and receivers are connected, (2) computing how multicast trees must be updated, and (3) updating flow entries regarding multicast in switches. We explain the group membership database in Section 3.3. To handle group membership changes quickly, we need to reduce time-consuming tasks, such as tree computations in the controller, and flow entry modifications in switches.

Our approach is to calculate multiple trees per group. Each tree covers all nodes where receivers may be connected, and is stored in the tree database in the controller. The trees are calculated when both a sender and one or more receivers appear. The trees are deleted when neither sender nor receiver exists in a group. The controller maintains and modifies tree data as below.

In order for the controller to extract switches whose flow entries should be modified, the controller should know which switches have flow entries used for forwarding multicast packets. A simple way to do this is to query switches to send flow entries regarding multicast forwarding to the controller, but this brings much overhead such as delay until the controller receives all responses. We store such state in the controller by defining a subtree of each tree. The subtree covers a sender and all receivers, and is installed in switches to deliver packets. Each node and edge in the tree has a flag that indicates whether the node or edge is included in the subtree. The flag is on when the corresponding switch or link is in the subtree (the active state), and the flag is off when the corresponding one is not in the subtree (the inactive state).

When a new receiver is added to a group, firstly the controller retrieves the trees used by the group from the database, or calculates the trees and stores them to the database. Then, for each tree, the controller executes Algorithm 1 to include the new leaf node, where the new receiver is connected, in the subtree, and to list nodes whose corresponding switches must update their update flow entries (*update_nodes* in Algorithm 1). The controller follows the tree to the root node from the new leaf node until the controller visits the active node. While following the tree, the controller changes a state of visited nodes and edges to active, and adds the visited nodes to the list of nodes whose corresponding switches must update their flow entries. When adding a node to *update_nodes*, edges to the parent node and to the child nodes that are active are also saved in the list, and this data is used for constructing new flow entries. Then, the controller fetches edges that are not directly related to the tree, such as edges to a sender and to receivers, from the group membership database in Section 3.3, and updates flow entries in the switches whose corresponding nodes are in *update_nodes*. The flow entries are updated in the order of *update_nodes* so that packets are transmitted after all switches are ready to deliver. For the trees other than the tree used for packet delivery, the controller does not install any flow entry in the root switch, so that the flow entry does not override the flow entry installed by the tree used for packet delivery.

The process of removing a receiver from a group is similar to the process of adding a receiver. Firstly, the controller retrieves the trees used by the group from the database. Then, for each tree, the controller executes Algorithm 2 to list nodes whose corresponding switches must update their flow entries to remove the receiver. The controller follows the tree from the leaf node where the receiver is connected until the controller finds the node that has other receivers or child nodes that are active. While following the tree, the controller sets a state of visited nodes and edges to inactive except the node that the controller stops following the tree, which must be active. Finally, the controller updates or re-

© 2016 Information Processing Society of Japan

Algorithm 1 Algorithm to list nodes when adding a new receiver

```
update_nodes = []
node ← a leaf node where a new receiver is attached
append node to update_nodes
while a state of node is inactive do
    set a state of node to active
    if node's parent node exists then
        set a state of the edge between node and node's parent node to active
        append node's parent node to update_nodes
        node ← node's parent node
    else
        break
    end if
end while
```

Algorithm 2 Algorithm to list nodes when removing a receiver

```
update_nodes = []
node ← a leaf node where a receiver is left
append node to update_nodes
while no child node of node is active and node has no receiver do
    set a state of node to inactive
    if node's parent node exists then
        set a state of the edge between node and node's parent node to inac-
        tive
        append node's parent node to update_nodes
        node ← node's parent node
    else
        break
    end if
end while
```

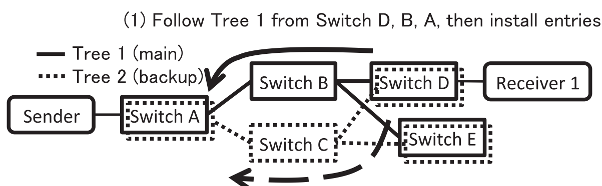

Fig. 2 An example of processing multiple trees.

moves flow entries in the switches whose corresponding nodes are in *update_nodes*. The flow entries are updated or removed in the reverse order of *update_nodes* to stop delivering packets first.

Figure 2 gives an example of adding a receiver (Receiver 1) to the trees. There are two trees, Tree 1 shown by solid lines, and Tree 2 by dotted lines.

If Receiver 1 is the first receiver in the group, the controller calculates one tree (Tree 1) for packet delivery, and another one (Tree 2) for backup. Then, the controller follows nodes of Switch D, B, and A on Tree 1 in order, changes the state of these nodes and edges between them to active, and installs flow entries in these switches. Next, the controller follows nodes of Switch D, C, and A on Tree 2 in order, changes the state of these nodes and edges between them to active, and installs flow entries in the switches except Switch A.

When another receiver joins in the group, for example a new receiver is connected to Switch E, the controller follows the trees and installs flow entries like the case of Receiver 1. The controller stops following the tree at the node of Switch B or C because these nodes have already been active. In this case, the controller modifies the flow entries in Switch B and C, and installs new flow entries in Switch E.

When a receiver at Switch E leaves from the group, the controller follows the trees from the node of Switch E to the nodes of Switch B or C because the nodes of Switch B and E have active edges to the node of Switch D. Then, the controller removes flow entries in Switch E, changes the state of the node of Switch E and the edge between it and its parents (the nodes of Switch B and C) to inactive, and modifies the flow entries in Switch B and C. When Receiver 1 also leaves, the controller follows the trees until the node of Switch A, removes the flow entries from all switches, and deletes the trees stored in the controller's database because no receiver exists in the group.

In our proposed method, the controller needs to calculate trees only when a new group appears, and reuses them when the controller handles group membership changes. In addition, the controller does not need to communicate with switches to retrieve the state of the trees because the state of the trees is already stored in the controller. Therefore, the controller can handle modifications to multiple trees quickly, and a few switches communicate with the controller to update their flow entries.

3.2 Failure Recovery

To make multicast reliable, the multicast trees must be recovered quickly when a failure occurs in networks. MPLS Fast Reroute (FRR) mechanism works well by setting up backup paths or trees in advance and by rerouting to one of the backup paths at the time of failures. We also use the preplanned approach like MPLS FRR, but MPLS FRR limits the flexibility of backup trees that the controller computes because MPLS FRR requires the parent router of the failed router or link to reroute the traffic.

As explained above, in our proposed method, the controller computes multiple trees per group, and installs the subtrees that need to deliver packets to receivers at the same time in the same way of the tree for packet delivery except the root switch.

The network should avoid duplicate multicast packet delivery and packet loop both in normal state and during failure recovery. The controller assigns a unique ID within a group to each tree, and embeds it into a packet header like MPLS. The root switch embeds the tree ID used for packet delivery into packets by rewriting a part of packet headers. Other switches forward the packets based on the tree ID, the source IP address and the multicast address in header fields. The switches rewrite the header fields in the packets to the appropriate ones when they send the packets to receivers.

When a failure is detected at a switch or other devices, the switch or other devices notify the failure to the controller. For each group, the controller checks whether the tree currently used for packet delivery has become disconnected by the failure. Then, the controller executes the tree recovery procedure for each group whose tree currently used for packet delivery has been disconnected.

The tree recovery procedure is as follows. The controller selects one of backup trees unaffected by the failure, and modifies the flow entry in the root switch to embed an ID of the selected

© 2016 Information Processing Society of Japan

backup tree into packets. If more than one backup tree are unaffected, the controller selects one tree by the criteria that the network operators defined, for example, priority, minimum cost of all edges, and disjointness from trees used by other groups. This criteria is out of scope in this paper.

By switching a tree at the root switch, we can use any algorithm to compute trees, such as vertex redundant, edge redundant and least disjoint trees. In addition, the controller does not need to update flow entries in switches other than the root switch, and multicast packet delivery through a new tree is started quickly.

3.3 Group Membership Databases and Tree Database

To setup multicast trees, the multicast controller must find switches and ports where senders and receiver are attached. Such location data is stored in the group membership database for senders (the senders database) and that for receivers (the receivers database) in the controller. The controller also needs to hold tree status, such as active or inactive nodes and edges, and such status is also stored in the tree database.

The databases identify a multicast group by a pair of a sender IP address and a multicast IP address. The controller sometimes needs to look up the database without a sender IP address, for example, a receiver will join in or leave from groups without a sender IP address. In such cases, the controller uses 0.0.0.0 as a sender IP address, and the databases return results appropriately. For example, when the controller queries receivers belonging to the specific group, which means the group has a sender IP address other than 0.0.0.0, the database merges lists of receivers that specify the sender IP address when joining and that do not specify the sender IP address, and returns the merged list.

Each record in the senders database contains a switch ID and a port number of the root switch where a sender is connected. The controller will look up the senders database in two ways, specifying both a sender IP address and a multicast address to retrieve the location of a sender, and specifying only a multicast address to fetch a list of senders using a specific multicast address. To make these lookups fast, the senders database is indexed by multicast IP addresses using hash tables, and each entry in the hash tables is a list of the records that have the same multicast IP address.

The records in the receivers database contain a list of receivers. Each receiver is identified by a switch ID and a port number of the leaf switch where a receiver is connected, and it includes other data related to state management such as ones required by IGMP. The controller makes two kinds of queries to the receivers database. One is to retrieve one receiver with a sender IP address, a multicast address, a switch ID and a port number, and this type of query is often executed to update the state of a specific receiver. The other is to fetch all receivers joining in the group when updating multicast trees. To handle the latter case quickly, the receivers database is indexed by multicast IP addresses using hash tables, and receivers in each group are stored as a list per group.

The controller must be able to store several data per group in the tree database, such as multiple trees including backup trees, a subtree of each tree by indicating whether each node and edge is active or not, data to show which tree is used for packet delivery. When the controller updates trees by adding or removing a receiver, the controller must quickly look up the leaf node of each tree in the group, where the receiver is attached and used. When the controller handles failure recovery, the controller must find the failed node or edge in trees quickly to check whether the failed node or edge is active in trees. To meet above points, the tree database manages entries as follows. The top level record is an entry of a group (group entry hereafter), which consists of a list of trees and its IDs (tree entries), and the tree ID used for packet delivery in addition to (source IP address, multicast address) pair. The group entries are indexed by (source IP address, multicast address) pair. Each tree entry consists of a list of nodes and edges, and indexed by nodes. Each node and edge has a state, active or inactive, and pointers to the physical entity such as switch ID and and a port number.

4. Implementation of Prototype System

We have implemented our prototype multicast controller using C and Trema. **Figure 3** shows an overview of our design of the OpenFlow controller. Switch Communication, Event Dispatcher and Topology modules are provided as a part of Trema. Switch Communication module manages OpenFlow channels to the switches. The Event Dispatcher forwards messages from switches to pre-configured modules. Topology module collects network topology by sending and receiving LLDP packets from each port in switches. This module also handles switch status related messages like port status changes, and provides topology data to other modules.

The Multicast Controller in Fig. 3 is the core of multicast tree management. It consists of Multicast Tree Management module, Sender Management module, Receiver Management module, and Multicast Tree Switching module. The Multicast Packet Dispatcher dispatches packets in Packet-In messages from switches to the appropriate modules, such as IGMP packets to the Receiver Management module, and other packets to the Sender Management module.

The Sender Management module stores and updates the senders database, and provides an interface for the senders database to other modules. This module sets switches to forward unknown multicast packets to the controller. When the controller receives such packets, this module updates the senders database. If a new record is added to the senders database, this module sends a notification to the Multicast Tree Management module. In addition to the parameters described in Section 3.3, each record in the senders database includes a sender's MAC address for rewriting the source MAC address in packets to the original one when

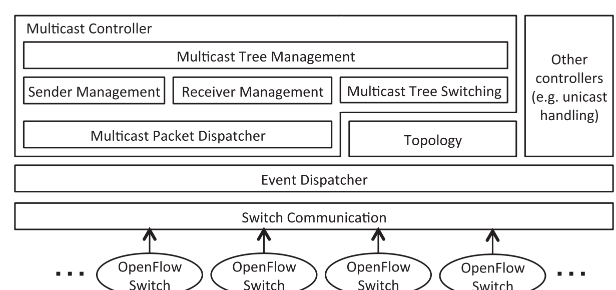

Fig. 3 An overview of our controller design.

© 2016 Information Processing Society of Japan

switches output the packets to receivers because the tree IDs are embedded in the source MAC address.

The Receiver Management module is in charge of the receivers database. To collect membership, this module acts as a multicast router in IGMP specification [8]. Data of the receivers is stored in the receivers database. When the receivers database is updated such as when a new receiver is added, this module notifies it to the Multicast Tree Management module. IGMP requires multicast routers to have a state of a receiver, and we include such state into the receivers database.

The Multicast Tree Management module is responsible for the tree database and flow entries regarding multicast packet forwarding. This module receives sender and receiver change events from the Sender Management and the Receiver Management modules, computes and updates multicast trees including backup trees in the tree database, and modifies flow entries in switches. This module retrieves records from the senders database in the Sender Management module and from the receivers database in the Receiver Management module if necessary. This module also receives and handles multicast tree change requests from the Multicast Tree Switching module.

The controller computes two trees that have few common edges by Dijkstra's SPF algorithm. The controller calculates a tree with the original costs of edges, and this tree is used for packet delivery. The backup tree is computed in the graph created by adding the sum of original costs of all edges to the costs of edges included in the tree for packet delivery. A tree ID is embedded in a source MAC address in packets.

The Multicast Tree Switching module handles topology change events. A link down event is notified to the Multicast Tree Switching module via the Topology module. For each multicast group, the Multicast Tree Switching module retrieves trees in the group including backup trees from the tree database in the Multicast Tree Management module, and checks whether the trees are disconnected by the link down. If the tree for packet delivery is disconnected but a backup tree is connected, this module sets the ID of the backup tree to the tree ID for packet delivery, and notifies the Multicast Tree Management module to update a flow entry in the root switch.

5. Evaluation

We have evaluated the processing time for group membership changes and failure recovery in our prototype controller, and packet loss duration at the time of failure using physical switches.

As a system for comparison, we implemented a controller that has the following modes for multicast tree management. One mode is whether the controller computes redundant trees for backup or not (Redundant or No Redundant). If this mode is Redundant, there is a mode to specify that the controller installs the trees other than the tree for packet delivery (Install or No Install). Another mode is whether the controller precomputes the trees or not (Precompute or No Precompute). If this mode is Precompute, the controller computes trees that cover all nodes and stores them in the tree database when the group is created, and if not, the controller computes trees every time that the controller needs to obtain new paths or trees, and only stores the necessary parts of the trees. In summary, there are six modes, Redundant - Install - Precompute (our proposed method), Redundant - Install - No Precompute, Redundant - No Install - Precompute (CastFlow [17] approach), Redundant - No Install - No Precompute, No Redundant - Precompute, and No Redundant - No Precompute.

5.1 Processing Time of Controllers for Group Membership Changes and Failure Recovery

We have evaluated how much our proposed method shortens the processing time of group membership changes and that of failure recovery in a controller. We measured the processing time to add a new receiver, to remove a receiver, and to restore packet delivery from a link failure.

We used a typical three-layered hierarchical topology shown in **Fig. 4**, which is a typical tree topology in private networks. The topology consists of the root switch and three layers, core, aggregation, edge layers, and they are connected in a redundant way. Each aggregation switch is connected to 10 edge switches, and one receiver is connected to each edge switch. In our evaluation, we use three types of networks that have one, two, and three sets of two aggregation and 10 edge switches, which have 10, 20, and 30 edge switches in total respectively.

We measured the time to add a first receiver and others separately, because these processes are different. The process in a first receiver includes creation of a new record in the tree database and tree computation, but the controller does not need compute trees when adding other receivers. Similarly, we measured the time to remove a last receiver and others separately because the former includes deleting a group entry from the tree database but the latter does not.

We measured the time to add a first receiver or to remove a last receiver 10 times each. We also measured the time to add or remove other receivers 10 times by adding or removing them one by one. To produce failures many times, we repeatedly set a link between the root switch and a core switch to down and up, then set the link to the other core switch to down and up. We produced failures 10 times per mode.

We recorded the time when the controller received port status messages and Packet-In messages containing IGMP messages, started the process of removing a receiver [*1], and sent flow modification (flow-mod) messages to switches.

We virtually constructed the network in a PC using the virtual ethernet link (veth) and Open vSwitch [22], which had Xeon X5355 CPU (2.66 GHz, 4 core), 8 GB RAM and run CentOS 6.4. The controller runs on another PC that had Core 2 Duo T7400

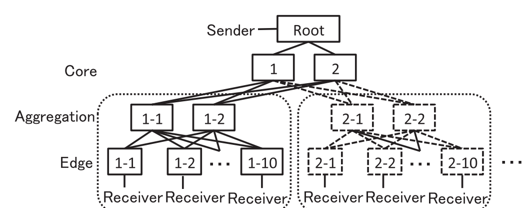

Fig. 4 A three-layer hierarchical topology for evaluation.

[*1] IGMP [8] requires a router to delay removing a receiver until the router confirms that no other receiver is left.

© 2016 Information Processing Society of Japan

CPU (2.16 GHz, 2 core), 4 GB RAM and runs Ubuntu 12.04, and was directly connected to the PC running Open vSwitch at 1 Gbps.

Figures 5 and **6** shows the processing time in the controller to add a first and another receiver. **Figures 7** and **8** shows the processing time in the controller to remove a last or another receiver. The x-axis shows the number of edge switches, and the y-axis shows the processing time in milliseconds. The arrows point to the lines corresponding to the modes. The values are the average time we have measured.

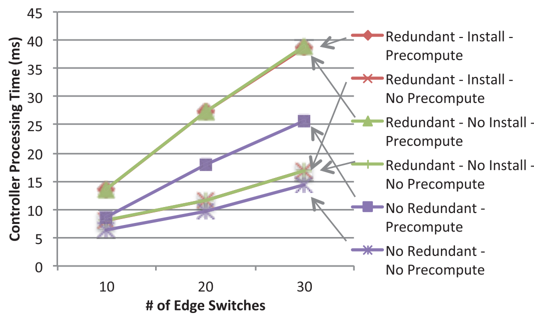

Fig. 5 Processing time to add a first receiver.

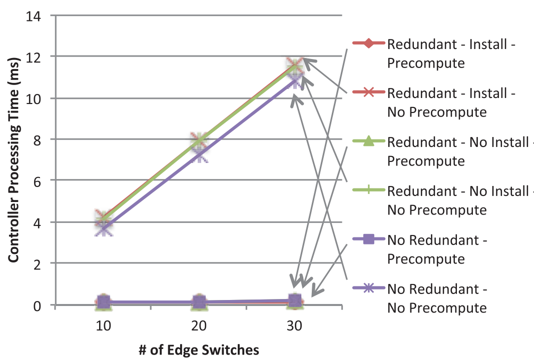

Fig. 6 Processing time to add a receiver other than the first one.

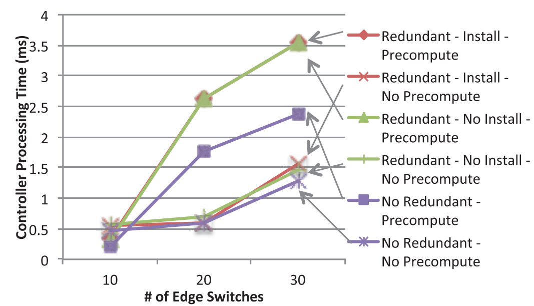

Fig. 7 Processing time to remove a last receiver.

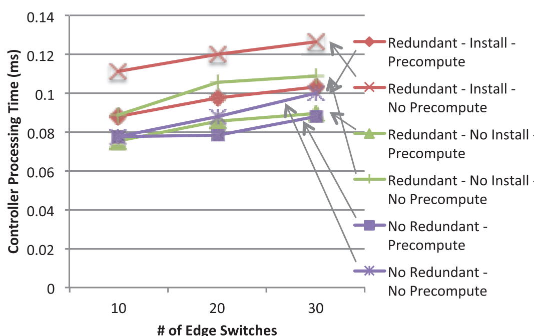

Fig. 8 Processing time to remove a receiver other than the last one.

Figure 5 shows the overhead required to compute redundant trees. The time in "Redundant - No Install - Precompute" is about 1.5 times longer than the time in "No Redundant - Precompute." Another overhead is to precompute trees. For example, in "Redundant - Install" cases, the time in the "Precompute" case is about 2 to 3 times longer than the time in the "No Precompute" case. There is little difference between "Redundant" and "No Redundant" cases.

In the case of adding another receiver than the first one (Fig. 6), a trend is changed. In the "Precompute" cases, the processing time is less than 1 ms regardless of the number of switches, although in the "No Precompute" cases it takes more than 3 ms and the time increases as the number of switches increases.

In Fig. 7, the controller takes more time to remove the last receiver in the "Precompute" cases than in the "No Precompute" cases. In the "Precompute" cases, it takes about 1.5 times longer time in the "Redundant" cases than in the "No Redundant" cases. There is little difference between three "No Precompute" cases. As with the case of adding a first receiver (Fig. 5), there is little difference between the "Redundant" and "No Redundant" cases.

According to Fig. 8, there seems to take a little longer time to remove a receiver other than the last one. Overall, the values are small compared to the cases of adding receivers and removing the last receiver.

Figure 9 shows the processing time to restore packet delivery from a link failure. As with the addition and removal of receivers, the x-axis shows the number of edge switches, and the y-axis shows the processing time in milliseconds. Figure 9 shows only the cases where trees are precomputed because the procedures are not changed whether trees are precomputed or not. The values are the average time we measured.

In Fig. 9, the time in "No Redundant - No Install" is increased in proportion to the number of edge switches. The time in the other redundant cases are almost the same regardless of the number of edge switches, and much shorter than in no redundant case.

5.2 The Recovery Time of the Multicast Tree

For fast failure recovery, it is important that switches install new flow entries quickly as well as shortening the processing time in a controller. We have measured the duration that receivers do not receive multicast packets when a failure occurs in a network using a physical switch.

We used a topology in **Fig. 10**. The Root Switch, Core Switch 1 and Leaf Switch were software switch, and the Core Switch 2 was a physical switch. The sender was connected to the root switch,

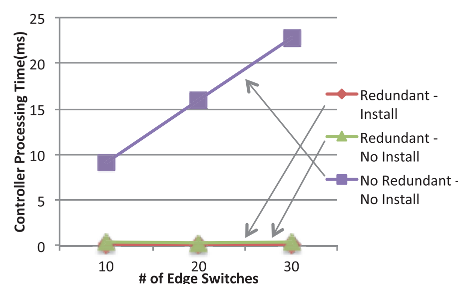

Fig. 9 Processing time to recover from a link failure.

© 2016 Information Processing Society of Japan

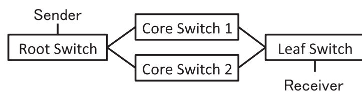

Fig. 10 A topology used for evaluation of packet loss duration during a failure recovery.

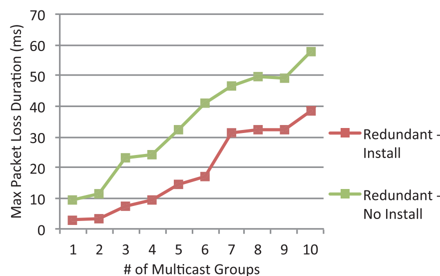

Fig. 11 Maximum packet loss duration when a link between root switch and core switch 1 is down.

and the receiver was in the edge switch. To induce a failure where installation to flow entries to the physical switch is necessary, we shut down a port on the root switch to the Core Switch 1. This makes the controller installing flow entries to the Core Switch 2 (Physical Switch) when backup trees are not installed in advance, but no installation to the Core Switch 2 is necessary when backup trees are installed in advance. We produced failures 10 times in total.

A sender transmitted packets to each group with incremental sequence numbers every millisecond. The receivers monitored sequence numbers and intervals of packet arrivals, and recorded the intervals when the number in a packet was jumped to two or more bigger value at once.

We created one to ten multicast groups. The root switches in all groups are Root Switch, and the receivers of all groups are connected to Leaf Switch. In each failure, we regard the max packet loss duration as the largest value of packet loss duration in multicast groups that exists.

We configured software switches on one PC, which had Intel Atom C2358 CPU and 4 GB RAM. The physical switch was NEC PF5240. The sender and receiver PCs were almost the same as the controller in Section 5.1, but the size of RAM was 1 GB in the sender PC and 512 MB in the receiver PC. Both were connected to switches at 1 Gbps.

Figure 11 shows maximum packet loss duration when a failure occurs. The x axis shows the number of multicast groups exists, and the y axis the max packet loss duration on average. As with Fig. 9, Fig. 11 includes only cases where trees are precomputed. No Redundant case is not included because it would take longer than in Redundant - No Install case due to calculating trees.

The values are unstable when more multicast groups exists, especially seven to ten groups. We believe this is due to a jitter added by software switches.

6. Discussion

The results of the processing time in the controller show that the tree precomputation approach can dramatically shorten the time to add a receiver except a first receiver. This reduction is because, as we have expected, a controller uses the same tree once the controller calculates a tree in the tree precomputation approach, instead of calculating trees every time that a receiver joins in a group. The controller can add a receiver in $O(N)$ with the tree precomputation approach, but tree computation requires $O(N^2)$ for example in Dijkstra's algorithm, where N is the number of switches. We cannot see a large difference between cases where redundant trees are installed or not at the time of adding a receiver.

When a first receiver joins in, the controller performs more processes in the tree precomputation approach, including computing a tree, allocating memory space for a new entry in the databases, etc. This is the reason why the controller in the tree precomputation approach takes longer time to add a first receiver. When the controller computes redundant trees, it takes more time to compute additional trees. As with the case of adding another receiver, there seems little overhead on the processing time by installing redundant trees.

Considering many multicast applications such as video streaming and data distribution to many hosts, taking more time in a first receiver should not be a significant problem. In such applications, a controller often performs the processes to add a receiver to an existing group rather than to a new group, because a new group would be rarely created.

As for the processing time to remove a last receiver from a group (Fig. 7), the controller takes longer with the tree precomputation approach than without precomputing trees. There is also a difference whether the controller uses redundant trees or not. This is due to the time to delete trees from the tree database. The controller deletes all trees used by the group when the group has no receiver. If the trees are precomputed, the controller should deallocate memory space for all nodes and edges one by one, but the controller only deallocate memory space for the path from the root node to the left node if the trees are not precomputed. There is a room to optimize this procedure, such as allocating and deallocating memory space for entries in the tree database at once.

For removing another receiver, the controller takes slightly longer without the tree precomputation approach than with the tree precomputation approach. The reason is that the controller without the tree precomputation approach deallocates memory space for unnecessary nodes and edges, but the controller in the tree precomputation approach does not need to deallocate it.

In summary, our proposed method greatly increases the controller's capacity to handle addition or removal of receivers to or from existing groups, which is one of important characteristics in private networks. It takes a slightly longer time to create or remove a group entry in the tree database, but it is not a significant problem because it merely occurs in practice compared to addition or removal of receivers in existing multicast groups.

Our proposed method can also reduce the time to repair multicast trees. The processing time for failure recovery (Fig. 9) increases linearly as the network size increases in "No Redundant - No Install" case. The processing times in other two cases ("Redundant - Install" and "Redundant - No Install") are greatly shorter than "No Redundant - No Install" case, and hardly increase as the network grows. This is because the controller recalculates a tree if no redundant or backup tree is available.

There is little difference between "Redundant - Install" and

© 2016 Information Processing Society of Japan

"Redundant - No Install" in terms of the processing time in the controllers, but there is a large difference when we consider the number of flow entries that should be installed in physical switches (Fig. 11). The larger the number of flow entries the controller requires to install in physical switches, more time it takes until multicast packet delivery in all multicast trees. Although the values in Fig. 11 are slightly smaller than reported [12], we can see the tendency that software switches handles flow modifications faster than physical switches. We believe the difference between [12] and us would be due to a small difference in implementation of switches.

When a redundant or backup tree is available and installed, the controller only needs to replace a flow entry in the root switch, therefore the number of flow entries that are modified at a single switch can be reduced, especially at the core of networks such as core switches. Our proposed method cannot work effectively when many multicast groups share the same root switch, but we can avoid this by allocating senders to different root switches, using a software switch as a root switch, etc.

Considering a video streaming as an application of multicast, it would be better to minimize frame loss, which corresponds to packet loss in the network layer, because losing one frame disturbs several frames [16]. Our proposed method can reduce the possibility of frame loss because our proposed method can restore packet delivery quickly. Let us assume that a frame rate of a video is 30, which is close to values used in television. In this case, frames are sent at the interval of about 33 msec. According to the results of our evaluation (Section 5.2), one frame should be lost without our proposed method if more than five multicast groups exists, but our proposed method can increase the number of multicast groups to seven or more. In addition, as we have mentioned, the packet loss duration can be reduced when root switches are not shared among multicast groups.

In real situations, there are some factors to make the packet loss duration longer. One factor is a failure detection delay. In our experiments, we set interfaces to down, and a failure detection delay in a switch was very low. If switches or other devices cannot detect failures immediately, the packet loss duration would increase due to the detection delay.

The other factor is latency. Because we connected each virtual switch and the PCs directly and all devices were in our lab, the latencies among switches and the controller were very low. When the latency becomes high, the packet loss duration would be long because packets are rerouted at the root switch. Our proposed method, "Redundant - Install - Precompute," is beneficial in such cases because the processing time in the controller is shorter and latencies among switches and the controller equally increase the packet loss duration in all cases. In real environments such as campus or enterprise networks, latency would be small because switches are located in a small area, and latency would be smaller than the time to compute trees or to install flow entries in switches. Therefore, the problem caused by latency would not be significant.

One of the overheads in our proposed method is the memory usage. When a controller precomputes and stores multiple trees, the controller uses more memory space than the cases where trees are not precomputed or the controller stores one tree per group. In this perspective, our proposed method uses the largest memory space in all cases that we have evaluated. Although total size of the tree database depends on the tree size and the number of multicast groups, we think this problem is not significant because tens of gigabytes of memory are available in current PCs. For example, in our proposed method, the data size of each node and edge on a tree is less than 20 bytes, and more than 53 million nodes and edges in total can be stored per GB. In addition, when a group has more receivers, trees would cover most of switches, and the overhead becomes small.

Another overhead is the increase of the number of flow entries that switches store due to trees for backup. This overhead can be negligible in typical private networks, which often have hierarchical tree topology and redundant networking devices and links in each layer. In such networks, when a failure occurs, the traffic is generally rerouted to another device in the same layer. In the case of multicast using OpenFlow without our proposed method, flow entries are removed from switches, and installed to other switches where multicast packets are rerouted. In other words, flow entries for backup trees are installed at the time of failures, and a capacity of each switch, such as the number of flow entries that a switch can install, must be designed with consideration of backup trees. In this perspective, our proposed method just installs flow entries that are needed to reroute packets before a failure occurs, and switches do not require extra capacity to support our proposed method.

One of the problems in MPLS based approaches is that all switches or routers on a tree must handle messages to modify the tree because of the specifications of a signaling protocol, RSVP-TE. This means that switches or routers must process messages to change some parts of trees even when they do not have to change their state. By calculating the difference of multicast trees centrally, only switches that have to modify its flow entries need to process flow modification messages from the controller. Furthermore, switches near the core of a network do not have to modify its flow entries frequently if the group has many receivers and its tree covers many parts of the network. These characteristics would greatly decrease loads of switches.

There is a tradeoff where to reroute multicast packets, or how to construct backup trees before a failure occurs. One choice is to reroute packets at switches where a failure is detected, called local repair, and this approach is used by MPLS Fast Reroute and the fast failover group feature in OpenFlow 1.1 and later. Mechanisms based on local repair can dramatically reduce packet loss when latency between routers and switches is long such as WAN, but trees used by local repair after failures are detected are not necessarily efficient in terms of metrics such as the number of flow entries used by backup trees (backup trees or paths from each switch must be created), bandwidth usage, and the number of routers and switches that a tree covers (rerouted packets may go through additional links to reach to receivers via backup paths).

Another choice is to reroute packets at routers or switches near the root router or switch. This choice has a benefit that backup trees can be created with various metrics, such as minimizing

© 2016 Information Processing Society of Japan

routers or switches that rerouted packets go through, and avoid wasting bandwidth caused by rerouting packets. A drawback in this choice is the increase of packet loss at the time of failures because packets during delivery on a network are lost, and there seems no way to notify failures at a router or a switch that reroute packets. The former is negligible in private networks because latencies between switches are small. The latter can be solved by centrally managing multicast trees.

Rerouting packets at a switch other than the root switch is the same with executing local repair at the switch, and still has a problem that the number of flow entries used by backup increases in local repair approach. If we can prepare for a backup tree that cover the failures in any switch, we can reroute packets at the root switch, and only one backup tree is needed to recover multicast trees. The overhead caused by one backup tree is acceptable as we have explained above.

7. Conclusion

In this paper, we propose a method to manage multicast trees in an OpenFlow controller in private networks, which supports both dynamic membership changes and fast failure recovery that takes into account slow flow entry setup performance in physical switches. Our key idea is to compute trees covering all switches in advance, and to change status of each node or edge when a receiver is added or removed. This reduces the number of tree computation times. For failure recovery, our proposed method allows setting up multiple trees including backup trees in advance, which are computed by any algorithm, and to switch to another tree used for packet delivery with minimum modifications to flow entries.

Our evaluation using our prototype controller shows our proposed method can reduce the processing time in the controller to add or remove a receiver and to recover trees from failures, and we also show that packet loss duration due to a failure in physical network is also reduced. We also show that there is little difference in the controller processing time whether redundant trees are installed in advance or not, but the difference becomes large in physical switches due to the number of flow entries to be modified. Although our proposed method has some disadvantages, these disadvantages would not be significant in practice because these rarely occurs, and PCs running a controller program often have enough computing resources to reduce the impact of these problems, and the characteristics of private networks.

Future work would include reducing the number of flow entries in switches. Our proposed method requires at least two flow entries per group to implement fast failure recovery. There is still ample margin to reduce the number of flow entries if the controller can summarize flow entries among groups. Another one is to improve fast failure recovery in networks that have high latencies. If latencies are high, the packet loss duration becomes long in our proposed method.

References

[1] Adams, A., Nicholas, J. and Siadak, W.: Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised), RFC 3973, IETF (2005).

[2] Aggarwal, R., Papadimitriou, D. and Yasukawa, S.: Extensions to Resource Reservation Protocol - Traffic Engineering (RSVP-TE) for Point-to-Multipoint TE Label Switched Paths (LSPs), RFC 4875, IETF (2007).

[3] Capone, A., Cascone, C., Nguyen, A. and Sanso, B.: Detour planning for fast and reliable failure recovery in SDN with OpenState, *IEEE DRCN 2015*, pp.25–32 (2015).

[4] Cui, J.-H., Faloutsos, M. and Gerla, M.: An architecture for scalable, efficient, and fast fault-tolerant multicast provisioning, *IEEE Network*, Vol.18, No.2, pp.26–34 (2004).

[5] Dijkstra, E.W.: A note on two problems in connexion with graphs, *Numerische Mathematik*, Vol.1, pp.269–271 (1959).

[6] Estrin, D., Farinacci, D., Helmy, A., Thaler, D., Deering, S., Handley, M., Jacobson, V., Liu, C., Sharma, P. and Wei, L.: Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification, RFC 2117, IETF (1997).

[7] Fei, A., Cui, J., Gerla, M. and Faloutsos, M.: Aggregated multicast: An approach to reduce multicast state, *IEEE GLOBECOM '01*, Vol.3, pp.1595–1599 (2001).

[8] Fenner, W.: Internet Group Management Protocol, Version 2, RFC 2236, IETF (1997).

[9] Open Networking Foundation: OpenFlow Switch Specification version 1.1 (2011).

[10] Gyllstrom, D., Braga, N. and Kurose, J.: Recovery from link failures in a Smart Grid Communications, *IEEE SmartGridComm 2014*, pp.254–259 (2014).

[11] Hasegawa, T., Kamimura, K., Hoshino, H. and Ano, S.: IP-based HDTV broadcasting system architecture with non-stop service availability, *IEEE GLOBECOM '05*, Vol.1, pp.337–342 (2005).

[12] Huang, D.Y., Yocum, K. and Snoeren, A.C.: High-fidelity Switch Models for Software-defined Network Emulation, *HotSDN '13*, pp.43–48, ACM (2013).

[13] Kotani, D., Suzuki, K. and Shimonishi, H.: A design and implementation of OpenFlow Controller handling IP multicast with Fast Tree Switching, *IEEE/IPSJ SAINT '12*, pp.60–67 (2012).

[14] Li, G., Wang, D. and Doverspike, R.: Efficient Distributed MPLS P2MP Fast Reroute, *IEEE INFOCOM '06*, pp.1–11 (2006).

[15] Li, X. and Freedman, M.J.: Scaling IP Multicast on Datacenter Topologies, *ACM CoNEXT '13*, pp.61–72 (2013).

[16] Liang, Y.J., Apostolopoulos, J.G. and Girod, B.: Analysis of Packet Loss for Compressed Video: Effect of Burst Losses and Correlation Between Error Frames, *IEEE Trans. Circuits and Systems for Video Technology*, Vol.18, No.7, pp.861–874 (2008).

[17] Marcondes, C., Santos, T., Godoy, A., Viel, C. and Teixeira, C.: CastFlow: Clean-slate multicast approach using in-advance path processing in programmable networks, *IEEE ISCC'12*, pp.94–101 (2012).

[18] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. and Turner, J.: OpenFlow: Enabling innovation in campus networks, *ACM SIGCOMM Comput. Commun. Rev.*, Vol.38, pp.69–74 (2008).

[19] Médard, M., Finn, S.G. and Barry, R.A.: Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs, *IEEE/ACM Trans. Netw.*, Vol.7, pp.641–652 (1999).

[20] Mochizuki, K., Shimizu, M. and Yasukawa, S.: Multicast Tree Algorithm Minimizing the Number of Fast Reroute Protection Links for P2MP-TE Networks, *IEEE GLOBECOM '06*, pp.1–5 (2006).

[21] Moy, J.: Multicast Extensions to OSPF, RFC 1584, IETF (1994).

[22] Open vSwitch, available from ⟨http://openvswitch.org/⟩.

[23] Pan, P., Swallow, G. and Atlas, A.: Fast Reroute Extensions to RSVP-TE for LSP Tunnels, RFC 4090, IETF (2005).

[24] Reitblatt, M., Canini, M., Guha, A. and Foster, N.: FatTire: Declarative Fault Tolerance for Software-defined Networks, *HotSDN '13*, pp.109–114, ACM (2013).

[25] Siachalou, S. and Georgiadis, L.: Algorithms for precomputing constrained widest paths and multicast trees, *IEEE/ACM Trans. Networking*, Vol.13, No.5, pp.1174–1187 (2005).

[26] Siew, D.C.K. and Gang, F.: Tree-Caching for Multicast Connections with End-to-End Delay Constraint, *IEICE Trans. Communications*, Vol.84, No.4, pp.1030–1040 (2001).

[27] Trema: Full-Stack OpenFlow Framework for Ruby/C, available from ⟨http://trema.github.com/trema/⟩.

[28] Waitzman, D., Partridge, C. and Deering, S.E.: Distance Vector Multicast Routing Protocol, RFC 1075, IETF (1988).

[29] Whetten, B., Vicisano, L., Kermode, R., Handley, M., Floyd, S. and Luby, M.: Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer, RFC 3048, IETF (2001).

[30] Wijnands, I., Minei, I., Kompella, K. and Thomas, B.: Label Distribution Protocol Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths, RFC 6388, IETF (2011).

[31] Xue, G., Chen, L. and Thulasiraman, K.: Quality-of-service and quality-of-protection issues in preplanned recovery schemes using re-

© 2016 Information Processing Society of Japan

dundant trees, *IEEE Journal on Selected Areas in Communications*, Vol.21, No.8, pp.1332–1345 (2003).
[32] Yasukawa, S.: Signaling Requirements for Point-to-Multipoint Traffic-Engineered MPLS Label Switched Paths (LSPs), RFC 4461, IETF (2006).
[33] Yu, Y., Zhen, Q., Xin, L. and Shanzhi, C.: OFM: A Novel Multicast Mechanism Based on OpenFlow, *Advances in Information Sciences and Service Sciences* (*AISS*), Vol.4, No.9, pp.278–286 (2012).
[34] Zou, J., Shou, G., Guo, Z. and Hu, Y.: Design and implementation of secure multicast based on SDN, *IEEE IC-BNMT '13*, pp.124–128 (2013).

Daisuke Kotani is a Ph.D. student in Graduate School of Informatics, Kyoto University, Kyoto Japan. He received B.E. and M.Inf. degrees from Kyoto University in 2011 and 2012 respectively. His research interests include Internet architectures and distributed computing.

Kazuya Suzuki received M.E. degree in Electrical Engineering from Tokyo Metropolitan University in 1997, and Ph.D. degree in Systems Management from University of Tsukuba in 2011. He is currently a principal researcher of the Cloud Systems Research Laboratories, NEC Corporation and a visiting associate professor of Graduate School of Information Systems, the University of Electro-Communications. His research interests include availability improvement in unicast routing and software-defined networking.

Hideyuki Shimonishi received M.E. and Ph.D. degrees from the Graduate School of Engineering Science, Osaka University, Osaka, Japan, in 1996 and 2002. He joined NEC Corporation in 1996 and has been engaged in research on traffic management in high-speed networks, switch and router architectures, and traffic control protocols. As a visiting scholar in the Computer Science Department at the University of California at Los Angeles, he studied next-generation transport protocols. He now works in NEC Cloud System Research Laboratories and is engaged in research on technologies for future Internet architectures including SDN and NFV.

© 2016 Information Processing Society of Japan