Chapter 4

Lecture 4: Multicast and Group Tables

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/Coe8agGmalUVt3I` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_4` in which you can do the exercises. You will also need a sample video for this exercise that is available here: `https://surfdrive.surf.nl/files/index.php/s/4UNli9HfzxQtYp9`

Solution

The solution files are available here: `https://surfdrive.surf.nl/files/index.php/s/NOBxYO9j1ci7rqG`

4.1 Implementing Multicast in OpenFlow

Multicast is an efficient method for sending the same data to a group of hosts, as the same packet is not transmitted through the same link multiple times. For example, multicast would be very effective for streaming video content to multiple devices. In data centers, it is applied to significantly reduce the bandwidth required for group communication.

SDN is nicely suited to multicast traffic, as the controller has a central view of the whole network and can thus easily compute and install multicast trees. In these exercises, you will implement a multicast controller app yourself and test it by streaming a video to multiple hosts.

A complete version of this application should have the following functionality:

- Receive all IGMPv3 packets sent to the network.
- Maintain multicast groups [1] automatically based on INCLUDE and EXCLUDE packets (we will ignore all other IGMPv3 packets).
- When a host starts sending (non-IGMP) packets to a multicast group, install flow and group entries in the network to send these packets to all members of the group. *Packets should not be sent over the same link multiple times!*
- When a host leaves a multicast group, ensure no packets from that group are being sent to it anymore by removing or updating installed flow and group entries.
- When a host joins a multicast group, ensure packets being sent to that group are also being sent to this host, by adding or updating installed flow and group entries.

4.2 Exercise 1. - Processing IGMPv3 packets

We provide you a basic Ryu controller app that keeps track of the network topology and logs all incoming packets. In addition, the app installs a default table-miss entry in each switch that sends any unmatched packets to the controller.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```

[1] In your app

```
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto
from ryu.lib.packet import igmp

from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link, get_host

import networkx as nx

class Multicast_Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Multicast_Controller, self).__init__(*args, **kwargs)

        self.network = nx.DiGraph()
        self.subscribers = {}

    def log(self, message):
        self.logger.info(message)
        return

    #This function is triggered before the topology controller flows are added
    #But late enough to be able to remove flows
    @set_ev_cls(ofp_event.EventOFPStateChange, [CONFIG_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Delete any possible currently existing flows.
        del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL, out_port=ofp.OFPP_ANY,
            out_group=ofp.OFPG_ANY, command=ofp.OFPFC_DELETE)
        dp.send_msg(del_flows)

        #Delete any possible currently exising groups
        del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
            group_id=ofp.OFPG_ALL)
        dp.send_msg(del_groups)

        #Make sure deletion is finished using a barrier before additional flows are added
        barrier_req = parser.OFPBarrierRequest(dp)
        dp.send_msg(barrier_req)

    #Switch connected
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Add default flow
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)

    #Topology Events
    @set_ev_cls(event.EventSwitchEnter)
    def switchEnter(self,ev):
```

```
        switch = ev.switch

        self.network.add_node(switch.dp.id, switch = switch, flows= {}, host = False)
        self.log('Added switch ' + str(switch.dp.id))

    @set_ev_cls(event.EventSwitchLeave)
    def switchLeave(self,ev):
        switch = ev.switch
        sid = switch.dp.id

        if sid in self.network:
            #NOTE: In actual applications one should include some code to properly handle switches leaving
            #and re-entering!

            self.network.remove_node(sid)
            self.log('Removed switch ' + str(sid))

    @set_ev_cls(event.EventLinkAdd)
    def linkAdd(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        self.network.add_edge(src, dst, src_port = link.src.port_no, dst_port = link.dst.port_no, live = True)
        self.log('Added link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventLinkDelete)
    def linkDelete(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        if (src,dst) in self.network.edges():
            #NOTE: In actual applications one should include some code to properly handle link failures!

            self.network.remove_edge(src, dst)
            self.log('Removed link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventHostAdd)
    def hostFound(self,ev):
        host = ev.host
        switch = host.port.dpid
        mac = host.mac

        self._hostFound(switch, host.port.port_no, mac)

    def _hostFound(self, switch_id, port, mac):
        if mac not in self.network:
            self.network.add_node(mac, host = True)
            self.network.add_edge(mac, switch_id, src_port = -1, dst_port = port, live = True)
            self.network.add_edge(switch_id, mac, src_port = port, dst_port = -1, live = True)
            self.log('Added host ' + mac + ' at switch ' + str(switch_id))

    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        pkt = packet.Packet(msg.data)
        eth = pkt[0]
```

```
        if eth.protocol_name != 'ethernet':
            #Ignore non-ethernet packets
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return

        self.log('Received ethernet packet')
        src = eth.src
        dst = eth.dst
        self.log('From ' + src + ' to ' + dst)

        #host detection
        if src not in self.network:
            self._hostFound(dp.id, msg.match['in_port'], src)

        if eth.ethertype == ether_types.ETH_TYPE_IP:
            self.process_ip(dp, pkt)
            return

    def process_ip(self, dp, pkt):
        eth_pkt = pkt[0]
        ip_pkt = pkt[1]

        #IGMP message
        if ip_pkt.proto == in_proto.IPPROTO_IGMP:
            igmp_pkt = pkt[2]
            self.processIGMP(eth_pkt.src, ip_pkt.src, igmp_pkt)
            return

    def processIGMP(self, eth_src, ip_src, igmp_pkt):
        self.log('IGMP packet')
        # Implement this function yourself
```

Now, to be able to support hosts running IGMPv3, the app should at the very least be able to parse IGMPv3 membership reports. Whenever a host wants to join or leave a multicast group, it will send a membership report to the network. An IGMPv3 membership report is basically just a list of *group records*, one for each group the host wants to leave or join (or report on). Each of these group records, in turn, contains a multicast address (the group!), a list of source addresses, and a record field type. By including a list of source addresses, IGMPv3 allows hosts to make their own selection of source hosts to listen to. However, in this exercise, we will assume hosts always want to listen to all sources in a multicast group. Thus, we only support group records with empty source address lists.

If a host wants to join a group, the group record will have the `CHANGE_TO_EXCLUDE_MODE` type and an empty source list (indicating it wants to exclude none of the sources, i.e., listen to the whole group). Conversely, when a host wants to leave a group, the group record will have the `CHANGE_TO_INCLUDE_MODE` type and an empty source list.

Exercise:

Extend your app with the ability to process IGMPv3 report messages. For now, you only have to keep track of which hosts are subscribed to which multicast groups. You can use the Packet library (`https://ryu.readthedocs.io/en/latest/library_packet_ref/packet_igmp.html`) to extract all relevant properties. You can determine if an (IPv4) packet is an IGMP packet by importing `ryu.lib.packet.in_proto` and testing if `ip_pkt.proto == in_proto.IPPROTO_IGMP` (where `ip_pkt` is the extracted IPv4 packet. For any IGMP message `igmp_pkt`, you would have to do the following:

- Check if the message is an IGMPv3 membership report (`igmp_pkt.protocol_name == 'igmpv3_report'`), do nothing if not.
- Ignore any IGMPv3 messages with non-empty source lists
- For each group record `record`
  - If `record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE`, add the source host of the packet to the multicast group
  - If `record.type_ == igmp.CHANGE_TO_INCLUDE_MODE`, remove the source host of the packet from the multicast group

- Log any changes you make to the multicast groups

Hint: use `self.subscribers` to store subscribers when implementing the `processIGMP` function.

You can test your app as follows:

1. Start Mininet with the binary tree topology of depth 3 and connect it to a Ryu controller running your app.
2. Add a multicast route to the route table of hosts `h1`, `h2`, `h7`, and `h8` (if you want, you can create an automation script for this, as you will need to repeat this step for the next two exercises as well):
   `h* route add -net 224.0.0.0 netmask 240.0.0.0 dev h*-eth0`
3. Connect hosts `h1`, `h2`, and `h8` to an imaginary multicast stream; for each host:
   (a) Start vlc (if VLC does not start properly or complains about being run in root mode, run the following command (outside mininet) first: `sudo sed -i 's/geteuid/getppid/' /usr/bin/vlc`)
   (b) Click on `media -> Open Network Stream`
   (c) Fill in `udp://@224.0.0.14` as the network address and press `play`
   (d) You should see the host being added to the group (one or multiple times, depending on how many messages VLC decides to send)
4. Shut down VLC on all hosts one by one. Check if the app has removed the hosts from the group again.

Solution

```python
def processIGMP(self, eth_src, ip_src, igmp_pkt):
    self.log('IGMP packet')

    #Only support IGMPV3 membership reports
    if igmp_pkt.protocol_name != 'igmpv3_report':
        return

    records = igmp_pkt.records
    for record in records:
        if not record.srcs:
            address = record.address
            self.log('Record change for group ' + address)

            if record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE:
                if address not in self.subscribers:
                    self.subscribers[address] = set()

                self.subscribers[address].add(eth_src)

                self.log('Added ' + eth_src)

            elif record.type_ == igmp.CHANGE_TO_INCLUDE_MODE:
                if address not in self.subscribers:
                    self.subscribers[address] = set()

                self.subscribers[address].discard(eth_src)

                self.log('Removed ' + eth_src)
```

4.3 Exercise 2. - Routing multicast traffic

When a host starts sending traffic to a multicast group, the app should route this traffic to all subscribers of the group. For now, we will assume that all subscribers are known as soon as a source host starts streaming. Thus, your app will only have to install new flow and group entries when it receives a multicast packet from a new source host + multicast destination address combination. To support multicast, we will sometimes need to output a single packet over multiple ports. To do so, we will make use of the group table.

4.3.1 Group Table

Besides outputting packets directly to a port, OpenFlow also allows flow entries to indicate that packets should be processed by a *group* instead. A group is essentially an ordered list of *action buckets*: a set of actions and associated parameters. The way packets are processed by the group depends on its type:

- **all**: The all group, which we will use in this exercise, simply executes the actions of all its action buckets. Thus, if we insert an action bucket for each output port, the packet will be forwarded to each of these ports.
- **select**: The select group is quite interesting, as it allows the controller to divide traffic load over multiple paths. The select group executes one action bucket per packet. The trick is that it selects this bucket in such a way that overall, the load is equally shared across all action buckets in the group.
- **indirect**: The indirect group only allows the controller to specify one action bucket, and will simply execute this single bucket.
- **fast failover**: The fast failover group is used for rapidly switching over to a backup path after a link or node failure, without requiring any response from the controller. Each action bucket in a fast failover group is associated with a specific port (or even another group). The group executes the first bucket with a *live* port. This means that as soon as the port of the first bucket (forwarding packets over the primary path) goes down, it will automatically start executing the second bucket (forwarding packets over a backup path) instead.

You might wonder why we need the indirect, or even the all group, as we can also output packets to one or more ports by directly applying one or more `OFPActionOutput` actions. The main benefit of both of these group types is that we can point multiple flow entries to the same group. This not only allows us to change the behavior of multiple flow entries simultaneously but can also help from an organizational perspective. For example, a network operator could pre-install a spanning tree in the network, using the all group, and broadcast any packets to all nodes in the network simply by forwarding them to the pre-installed group entries.

Furthermore, not all switches necessarily support forwarding a packet to multiple ports simultaneously by applying multiple `OFPActionOutput` actions. In contrast, all OF 1.3 switches are required to support the all group type. However, the main reason for using group entries in these exercises is simply to learn how to install and use group entries, as they are an essential part of the OpenFlow protocol.

4.3.2 Installing OF Group Entries - Example

You can install an all group as follows:

```
dp = self.network.nodes[switch_id]['switch'].dp
ofp = dp.ofproto
parser = dp.ofproto_parser

ports = [1,2,5]

action_lists = [[parser.OFPActionOutput(port)] for port in ports]
buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list) for action_list in action_lists]

#Make sure to choose a unique group_id for each group entry!
cmd = parser.OFPGroupMod(dp, ofp.OFPGC_ADD, ofp.OFPGT_ALL, group_id, buckets)
dp.send_msg(cmd)
```

To send packets to the group, use the `OFPActionGroup` action:

```
match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=dst_address)
actions = [parser.OFPActionGroup(group_id)]
instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
cmd = parser.OFPFlowMod(datapath=dp, priority=2,
    match=match, instructions=instr)
dp.send_msg(cmd)
```

4.3.3 Exercise Instructions

Multicast packets can be recognized by their Ethernet address. If the least significant bit of the first octet is a 1, it is a multicast address. For any multicast packet,

- IGMP packets also have a multicast address, so first determine if the packet is an IGMP packet or "normal" multicast traffic.

- Extract the destination IP address (IGMP uses IP multicast addresses, and not Ethernet addresses)
- If the source host + multicast destination combination is already known, we can ignore the packet (as we already installed all required flow and group entries).
- Otherwise, the app needs to install the required flow and group entries to route traffic from the source host to all subscribers of the group. The easiest way to do so is to compute and install the shortest path between each subscriber and the source host (this will result in a Shortest-Path Tree (SPT)). We just need to make sure that we only install one flow and group entry in each switch and do not duplicate packets over the same link. To route packets from the source to one or more subscribers, you will need to check each switch on the shortest paths:
  - If the switch already forwards packets to the next switches on the path: do nothing
  - Otherwise, and if the switch already contains a group entry for this multicast group: modify this group entry (`OFPGroupMod(dp, ofp.OFPGC_MODIFY, ...)`) to add additional action bucket(s).
  - Otherwise, install both a new all group entry, as well as a new flow entry pointing to this group.

  Hint: You can either match on both the source address and the IP destination address, or only on the IP destination address. In the latter case, you only need to install up to a single flow and group entry in each switch for each multicast group (saving memory). While in the former case you need to install entries for each source host + multicast group combination.

Similarly to the `dump-flows s*` command, you can use
`sudo ovs-ofctl --protocols=OpenFlow13 dump-groups s*` to show all group entries of switch `s*`.

To test your app, you will stream a video from one host to the others. Download the video from here if you haven't done so yet: `https://surfdrive.surf.nl/files/index.php/s/4UNli9HfzxQtYp9`

Test your app as follows:

1. Start Mininet with the binary tree topology of depth 3 and connect it to a Ryu controller running your app.
2. Add a multicast route to the route table of hosts `h1`, `h2`, `h7`, and `h8`:
   `h* route add -net 224.0.0.0 netmask 240.0.0.0 dev h*-eth0`
3. connect hosts `h1`, `h2`, and `h8` to the stream; for each host:
   (a) Start vlc
   (b) Click on `media -> Open Network Stream`
   (c) Fill in `udp://@224.0.0.14` as the network address and press `play`
4. Start vlc on host `h7`
5. Start streaming from this host as follows:
   (a) Click on `media -> Stream...`
   (b) Add `TestVideo.mp4` (which you downloaded before) and press stream
   (c) Press `next`, select `UDP (legacy)` as destination and press `add`
   (d) Fill in `224.0.0.14` as the address and press `next`
   (e) Click through the rest of the dialog and start streaming
6. You should now be receiving the stream on all three hosts!

Note: It might take a few seconds until you can see the actual video playback.

Solution

In our solution, we first compute the whole SPT by computing the shortest paths from the source host to each subscriber. We then iterate over each switch in the tree and install an all table that forwards packets to all its outgoing links in the tree.

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```

```
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto
from ryu.lib.packet import igmp

from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link, get_host

import networkx as nx

class Multicast_Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    LOWPRIO = 1
    MEDPRIO = 2
    HIGHPRIO = 3
    HIGHESTPRIO = 4

    def __init__(self, *args, **kwargs):
        super(Multicast_Controller, self).__init__(*args, **kwargs)

        self.network = nx.DiGraph()
        self.subscribers = {}
        self.sources = {}
        self.group_ids = {}

        self.max_group_id = 0

    def log(self, message):
        self.logger.info(message)
        return

    #This function is triggered before the topology controller flows are added
    #But late enough to be able to remove flows
    @set_ev_cls(ofp_event.EventOFPStateChange, [CONFIG_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Delete any possible currently existing flows.
        del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL, out_port=ofp.OFPP_ANY,
            out_group=ofp.OFPG_ANY, command=ofp.OFPFC_DELETE)
        dp.send_msg(del_flows)

        #Delete any possible currently exising groups
        del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
            group_id=ofp.OFPG_ALL)
        dp.send_msg(del_groups)

        #Make sure deletion is finished using a barrier before additional flows are added
        barrier_req = parser.OFPBarrierRequest(dp)
        dp.send_msg(barrier_req)

    #Switch connected
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Add default flow
        match = parser.OFPMatch()
```

```
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)

    #Topology Events
    @set_ev_cls(event.EventSwitchEnter)
    def switchEnter(self,ev):
        switch = ev.switch

        self.network.add_node(switch.dp.id, switch = switch, groups = {}, host = False)
        self.log('Added switch ' + str(switch.dp.id))

    @set_ev_cls(event.EventSwitchLeave)
    def switchLeave(self,ev):
        switch = ev.switch
        sid = switch.dp.id

        if sid in self.network:
            #NOTE: In actual applications one should include some code to properly handle switches leaving
            #and re-entering!

            self.network.remove_node(sid)
            self.log('Removed switch ' + str(sid))

    @set_ev_cls(event.EventLinkAdd)
    def linkAdd(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        self.network.add_edge(src, dst, src_port = link.src.port_no, dst_port = link.dst.port_no, live = True)
        self.log('Added link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventLinkDelete)
    def linkDelete(self,ev):
        link = ev.link
        src = link.src.dpid
        dst = link.dst.dpid

        if (src,dst) in self.network.edges():
            #NOTE: In actual applications one should include some code to properly handle link failures!

            self.network.remove_edge(src, dst)
            self.log('Removed link from ' + str(src) + ' to ' + str(dst))

    @set_ev_cls(event.EventHostAdd)
    def hostFound(self,ev):
        host = ev.host
        switch = host.port.dpid
        mac = host.mac

        self._hostFound(switch, host.port.port_no, mac)

    def _hostFound(self, switch_id, port, mac):
        if mac not in self.network:
            self.network.add_node(mac, host = True)
            self.network.add_edge(mac, switch_id, src_port = -1, dst_port = port, live = True)
            self.network.add_edge(switch_id, mac, src_port = port, dst_port = -1, live = True)
            self.log('Added host ' + mac + ' at switch ' + str(switch_id))

    def install_group(self, switch_id, dst_address, dsts, group_id):
```

```
        """Iff group group_id has not yet been installed at switch switch_id,
        install flow and group entry in switch switch_id that forwards packets to all dsts.

        Arguments:
        switch_id: id of switch to add flow to
        dst_address: IP address of the destination (i.e. the group address)
        dsts: switch or host ids to output packet to
        group_id: id of the all group associated with dst_address
        """
        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        ports = [self.network[switch_id][dst]['src_port'] for dst in dsts]

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups:
            #Add group entry
            action_lists = [[parser.OFPActionOutput(port)] for port in ports]
            buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
                for action_list in action_lists]
            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_ADD, ofp.OFPGT_ALL, group_id, buckets)
            dp.send_msg(cmd)

            groups[group_id] = set(ports)

            #Add flow entry pointing to group
            match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=dst_address)
            actions = [parser.OFPActionGroup(group_id)]
            instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
            cmd = parser.OFPFlowMod(datapath=dp, priority=self.MEDPRIO,
                match=match, instructions=instr)
            dp.send_msg(cmd)

        self.log('INSTALLED GROUP ' + str(group_id) + ' AT SWITCH ' + str(switch_id))

    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        pkt = packet.Packet(msg.data)
        eth = pkt[0]

        if eth.protocol_name != 'ethernet':
            #Ignore non-ethernet packets
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return

        self.log('Received ethernet packet')
        src = eth.src
        dst = eth.dst
        self.log('From ' + src + ' to ' + dst)

        #host detection
        if src not in self.network:
            self._hostFound(dp.id, msg.match['in_port'], src)
```

```
        if eth.ethertype == ether_types.ETH_TYPE_IP and self.isMulticast(eth.dst):
            self.process_multicast(dp, pkt)
            return

    def process_multicast(self, dp, pkt):
        eth_pkt = pkt[0]
        ip_pkt = pkt[1]

        #IGMP message
        if ip_pkt.proto == in_proto.IPPROTO_IGMP:
            igmp_pkt = pkt[2]
            self.processIGMP(eth_pkt.src, ip_pkt.src, igmp_pkt)
            return

        if ip_pkt.protocol_name != 'ipv4':
            #ignore IPv6
            return

        ip_dst = ip_pkt.dst
        ip_src = ip_pkt.src
        eth_src = eth_pkt.src

        if (ip_dst not in self.sources
            or eth_src not in self.sources[ip_dst]):
            self.add_source(dp, eth_src, ip_dst)


    def add_source(self, dp, eth_src, multicast_address):
        if multicast_address not in self.sources:
            self.sources[multicast_address] = set()

            self.max_group_id = self.max_group_id + 1
            self.group_ids[multicast_address] = self.max_group_id

        group_id = self.group_ids[multicast_address]

        self.sources[multicast_address].add(eth_src)

        self.log('Adding source ' + eth_src + ' to multicast group ' + multicast_address)

        if multicast_address not in self.subscribers:
            #no subscribers
            self.install_group(dp.id, multicast_address, [], group_id)
            return

        next_hops = {}

        for subscriber in self.subscribers[multicast_address]:
            try:
                path = nx.shortest_path(self.network, source=eth_src, target=subscriber)

                for i in range(1,len(path)-1):
                    switch = path[i]

                    if switch not in next_hops:
                        next_hops[switch] = set()

                    next_hops[switch].add(path[i+1])

            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(eth_src) + ' to ' + subscriber)
```

```
        for switch in next_hops:
            #If the group entry has already been installed, we do not need to modify anything
            #So we can safely use this method
            self.install_group(switch, multicast_address, next_hops[switch], group_id)


    def processIGMP(self, eth_src, ip_src, igmp_pkt):
        self.log('IGMP packet')

        #Only support IGMPV3 membership records
        if igmp_pkt.protocol_name != 'igmpv3_report':
            return

        records = igmp_pkt.records
        for record in records:
            if not record.srcs:
                address = record.address
                self.log('Record change for group ' + address)

                if record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE:
                    if address not in self.subscribers:
                        self.subscribers[address] = set()

                    self.subscribers[address].add(eth_src)

                    self.log('Added ' + eth_src)

                elif record.type_ == igmp.CHANGE_TO_INCLUDE_MODE:
                    if address not in self.subscribers:
                        self.subscribers[address] = set()

                    self.subscribers[address].discard(eth_src)

                    self.log('Removed ' + eth_src)


    def isMulticast(self, eth_dst):
        bit = eth_dst[1]
        return int(bit, 16) % 2 == 1
```

4.4 Exercise 3. - Dynamic Groups

Our final goal is to support dynamic multicast groups as well. Hosts can join and leave a group at any time. Thus, the controller needs to modify the multicast tree on the fly, as new subscribers are joining or leaving the group.

Extend your IGMPv3 processing code to

- Add paths from all sources to the new subscriber if a host subscribes to a group. You can do so by computing a shortest path from each source to the new subscriber, and iterating over each switch on each of these paths:
  - If the switch already forwards packets to the next switch on the path: do nothing
  - Otherwise, and if the switch already contains a group entry for this multicast group: modify this group entry (`OFPGroupMod(dp, OFPGC_MODIFY, ...)`) to add an additional action bucket.
  - Otherwise, install both a new all group entry, as well as a new flow entry pointing to this group.
- Remove the paths from all sources to a host if it decides to leave a group. You are allowed to assume the network remains static after a subscriber joins a multicast group (which means that the same NetworkX shortest path computation should result in exactly the same path again). You can do so by computing a shortest path from each source to the subscriber. Unfortunately, we can not just start modifying group entries on all the switches on these paths, as parts of this path may also be used to route packets to other subscribers. Thus, for each of these paths, we move *backward* starting from the last switch on the path:

- If the switch still forwards packets to the next hop on the path, modify the group entry to not forward packets to this port anymore. If this would result in an empty group entry, remove the group entry using `OFPGroupMod(dp, ofp.ofpOFPGC_DELETE,...)`. Deleting a group entry also removes the associated flow entry automatically.
- If you removed the group entry in the previous step (or there was no group entry), that means the previous switch on the path also does not need to forward packets to this switch anymore, so we can move to this switch and start at step 1 again.
- If the current switch still contains a group entry for the multicast group, that means the previous switch on the path still needs to forward packets to the current switch, so we do not need to modify any other switches on this path.

Test your app as follows:

1. Start Mininet with the binary tree topology of depth 3 and connect it to a Ryu controller running your app.
2. Add a multicast route to the route table of hosts `h1`, `h2`, `h7`, and `h8`:
   `h* route add -net 224.0.0.0 netmask 240.0.0.0 dev h*-eth0`
3. Start vlc on host `h7`
4. Start streaming from this host as follows:
   (a) Click on `media -> Stream...`
   (b) Add `TestVideo.mp4` (which you downloaded before) and press stream
   (c) Press `next`, select `UDP (legacy)` as destination and press `add`
   (d) Fill in `224.0.0.14` as the address and press `next`
   (e) Click through the rest of the dialog and start streaming
5. connect hosts `h1`, `h2`, and `h8` to the stream; for each host:
   (a) Start vlc
   (b) Click on `media -> Open Network Stream`
   (c) Fill in `udp://@224.0.0.14` as the network address and press `play`
6. You should now be receiving the stream on all three hosts!
7. Disconnect and reconnect from the stream a few times on all receiving hosts. If the app is working properly, you should be able to see the video again after reconnecting to the stream. In addition, when you stop the stream at one host, it should continue at all the other hosts.
8. Check if the flow and group entries are properly removed/modified and no multicast packets are being received at the host anymore after exiting vlc with `dump-flows` and Wireshark.

If you receive an influx of packets at the controller after exiting VLC, your app is probably not modifying all group entries properly, and a switch is forwarding traffic to another switch that was removed from the multicast tree. Remember that the initial switch connected to the source host will always keep receiving packets, even if the group has no subscribers. To prevent the controller from being flooded by these packets, you can either install a low-priority flow entry in this switch that drops all packets to the multicast group or add logic to your controller to ensure that the group entry in this switch is never removed.

Solution

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4
from ryu.lib.packet import in_proto
from ryu.lib.packet import igmp
```

```
from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link, get_host

import networkx as nx

class Multicast_Controller(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    LOWPRIO = 1
    MEDPRIO = 2
    HIGHPRIO = 3
    HIGHESTPRIO = 4

    def __init__(self, *args, **kwargs):
        super(Multicast_Controller, self).__init__(*args, **kwargs)

        self.network = nx.DiGraph()
        self.subscribers = {}
        self.sources = {}
        self.group_ids = {}

        self.max_group_id = 0

    def log(self, message):
        self.logger.info(message)
        return

    #This function is triggered before the topology controller flows are added
    #But late enough to be able to remove flows
    @set_ev_cls(ofp_event.EventOFPStateChange, [CONFIG_DISPATCHER])
    def state_change_handler(self, ev):
        dp = ev.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Delete any possible currently existing flows.
        del_flows = parser.OFPFlowMod(dp, table_id=ofp.OFPTT_ALL, out_port=ofp.OFPP_ANY,
            out_group=ofp.OFPG_ANY, command=ofp.OFPFC_DELETE)
        dp.send_msg(del_flows)

        #Delete any possible currently exising groups
        del_groups = parser.OFPGroupMod(datapath=dp, command=ofp.OFPGC_DELETE,
            group_id=ofp.OFPG_ALL)
        dp.send_msg(del_groups)

        #Make sure deletion is finished using a barrier before additional flows are added
        barrier_req = parser.OFPBarrierRequest(dp)
        dp.send_msg(barrier_req)

    #Switch connected
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        dp = ev.msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        #Add default flow
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=0, match=match, instructions=instr)
        dp.send_msg(cmd)
```

```
#Topology Events
@set_ev_cls(event.EventSwitchEnter)
def switchEnter(self,ev):
    switch = ev.switch

    self.network.add_node(switch.dp.id, switch = switch, groups = {}, host = False)
    self.log('Added switch ' + str(switch.dp.id))

@set_ev_cls(event.EventSwitchLeave)
def switchLeave(self,ev):
    switch = ev.switch
    sid = switch.dp.id

    if sid in self.network:
        #NOTE: In actual applications one should include some code to properly handle switches leaving
        #and re-entering!

        self.network.remove_node(sid)
        self.log('Removed switch ' + str(sid))

@set_ev_cls(event.EventLinkAdd)
def linkAdd(self,ev):
    link = ev.link
    src = link.src.dpid
    dst = link.dst.dpid

    self.network.add_edge(src, dst, src_port = link.src.port_no, dst_port = link.dst.port_no, live = True)
    self.log('Added link from ' + str(src) + ' to ' + str(dst))

@set_ev_cls(event.EventLinkDelete)
def linkDelete(self,ev):
    link = ev.link
    src = link.src.dpid
    dst = link.dst.dpid

    if (src,dst) in self.network.edges():
        #NOTE: In actual applications one should include some code to properly handle link failures!

        self.network.remove_edge(src, dst)
        self.log('Removed link from ' + str(src) + ' to ' + str(dst))

@set_ev_cls(event.EventHostAdd)
def hostFound(self,ev):
    host = ev.host
    switch = host.port.dpid
    mac = host.mac

    self._hostFound(switch, host.port.port_no, mac)

def _hostFound(self, switch_id, port, mac):
    if mac not in self.network:
        self.network.add_node(mac, host = True)
        self.network.add_edge(mac, switch_id, src_port = -1, dst_port = port, live = True)
        self.network.add_edge(switch_id, mac, src_port = port, dst_port = -1, live = True)
        self.log('Added host ' + mac + ' at switch ' + str(switch_id))

def remove_forwarding_rule(self, switch_id, dst_address, dst, group_id):
    """Remove/Modify flow and group entry in switch switch_id to stop forwarding packets to dst.

    Returns True iff no group entry for group_id remains after performing this operation
    Arguments:
    switch_id: id of switch to add flow to
```

```
        dst_address: IP address of the destination (i.e. the group address)
        dst: switch or host id to output packet to
        group_id: id of the all group associated with dst_address
        """

        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        port = self.network[switch_id][dst]['src_port']

        self.log('REMOVING FLOW FROM SWITCH ' + str(switch_id) + ' TO PORT '
                 + str(port))
        self.log('DESTINATION = ' + str(dst_address))

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups or port not in groups[group_id]:
            return True

        if len(groups[group_id]) == 1:
            #Group entry only forwards to port
            #So we should remove the whole group

            del groups[group_id]

            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_DELETE, ofp.OFPGT_ALL, group_id, [])
            dp.send_msg(cmd)

            return True

        groups[group_id].discard(port)

        action_lists = [[parser.OFPActionOutput(port)] for port in groups[group_id]]
        buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
            for action_list in action_lists]
        cmd = parser.OFPGroupMod(dp, ofp.OFPGC_MODIFY, ofp.OFPGT_ALL, group_id, buckets)
        dp.send_msg(cmd)

        return False

    def install_forwarding_rule(self, switch_id, dst_address, dst, group_id):
        """Install/Modify flow and group entry in switch switch_id to forward packets to dst.

        Arguments:
        switch_id: id of switch to add flow to
        dst_address: IP address of the destination (i.e. the group address)
        dst: switch or host id to output packet to
        group_id: id of the all group associated with dst_address
        """

        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        port = self.network[switch_id][dst]['src_port']

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups:
            self.install_group(switch_id, dst_address, [dst], group_id)

        elif port not in groups[group_id]:
            groups[group_id].add(port)

            action_lists = [[parser.OFPActionOutput(port)] for port in groups[group_id]]
```

```
            buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
                for action_list in action_lists]
            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_MODIFY, ofp.OFPGT_ALL, group_id, buckets)
            dp.send_msg(cmd)

        self.log('ADDED FLOW FROM SWITCH ' + str(switch_id) + ' TO PORT '
                    + str(port))
        self.log('DESTINATION = ' + str(dst_address))

    def install_group(self, switch_id, dst_address, dsts, group_id):
        """Iff group group_id has not yet been installed at switch switch_id,
        install flow and group entry in switch switch_id that forwards packets to all dsts.

        Arguments:
        switch_id: id of switch to add flow to
        dst_address: IP address of the destination (i.e. the group address)
        dsts: switch or host ids to output packet to
        group_id: id of the all group associated with dst_address
        """
        dp = self.network.nodes[switch_id]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        ports = [self.network[switch_id][dst]['src_port'] for dst in dsts]

        groups = self.network.nodes[switch_id]['groups']
        if group_id not in groups:
            #Add group entry
            action_lists = [[parser.OFPActionOutput(port)] for port in ports]
            buckets = [parser.OFPBucket(0, ofp.OFPP_ANY, ofp.OFPG_ANY, action_list)
                for action_list in action_lists]
            cmd = parser.OFPGroupMod(dp, ofp.OFPGC_ADD, ofp.OFPGT_ALL, group_id, buckets)
            dp.send_msg(cmd)

            groups[group_id] = set(ports)

            #Add flow entry pointing to group
            match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=dst_address)
            actions = [parser.OFPActionGroup(group_id)]
            instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
            cmd = parser.OFPFlowMod(datapath=dp, priority=self.MEDPRIO,
                match=match, instructions=instr)
            dp.send_msg(cmd)

        self.log('INSTALLED GROUP' + str(group_id) + ' AT SWITCH ' + str(switch_id))

    #Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        pkt = packet.Packet(msg.data)
        eth = pkt[0]

        if eth.protocol_name != 'ethernet':
            #Ignore non-ethernet packets
            return

        #Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return
```

```
        self.log('Received ethernet packet')
        src = eth.src
        dst = eth.dst
        self.log('From ' + src + ' to ' + dst)

        #host detection
        if src not in self.network:
            self._hostFound(dp.id, msg.match['in_port'], src)

        if eth.ethertype == ether_types.ETH_TYPE_IP and self.isMulticast(eth.dst):
            self.process_multicast(dp, pkt)
            return

    def process_multicast(self, dp, pkt):
        eth_pkt = pkt[0]
        ip_pkt = pkt[1]

        #IGMP message
        if ip_pkt.proto == in_proto.IPPROTO_IGMP:
            igmp_pkt = pkt[2]
            self.processIGMP(eth_pkt.src, ip_pkt.src, igmp_pkt)
            return

        if ip_pkt.protocol_name != 'ipv4':
            #ignore IPv6
            return

        ip_dst = ip_pkt.dst
        ip_src = ip_pkt.src
        eth_src = eth_pkt.src

        if (ip_dst not in self.sources
            or eth_src not in self.sources[ip_dst]):
            self.add_source(dp, eth_src, ip_dst)


    def add_source(self, dp, eth_src, multicast_address):
        if multicast_address not in self.sources:
            self.sources[multicast_address] = set()

            self.max_group_id = self.max_group_id + 1
            self.group_ids[multicast_address] = self.max_group_id

        group_id = self.group_ids[multicast_address]

        self.sources[multicast_address].add(eth_src)

        self.log('Adding source ' + eth_src + ' to multicast group ' + multicast_address)

        #With low priority, drop all packets of this multicast group
        #This prevents the controller from being flooded by packets once all subscribers leave the group
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch(eth_type = 0x800, ipv4_dst=multicast_address)
        actions = []
        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=self.LOWPRIO,
            match=match, instructions=instr)
        dp.send_msg(cmd)

        if multicast_address not in self.subscribers:
```

```
            #no subscribers
            return

        next_hops = {}

        for subscriber in self.subscribers[multicast_address]:
            try:
                path = nx.shortest_path(self.network, source=eth_src, target=subscriber)

                for i in range(1,len(path)-1):
                    switch = path[i]

                    if switch not in next_hops:
                        next_hops[switch] = set()

                    next_hops[switch].add(path[i+1])

            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(eth_src) + ' to ' + subscriber)

        for switch in next_hops:
            #If the group entry has already been installed, we do not need to modify anything
            #So we can safely use this method
            self.install_group(switch, multicast_address, next_hops[switch], group_id)


    def add_subscriber(self, eth_address, multicast_address):
        if multicast_address not in self.subscribers:
            self.subscribers[multicast_address] = set()

        self.subscribers[multicast_address].add(eth_address)

        self.log('Adding ' + eth_address)

        if multicast_address not in self.sources:
            #no sources
            return

        group_id = self.group_ids[multicast_address]

        for source in self.sources[multicast_address]:
            try:
                path = nx.shortest_path(self.network, source=source, target=eth_address)

                for i in range(1,len(path)-1):
                    switch = path[i]
                    self.install_forwarding_rule(switch, multicast_address, path[i+1], group_id)
            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(source) + ' to ' + eth_address)

    def remove_subscriber(self, eth_address, multicast_address):
        self.log('Removing ' + eth_address)

        if multicast_address not in self.subscribers or eth_address not in self.subscribers[multicast_address]:
            return

        self.subscribers[multicast_address].discard(eth_address)

        if multicast_address not in self.sources:
            #no sources
            return
```

```
        group_id = self.group_ids[multicast_address]

        for source in self.sources[multicast_address]:
            try:
                #this is exactly the path we also used to route traffic to the subscriber earlier
                #except if the network has changed in the meantime (but we assume the network is static)
                path = nx.shortest_path(self.network, source=source, target=eth_address)


                for i in range(1,len(path)-1):
                    switch = path[-i-1]

                    if not self.remove_forwarding_rule(switch, multicast_address, path[-i], group_id):
                        #If the switch still routes other traffic for this group,
                        #then we should not remove the rest of the path
                        break;

            except(nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from ' + str(source) + ' to ' + eth_address)

    def processIGMP(self, eth_src, ip_src, igmp_pkt):
        self.log('IGMP packet')

        #Only support IGMPV3 membership records
        if igmp_pkt.protocol_name != 'igmpv3_report':
            return

        records = igmp_pkt.records
        for record in records:
            if not record.srcs:
                address = record.address
                self.log('Record change for group ' + address)

                if record.type_ == igmp.CHANGE_TO_EXCLUDE_MODE:
                    self.add_subscriber(eth_src, address)

                elif record.type_ == igmp.CHANGE_TO_INCLUDE_MODE:
                    self.remove_subscriber(eth_src, address)


    def isMulticast(self, eth_dst):
        bit = eth_dst[1]
        return int(bit, 16) % 2 == 1
```