LAMPIRAN

```
from ryu.base import app_manager
from ryu.controller import mac_to_port
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.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import arp
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import ipv6
from ryu.lib.packet import ether_types
from ryu.lib import mac, ip
from ryu.topology.api import get_switch, get_link
from ryu.app.wsgi import ControllerBase
from ryu.topology import event

from collections import defaultdict
from operator import itemgetter

import os
import random
import time

# Cisco Reference bandwidth = 1 Gbps
REFERENCE_BW = 10000000

DEFAULT_BW = 10000000

MAX_PATHS = float('Inf')

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

    def __init__(self, *args, **kwargs):
        super(ProjectController, self).__init__(*args,
**kwargs)
        self.mac_to_port = {}
        self.topology_api_app = self
        self.datapath_list = {}
        self.arp_table = {}
        self.switches = []
        self.hosts = {}
        self.multipath_group_ids = {}
        self.group_ids = []
        self.adjacency = defaultdict(dict)
        self.bandwidths = defaultdict(lambda:
defaultdict(lambda: DEFAULT_BW))
```

```
    def get_paths(self, src, dst):
        '''
        Get all paths from src to dst using DFS algorithm
        '''
        if src == dst:
            # host target is on the same switch
            return [[src]]
        paths = []
        stack = [(src, [src])]
        while stack:
            (node, path) = stack.pop()
            for next in set(self.adjacency[node].keys()) -
set(path):
                if next is dst:
                    paths.append(path + [next])
                else:
                    stack.append((next, path + [next]))
        print "Available paths from ", src, " to ", dst, "
: ", paths
        return paths

    def get_link_cost(self, s1, s2):
        '''
        Get the link cost between two switches
        '''
        e1 = self.adjacency[s1][s2]
        e2 = self.adjacency[s2][s1]
        bl = min(self.bandwidths[s1][e1],
self.bandwidths[s2][e2])
        ew = REFERENCE_BW/bl
        return ew

    def get_path_cost(self, path):
        '''
        Get the path cost
        '''
        cost = 0
        for i in range(len(path) - 1):
            cost += self.get_link_cost(path[i], path[i+1])
        return cost

    def get_optimal_paths(self, src, dst):
        '''
        Get the n-most optimal paths according to
MAX_PATHS
        '''
        paths = self.get_paths(src, dst)
        paths_count = len(paths) if len(
            paths) < MAX_PATHS else MAX_PATHS
        return sorted(paths, key=lambda x:
self.get_path_cost(x))[0:(paths_count)]
```

```
    def add_ports_to_paths(self, paths, first_port,
last_port):
        '''
        Add the ports that connects the switches for all
paths
        '''
        paths_p = []
        for path in paths:
            p = {}
            in_port = first_port
            for s1, s2 in zip(path[:-1], path[1:]):
                out_port = self.adjacency[s1][s2]
                p[s1] = (in_port, out_port)
                in_port = self.adjacency[s2][s1]
            p[path[-1]] = (in_port, last_port)
            paths_p.append(p)
        return paths_p

    def generate_openflow_gid(self):
        '''
        Returns a random OpenFlow group id
        '''
        n = random.randint(0, 2**32)
        while n in self.group_ids:
            n = random.randint(0, 2**32)
        return n


    def install_paths(self, src, first_port, dst,
last_port, ip_src, ip_dst):
        computation_start = time.time()
        paths = self.get_optimal_paths(src, dst)
        pw = []
        for path in paths:
            pw.append(self.get_path_cost(path))
            print path, "cost = ", pw[len(pw) - 1]
        sum_of_pw = sum(pw)
        paths_with_ports = self.add_ports_to_paths(paths,
first_port, last_port)
        switches_in_paths = set().union(*paths)

        for node in switches_in_paths:

            dp = self.datapath_list[node]
            ofp = dp.ofproto
            ofp_parser = dp.ofproto_parser

            ports = defaultdict(list)
            actions = []
            i = 0

            for path in paths_with_ports:
                if node in path:
```

```
                    in_port = path[node][0]
                    out_port = path[node][1]
                    if out_port not in ports[in_port]:
                        ports[in_port].append(out_port)
                i += 1

            for in_port in ports:

                match_ip = ofp_parser.OFPMatch(
                    eth_type=0x0800,
                    ipv4_src=ip_src,
                    ipv4_dst=ip_dst
                )
                match_arp = ofp_parser.OFPMatch(
                    eth_type=0x0806,
                    arp_spa=ip_src,
                    arp_tpa=ip_dst
                )

                out_ports = ports[in_port]
                print out_ports

                if len(out_ports) > 1:
                    group_id = None
                    group_new = False

                  if (node, src, dst) not in
self.multipath_group_ids:
                    group_new = True
                    self.multipath_group_ids[
                      node, src, dst] =
self.generate_openflow_gid()
                    group_id =
self.multipath_group_ids[node, src, dst]

                    buckets = []
                    # print "node at ",node," out ports :
",out_ports
                    for port in out_ports:
                        bucket_weight = 0
                        bucket_action =
[ofp_parser.OFPActionOutput(port)]
                        buckets.append(
                            ofp_parser.OFPBucket(
                                weight=bucket_weight,
                                watch_port=port,
                                watch_group=ofp.OFPG_ANY,
                                actions=bucket_action
                            )
                        )

                    if group_new:
                        req = ofp_parser.OFPGroupMod(
```

```
                                dp, ofp.OFPGC_ADD,
ofp.OFPGT_FF, group_id,
                                buckets
                            )
                            dp.send_msg(req)
                    else:
                    req = ofp_parser.OFPGroupMod(
                          dp, ofp.OFPGC_MODIFY, ofp.OFPGT_FF,
                            group_id, buckets)
                            dp.send_msg(req)
              actions = [ofp_parser.OFPActionGroup(group_id)]
                   self.add_flow(dp, 32768, match_ip, actions)
                   self.add_flow(dp, 1, match_arp, actions)
              elif len(out_ports) == 1:
         actions = [ofp_parser.OFPActionOutput(out_ports[0])]
                 self.add_flow(dp, 32768, match_ip, actions)
                 self.add_flow(dp, 1, match_arp, actions)
            print "Path installation finished in ",
time.time() - computation_start
            return paths_with_ports[0][src][1]

    def add_flow(self, datapath, priority, match, actions,
buffer_id=None):
        #print "Adding flow ", match, actions
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                                actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath,
buffer_id=buffer_id,
                                    priority=priority,
match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath,
priority=priority,
                                    match=match,
instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
CONFIG_DISPATCHER)
    def _switch_features_handler(self, ev):
        print "switch_features_handler is called"
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
```

```
        actions =
[parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,

ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    @set_ev_cls(ofp_event.EventOFPPortDescStatsReply,
MAIN_DISPATCHER) #
    def port_desc_stats_reply_handler(self, ev):
        switch = ev.msg.datapath
        for p in ev.msg.body:
            self.bandwidths[switch.id][p.port_no] =
p.curr_speed

    @set_ev_cls(ofp_event.EventOFPPacketIn,
MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protokol(ethernet.ethernet)
        arp_pkt = pkt.get_protokol(arp.arp)

        # avoid broadcast from LLDP
        if eth.ethertype == 35020:
            return

        if pkt.get_protokol(ipv6.ipv6):  .
            match =
parser.OFPMatch(eth_type=eth.ethertype)
            actions = []
            self.add_flow(datapath, 1, match, actions)
            return None

        dst = eth.dst
        src = eth.src
        dpid = datapath.id

        if src not in self.hosts:
            self.hosts[src] = (dpid, in_port)

        out_port = ofproto.OFPP_FLOOD

        if arp_pkt:
             #print dpid, pkt
            src_ip = arp_pkt.src_ip
            dst_ip = arp_pkt.dst_ip
            if arp_pkt.opcode == arp.ARP_REPLY:
                self.arp_table[src_ip] = src
```

```
                h1 = self.hosts[src]
                h2 = self.hosts[dst]
                out_port = self.install_paths(h1[0],
h1[1], h2[0], h2[1], src_ip, dst_ip)
                self.install_paths(h2[0], h2[1], h1[0],
h1[1], dst_ip, src_ip) # reverse
            elif arp_pkt.opcode == arp.ARP_REQUEST:
                if dst_ip in self.arp_table:
                    self.arp_table[src_ip] = src
                    dst_mac = self.arp_table[dst_ip]
                    h1 = self.hosts[src]
                    h2 = self.hosts[dst_mac]
                    out_port = self.install_paths(h1[0],
h1[1], h2[0], h2[1], src_ip, dst_ip)
                    self.install_paths(h2[0], h2[1],
h1[0], h1[1], dst_ip, src_ip) # reverse

        print pkt

        actions = [parser.OFPActionOutput(out_port)]

        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(
            datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port,
            actions=actions, data=data)
        datapath.send_msg(out)

    @set_ev_cls(event.EventSwitchEnter)
    def switch_enter_handler(self, event):
        switch = event.switch.dp
        ofp_parser = switch.ofproto_parser

        if switch.id not in self.switches:
            self.switches.append(switch.id)
            self.datapath_list[switch.id] = switch

            # Request port/link descriptions, useful for
obtaining bandwidth
            req =
ofp_parser.OFPPortDescStatsRequest(switch)
            switch.send_msg(req)

    @set_ev_cls(event.EventSwitchLeave, MAIN_DISPATCHER)
    def switch_leave_handler(self, event):
        print event
        switch = event.switch.dp.id
        if switch in self.switches:
            del self.switches[switch]
            del self.datapath_list[switch]
```

```
            del self.adjacency[switch]

    @set_ev_cls(event.EventLinkAdd, MAIN_DISPATCHER)
    def link_add_handler(self, event):
        s1 = event.link.src
        s2 = event.link.dst
        self.adjacency[s1.dpid][s2.dpid] = s1.port_no
        self.adjacency[s2.dpid][s1.dpid] = s2.port_no

    @set_ev_cls(event.EventLinkDelete, MAIN_DISPATCHER)
    def link_delete_handler(self, event):
return
```