75.43 Introducción a los sistemas distribuidos

Entrega Trabajo Práctico 3: Enlace

Integrantes:

Alumno	padron
Azcona, Gabriela Mariel	95363
Avigliano, Patricio Andres	98861
Blanco, Sebastian Ezequiel	98539

Fecha de Entrega: 23/10/2018

GitHub: https://github.com/BlancoSebastianEzequiel/Datacenter

Índice

1. Introducción teórica

- **SDN:**
  Software Defined Networking es un paradigma que puede considerarse reciente en el cual los dispositivos intermediarios encargados de conmutar paquetes son configurados por una entidad controladora por medio de software. Decimos dispositivos intermediarios porque este nuevo paradigma permite una configuración tan flexible que se pierde la distinción entre switches, routers, NATs; ahora cada dispositivo se configura según las necesidades particulares de la red en la que habita.

- **OpenFlow:**
  Es la herramienta que se utiliza para implementar esta nueva tecnología, mejor dicho es el protocolo por el cual se configuran los dispositivos intermediarios. La idea principal es reemplazar las tablas de ruteo de los routers y las tablas de direcciones Mac en los switches por tablas de flujo. Entonces un dispositivo OpenFlow decide qué hacer con los paquetes que le llegan en base a la tabla de flujo, por otro lado se configuran las políticas y el comportamiento que debe adoptar mediante el protocolo OpenFlow.

- **Control y Forwarding path:**
  Un dispositivo de internet por definición debe funcionar con la mayor velocidad posible, por ello su funcionamiento está implementado por hardware y el costo de implementarlo exclusivamente por software en cuanto a velocidad sería muy elevado. Es por eso que los dispositivos OpenFlow se dividen en 2 planos: el plano de datos o forwarding (hardware) y el plano de control (software), este último es el que se comunica por medio de OpenFlow con la entidad controladora que indicará cómo deberá ser administrado el dispositivo. El plano de datos hará lo que sea necesario con cada paquete según la tabla de flujos mientras que el plano de control gestionará las decisiones a tomar sobre la construcción de la tabla, modificación de algún parámetro de la cabecera (por ejemplo al implementar Network Address Translation) y políticas de seguridad, entre otras funcionalidades.

- **Concepto de flujo:**
  No existe una definición per se de lo que es un flujo pero lo entendemos como el conjunto de paquetes que esperamos que llegue de un mismo origen a un mismo destino (por destino y origen nos referimos a nivel enlace/red/transporte) con similar latencia y por el mismo camino. Un ejemplo podría ser la respuesta de un http get, todos los paquetes de la respuesta provienen del mismo origen, van hacia el mismo destino y se espera que lleguen medianamente uno detrás del otro (suponiendo no haya pérdidas). Para lo que es un dispositivo OpenFlow un flujo se define como la 10-tupla formada por (PortIn, VLANID, srcEth, dstEth, typeEth, srcIP, dstIP, protoIP, srcport, dstport) y es sobre estos campos que se definen las entradas en la tabla

de flujos, luego podrán utilizarse los campos que sean necesarios según las políticas adoptadas por el plano de control.

- **IP blackholing:**
  Es la decisión que se toma de descartar paquetes provenientes de una determinada dirección IP al detectar un ataque, los dispositivos OpenFlow permiten introducir políticas sobre lo que debe ser considerado como un ataque y en qué casos hacer IP blackholing de manera flexible y que se adapte a la sensibilidad de la red en la que está funcionando.

- **Firewall:**
  Es un mecanismo de seguridad cuya función es proteger la red interna frente amenazas de redes no confiables, para ello puede filtrar o redireccionar los paquetes que se consideran como no permitidos según ciertas reglas de seguridad. Por ejemplo si no queremos contestar paquetes ICMP podemos configurar para que todos los paquetes de ese protocolo sean descartados. Devuelta lo que permite OpenFlow es adaptar el firewall del dispositivo según las necesidades particulares de la red.

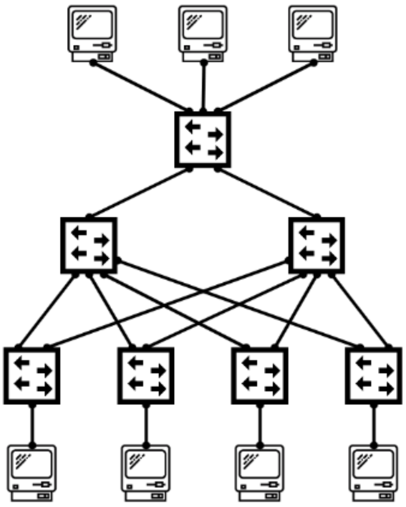

Figura 1: Ejemplo de como seria la topologia con un arbol de altura 3.

2. Objetivo

La idea del trabajo es familiarizarse con la tecnología de las SDNs y el protocolo OpenFlow, junto con las diversas problemáticas que permiten enfrentar; como objetivo secundario veremos una arquitectura de datacenters. Para ello simularemos mediante Mininet la estructura de un datacenter pequeño conectado bajo la topología Fat-Tree y configuraremos los switches mediante OpenFlow

3. Desarrollo

1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común?

2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?

3. ¿Se pueden reemplazar todos los routers de la Intenet por Swithces OpenFlow?
   Piense en el escenario interASes para elaborar su respuesta

4. Pruebas realizadas

5. Conclusiones

6. Anexos (Código)

```
from mininet.topo import Topo


class Topology(Topo):
    def __init__(self, number_of_levels=3, number_of_clients=3):
        """
        :type number_of_levels: int
        """
        Topo.__init__(self)
        self.level_links = {}
        self.sw_num = 1
        self.h_num = 1
        self.number_of_levels = number_of_levels
        self.number_of_clients = number_of_clients
        self.add_clients()
        self.add_switches_and_links()
        self.add_content_providers()

    def add_clients(self):
        self.level_links[0] = []
        for i in range(0, self.number_of_clients):
            self.level_links[0].append(self.addHost('h%s' % self.h_num))
            self.h_num += 1

    def add_switches_and_links(self):
        for level in range(0, self.number_of_levels):
            next_level = level + 1
            number_of_switches_in_level = 2 ** level
            self.level_links[next_level] = []
            for i in range(0, number_of_switches_in_level):
                sw = self.addSwitch('s%s' % self.sw_num)
                self.level_links[next_level].append(sw)
                self.sw_num += 1
                for device in self.level_links[level]:
                    self.addLink(sw, device)

    def add_content_providers(self):
        for sw in self.level_links[self.number_of_levels]:
            self.addLink(sw, self.addHost('h%s' % self.h_num))
            self.h_num += 1


topos = {
    'mytopo': (lambda levels=3, clients=3: Topology(levels, clients))
}
```

Listing 1: Topology

```
from pox.core import core
```

```
import pox.openflow.libopenflow_01 as of
from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST
from pox.lib.util import dpidToStr
from pox.lib.packet.packet_utils import _ethtype_to_str
from pox.host_tracker.host_tracker import host_tracker
import pox.lib.packet as pkt
from pox.lib.revent import *
from ecmp_table import ECMPTable

log = core.getLogger()


class Controller(object):

    def __init__(self):
        core.openflow.addListeners(self)

        def startup():
            core.openflow.addListeners(self, priority=0)
            core.openflow_discovery.addListeners(self)

        core.call_when_ready(startup, ('openflow', 'openflow_discovery'))

        self.event = None
        self.dpid = None
        self.in_port = None
        self.packet = None
        self.dst_dpid = None
        self.out_port = None
        self.table = ECMPTable()
        self.eth_packet = None
        self.ip_packet = None
        self.arp_packet = None
        self.icmp_packet = None
        self.tcp_packet = None
        self.udp_packet = None
        self.net_packet = None
        self.protocol_packet = None
        self.protocol = None
        self.arp_table = {}
        self.is_ip = True
        self.adjacency = {}
        self.host_tracker = host_tracker()
        log.info("controller ready")

    def add_adjacency(self, dpid1, port1, dpid2, port2):
        if dpid1 not in self.adjacency:
            self.adjacency[dpid1] = {}
        self.adjacency[dpid1][port1] = {
                "dpid": dpid2,
                "port": port2
            }

    def remove_adjacency(self, dpid, port):
        if dpid not in self.adjacency:
            return
        if port not in self.adjacency[dpid][port]:
            return
        del self.adjacency[dpid][port]

    def _handle_LinkEvent(self, event):
        log.info("--------------------------------------------------------")
        link = event.link
        if event.added:
            self.add_adjacency(link.dpid1, link.port1, link.dpid2, link.port2)
            self.add_adjacency(link.dpid2, link.port2, link.dpid1, link.port1)
        elif event.removed:
            self.remove_adjacency(link.dpid1, link.port1)
            self.remove_adjacency(link.dpid2, link.port2)
        log.info('link added is %s' % event.added)
        log.info('link removed is %s' % event.removed)
        log.info('switch1 %d' % link.dpid1)
        log.info('port1 %d' % link.port1)
        log.info('switch2 %d' % link.dpid2)
        log.info('port2 %d' % link.port2)
        log.info("--------------------------------------------------------")

    def _handle_ConnectionUp(self, event):
        log.debug("Connection %s" % (event.connection,))

        msg = of.ofp_flow_mod()
        msg.match.dl_dst = ETHER_BROADCAST
        msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
        event.connection.send(msg)

        msg = of.ofp_flow_mod()
        msg.match.dl_type = pkt.ethernet.IPV6_TYPE
        event.connection.send(msg)

        msg = of.ofp_flow_mod()
        msg.match.dl_type = pkt.ethernet.ARP_TYPE
        msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
        event.connection.send(msg)

    @staticmethod
    def print_msg(msg):
        print "++++++++++++++++++++++++++++++++++++++++++++++++"
        print msg
        print "++++++++++++++++++++++++++++++++++++++++++++++++"

    def fill_arp_table(self):
        entry = self.host_tracker.getMacEntry(self.addr_dst)
        if entry is None:
            log.info("HOST TRACKER COULD NOT FIND ENTRY DST")
            return
        self.arp_table[self.addr_dst] = {
            "dpid": entry.dpid,
            "port": entry.port
        }

    def print_adjacents(self):
        msg = ""
        for dpid in self.adjacency:
            msg += "dpid: %s: [" % dpid
            for port in self.adjacency[dpid]:
                msg += "%s, " % self.adjacency[dpid][port]["dpid"]
            msg += "]"
            log.info(msg)
            msg = ""

    def has_discovered_the_entire_topology(self):
        if len(self.adjacency.keys()) != 7:
            return False
        for dpid in self.adjacency:
            size = len(self.adjacency[dpid])
            if dpid in [4, 5, 6, 7] and size < 2:
                return False
            if dpid in [2, 3] and size < 4:
                return False
            if dpid == 1 and size < 1:
                return False
        return True

    def _handle_PacketIn(self, event):
        self.host_tracker._handle_PacketIn(event)
        if not self.has_discovered_the_entire_topology():
            log.info("Please wait... learning the topology")
            return
        self.event = event
        self.dpid = event.connection.dpid
        log.info("------------------------------------------------------------")
        self.print_adjacents()
        log.info("SWITCH %s" % self.dpid)
        self.in_port = event.port
        self.packet = event.parsed
        log.info("ports: %s" % event.connection.ports)
        log.info("ports: %s" % event.connection.ports)
        log.info("in port: %s" % self.in_port)
        if not self.packet.parsed:
            log.warning("%i %i ignoring unparsed packet" %
                        (self.dpid, self.in_port))
            return
        log.info("HOST SRC %s" % self.packet.src)
        log.info("HOST DST: %s" % self.packet.dst)
        self.eth_packet = self.packet.find(pkt.ethernet)
        self.addr_dst = self.packet.dst
        self.fill_arp_table()
        self.ip_packet = self.packet.find(pkt.ipv4)
        self.arp_packet = self.packet.find(pkt.arp)
        self.icmp_packet = self.packet.find(pkt.icmp)
        self.tcp_packet = self.packet.find(pkt.tcp)
        self.udp_packet = self.packet.find(pkt.udp)

        if not self.validate_protocols():

            return
        if not self.validate_net_packets():
            return

        if self.addr_dst not in self.arp_table:
            log.warning("Could not find dst")
            return self.flood()

        entry = self.arp_table[self.addr_dst]
        self.dst_dpid = entry["dpid"]
        if self.dpid == self.dst_dpid:
            log.info("Current switch is destination")
            self.out_port = entry["port"]
        else:
            if self.packet.dst.is_multicast:
                self.print_msg("MULTICAST")
                return self.flood()
            log.info("Finding minimun paths from %s to %s"
                     % (self.dpid, self.dst_dpid))
            minimun_paths = self.get_minimun_paths()
            log.info("finding out port")
            self.out_port = self.get_out_port(minimun_paths)
            if self.out_port is None:
                log.info("Could not find out port")
                return
        log.info("out port: %s" % self.out_port)
        log.info("Updating flow table")
        self.update_table()
        log.info("Sending packet")
        self.send_packet()

    def flood(self):
        log.info("FLOODING PACKET")
        msg = of.ofp_packet_out()
        msg.buffer_id = self.event.ofp.buffer_id
        msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
        msg.data = self.event.ofp
        msg.in_port = self.in_port
        self.event.connection.send(msg)

    def validate_protocols(self):
        if self.udp_packet is not None:
            log.info("UDP packet!")
            self.protocol = "UDP"
            self.protocol_packet = self.udp_packet
            return True
        elif self.tcp_packet is not None:
            log.info("TCP packet!")
            self.protocol = "TCP"
            self.protocol_packet = self.tcp_packet
            return True
        elif self.icmp_packet is not None:
            log.info("ICMP packet!")
            self.protocol = "ICMP"
            self.protocol_packet = self.icmp_packet
            return True
        else:
            log.warning("icmp, tcp and udp packets are None!")
            return False

    def validate_net_packets(self):
        if _ethtype_to_str[self.packet.type] == "IPV6":
            log.warning("DROP IPV6 packet")
            return False
        if self.eth_packet is None:
            log.warning("ETHERNET packet is None!")
            return False
        if self.ip_packet is not None:
            log.info("IP packet!")
            self.is_ip = True
            self.net_packet = self.ip_packet
        elif self.arp_packet is not None:
            log.info("ARP packet!")
            self.is_ip = False
            self.net_packet = self.arp_packet
        else:
            log.warning("ARP and TCP packets are None!")
            return False
        return True

    def match_protocol_packets(self, msg):
        if self.is_ip:
            msg.match.nw_src = self.net_packet.srcip
            msg.match.nw_dst = self.net_packet.dstip
            msg.match.nw_proto = self.net_packet.protocol
            return msg
        msg.match.nw_src = self.net_packet.protosrc
        msg.match.nw_dst = self.net_packet.protodst
        msg.match.nw_proto = self.net_packet.prototype
        return msg

    def match_packet(self, msg):
        if not self.is_ip:
            return msg
        msg.match.nw_src = self.ip_packet.srcip
        msg.match.nw_dst = self.ip_packet.dstip
        msg.match.nw_proto = self.ip_packet.protocol
        return msg

    def update_table(self):
        msg = of.ofp_flow_mod()
        msg.match.dl_type = self.eth_packet.type
        msg = self.match_packet(msg)
        msg.buffer_id = self.event.ofp.buffer_id
        if self.protocol != "ICMP":
            msg.match.tp_src = self.protocol_packet.srcport
            msg.match.tp_dst = self.protocol_packet.dstport
        msg.actions.append(of.ofp_action_output(port=self.out_port))
        self.event.connection.send(msg)
        self.balance_of_charges()

    def get_minimun_paths(self):
        adjacents = self.get_adjacents(self.dpid)
        if not adjacents:
            log.warning("NO ADJACENTS FOUND")
            return []
        paths = [[neighbour] for neighbour in adjacents]
        while not self.has_found_a_path(paths, self.dst_dpid):
            last_paths = paths[:]
            for path in last_paths:
                adjacents = self.get_adjacents(path[-1]["dpid"])
                for an_adjacent in adjacents:
                    if an_adjacent["dpid"] != self.dpid:
                        if not self.node_belongs_path(an_adjacent, path):
                            paths.append(path + [an_adjacent])
        return self.filter_paths(paths, self.dst_dpid)

    def node_belongs_path(self, node, path):
        dpid = node["dpid"]
        for a_node in path:
            if a_node["dpid"] == dpid:
                return True
        return False

    def get_out_port(self, paths_to_dst):
        if len(paths_to_dst) == 0:
            return None
        ports = self.get_all_ports(paths_to_dst)
        data = (
            ports,
            self.dpid,
            self.dst_dpid,
            self.protocol,
            self.packet.src,
            self.packet.dst
        )
        return self.table.get_port_applying_ecmp(data)

    def balance_of_charges(self):
        log.info("saving (%s, %s, %s): %s" %
                 (self.dpid, self.dst_dpid, self.protocol, self.out_port))
        data = (
            self.dpid,
            self.dst_dpid,
            self.protocol,
            self.packet.src,
            self.packet.dst,
            self.out_port
        )
        self.table.save_port(data)

    @staticmethod
    def get_all_ports(paths_to_dst):
        return [a_path[0]["port"] for a_path in paths_to_dst]

    @staticmethod
    def filter_paths(paths, dpid):
        paths_to_dst = []
        for path in paths:
            if path[-1]["dpid"] != dpid:
                continue
            paths_to_dst.append(path)
        return paths_to_dst

    def has_found_a_path(self, paths, dpid):
        for path in paths:
            if path[-1]["dpid"] == dpid:
                log.info("FOUND A PATH!")
                return True
        return False

    def get_adjacents(self, dpid):
        adjacents = []
        if dpid not in self.adjacency:
            return adjacents
        for port in self.adjacency[dpid]:
            adjacents.append({
                "dpid": self.adjacency[dpid][port]["dpid"],
                "port": port
            })
        return adjacents

    def filter_repeated(self, adjacents):
        filtered = []
        belongs = False
        for an_adjacent in adjacents:
            for final_adjacent in filtered:
                dpid_1 = final_adjacent["dpid"]
                dpid_2 = an_adjacent["dpid"]
                port_1 = final_adjacent["port"]
                port_2 = an_adjacent["port"]
                if dpid_1 == dpid_2 and port_1 == port_2:
                    belongs = True
                    break
            if not belongs:
                filtered.append(an_adjacent)
                belongs = False
        return filtered

    def send_packet(self):
        msg = of.ofp_packet_out()
        msg.actions.append(of.ofp_action_output(port=self.out_port))
        msg.data = self.event.ofp
        msg.buffer_id = self.event.ofp.buffer_id
        msg.in_port = self.in_port
        self.event.connection.send(msg)
```

Listing 2: Controller

```
import pox.lib.packet as pkt
from pox.core import core
import pox.openflow.libopenflow_01 as of
from time import time
from pox.lib.recoco import Timer
from pox.lib.revent import *

UDP_PROTOCOL = pkt.ipv4.UDP_PROTOCOL
IP_TYPE = pkt.ethernet.IP_TYPE
log = core.getLogger()


class Firewall(EventMixin):
    def __init__(self):
        self.MAX_UDP_PACKETS = 100
        self.MAX_UDP_TIME = 100
        self.last_udp_flow_packets = {}
        self.total_udp_flow_packets = {}
        self.blocked_udp_packets = {}
        self.dst_ip = None
        self.dpid = None
        core.openflow.addListenerByName(
            "FlowStatsReceived",
            self._handle_flowstats_received
        )
        Timer(5, self.request_for_switch_statistics, recurring=True)
        log.info("firewall ready")

    @staticmethod
    def print_msg(msg):
        print "++++++++++++++++++++++++++++++++++++++++++"
        print msg
        print "++++++++++++++++++++++++++++++++++++++++++"

    def request_for_switch_statistics(self):
        for connection in core.openflow.connections:
            body = of.ofp_flow_stats_request()
            connection.send(of.ofp_stats_request(body=body))

    def _handle_flowstats_received(self, event):
        log.info("handle denial of service")
        self.dpid = event.connection.dpid
        self.total_udp_flow_packets = {}
        for flow in event.stats:
            self.dst_ip = flow.match.nw_dst
            if self.dst_ip is None:
                log.info("DST IP IS NONE. COULD NOT HANDLE DoS")
                continue
            if not self.get_udp_flow(flow):
                continue
            self.evaluate_blocking()
            current = self.total_udp_flow_packets[self.dst_ip]
            self.last_udp_flow_packets[self.dpid] = {}
            self.last_udp_flow_packets[self.dpid][self.dst_ip] = current

    def get_udp_flow(self, flow):
        if self.dst_ip is None or flow.match.nw_proto != UDP_PROTOCOL:
            return False
        if self.dst_ip not in self.total_udp_flow_packets:
            self.total_udp_flow_packets[self.dst_ip] = flow.packet_count
        else:
            self.total_udp_flow_packets[self.dst_ip] += flow.packet_count
        return True

    def get_last_udp_flow_packets(self, dst_ip):
        if self.dpid not in self.last_udp_flow_packets:
            self.last_udp_flow_packets[self.dpid] = {}
            self.last_udp_flow_packets[self.dpid][dst_ip] = 0
            return 0
        elif dst_ip not in self.last_udp_flow_packets[self.dpid]:
            self.last_udp_flow_packets[self.dpid][dst_ip] = 0
        return self.last_udp_flow_packets[self.dpid][dst_ip]

    def evaluate_blocking(self):
        for dst_ip in self.total_udp_flow_packets:
            current = self.total_udp_flow_packets[dst_ip]
            last = self.last_udp_flow_packets.get(self.dpid, {}).get(dst_ip, 0)
            if (current - last) > self.MAX_UDP_PACKETS:
                self.block_udp_packet(dst_ip)
            else:
                self.unblock_udp_packet(dst_ip)

    def block_udp_packet(self, dst_ip):
        log.info("BLOCKING UDP PACKET IN %s" % dst_ip)
        if dst_ip not in self.blocked_udp_packets:
            log.info("Blocking ip: %s" % dst_ip)
            msg = of.ofp_flow_mod()
            msg.match.nw_proto = UDP_PROTOCOL
            msg.match.dl_type = IP_TYPE
            msg.priority = of.OFP_DEFAULT_PRIORITY + 1
            msg.match.nw_dst = dst_ip
            self.send_message_to_all(msg)
            self.blocked_udp_packets[dst_ip] = time()

    def unblock_udp_packet(self, dst_ip):
        if dst_ip not in self.blocked_udp_packets:
            return
        time_passed = time() - self.blocked_udp_packets[dst_ip]
        if time_passed < self.MAX_UDP_TIME:
            return
        del self.blocked_udp_packets[dst_ip]
        log.info("UNBLOCKING UDP PACKET IN %s" % dst_ip)
        log.info("unblocking ip: %s" % dst_ip)
        msg = of.ofp_flow_mod()
        msg.match.nw_proto = UDP_PROTOCOL
        msg.match.dl_type = IP_TYPE
        msg.command = of.OFPFC_DELETE
        msg.match.nw_dst = dst_ip
        self.send_message_to_all(msg)

    @staticmethod
    def send_message_to_all(msg):
        for a_connection in core.openflow.connections:
            a_connection.send(msg)
```

Listing 3: Firewall

```
def launch():
    import pox.log.color
    pox.log.color.launch()
    import pox.log
    pox.log.launch(format="[@@@bold@@@level%(name)-22s@@@reset] " +
                          "@@@bold%(message)s@@@normal")
    from pox.core import core
    import pox.openflow.discovery
    pox.openflow.discovery.launch()
    core.getLogger("openflow.spanning_tree").setLevel("INFO")
    import pox.openflow.spanning_tree
    pox.openflow.spanning_tree.launch()
    from controller import Controller
    core.registerNew(Controller)
    from firewall import Firewall
    core.registerNew(Firewall)
```

Listing 4: file to launch controller and firewall

```
import random


class ECMPTable(object):

    def __init__(self):
        self.table = {}

    def get_port_applying_ecmp(self, data):
        (ports, dpid, dst_dpid, protocol, src_addr, dst_addr) = data
        key = (dst_dpid, protocol, src_addr, dst_addr)
        if dpid not in self.table:
            self.table[dpid] = {}
            random.shuffle(ports)
            self.table[dpid][key] = ports[0]
            return self.table[dpid][key]
        elif key not in self.table[dpid]:
            is_used = False
            for a_port in ports:
                for a_key in self.table[dpid]:
                    if a_port == self.table[dpid][a_key]:
                        is_used = True
                if not is_used:
                    self.table[dpid][key] = a_port
                    return a_port
            random.shuffle(ports)
            self.table[dpid][key] = ports[0]
            return self.table[dpid][key]
        return self.table[dpid][key]

    def save_port(self, data):
        (dpid, dst_dpid, protocol, src_addr, dst_addr, port) = data
        key = (dst_dpid, protocol, src_addr, dst_addr)
        self.table[dpid] = {}
        self.table[dpid][key] = port
```

Listing 5: ecmp table

7. Referencias

1. Create a Learning Switch
   https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch