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

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, src-

port, 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.

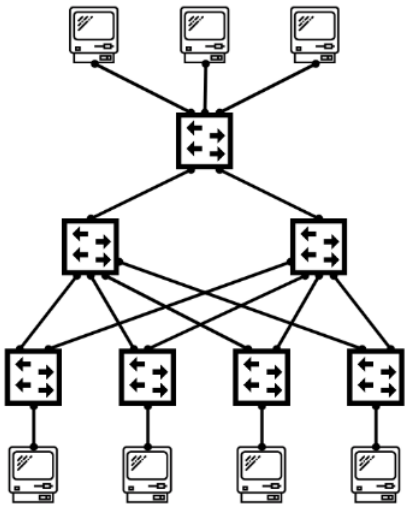

Figura 1: Ejemplo de como seria la topologia con un arbol de altura 3.

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

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

Pruebas realizadas

Conclusiones

Anexos (Código)

```
1  from mininet.topo import Topo
2
3
4  class Topology(Topo):
5      def __init__(self, number_of_levels=3, number_of_clients=3):
6          """
7          :type number_of_levels: int
8          """
9          Topo.__init__(self)
10         self.level_links = {}
11         self.sw_num = 1
12         self.h_num = 1
13         self.number_of_levels = number_of_levels
14         self.number_of_clients = number_of_clients
15         self.add_clients()
16         self.add_switches_and_links()
17         self.add_content_providers()
18
19     def add_clients(self):
20         self.level_links[0] = []
21         for i in range(0, self.number_of_clients):
22             self.level_links[0].append(self.addHost('h%s' %
    ↪ self.h_num))
23             self.h_num += 1
24
25     def add_switches_and_links(self):
```

```
26          for level in range(0, self.number_of_levels):
27              next_level = level + 1
28              number_of_switches_in_level = 2 ** level
29              self.level_links[next_level] = []
30              for i in range(0, number_of_switches_in_level):
31                  sw = self.addSwitch('s%s' % self.sw_num)
32                  self.level_links[next_level].append(sw)
33                  self.sw_num += 1
34                  for device in self.level_links[level]:
35                      self.addLink(sw, device)
36
37      def add_content_providers(self):
38          for sw in self.level_links[self.number_of_levels]:
39              self.addLink(sw, self.addHost('h%s' % self.h_num))
40              self.h_num += 1
41
42
43  topos = {
44      'mytopo': (lambda levels=3, clients=3: Topology(levels, clients))
45  }
```

Listing 1: Topology

```
1  from pox.core import core
2  import pox.openflow.libopenflow_01 as of
3  from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST
4  from pox.lib.util import dpidToStr
5  from pox.lib.packet.packet_utils import _ethtype_to_str
6  from pox.host_tracker.host_tracker import host_tracker
7  import pox.lib.packet as pkt
8  from pox.lib.revent import *
9  from ecmp_table import ECMPTable
10
11 log = core.getLogger()
12
13
14 class Controller(object):
15
16     def __init__(self):
17         core.openflow.addListeners(self)
18
19         def startup():
20             core.openflow.addListeners(self, priority=0)
21             core.openflow_discovery.addListeners(self)
22
23         core.call_when_ready(startup, ('openflow',
       ↪ 'openflow_discovery'))
24
25         self.event = None
26         self.dpid = None
27         self.in_port = None
28         self.packet = None
```

```
29          self.dst_dpid = None
30          self.out_port = None
31          self.table = ECMPTable()
32          self.eth_packet = None
33          self.ip_packet = None
34          self.arp_packet = None
35          self.icmp_packet = None
36          self.tcp_packet = None
37          self.udp_packet = None
38          self.net_packet = None
39          self.protocol_packet = None
40          self.protocol = None
41          self.arp_table = {}
42          self.is_ip = True
43          self.adjacency = {}
44          self.host_tracker = host_tracker()
45          log.info("controller ready")
46
47      def add_adjacency(self, dpid1, port1, dpid2, port2):
48          if dpid1 not in self.adjacency:
49              self.adjacency[dpid1] = {}
50          self.adjacency[dpid1][port1] = {
51                  "dpid": dpid2,
52                  "port": port2
53              }
54
55      def remove_adjacency(self, dpid, port):
56          if dpid not in self.adjacency:
57              return
58          if port not in self.adjacency[dpid][port]:
59              return
60          del self.adjacency[dpid][port]
61
62      def _handle_LinkEvent(self, event):
63
    ↪ log.info("------------------------------------------------------------")
64          link = event.link
65          if event.added:
66              self.add_adjacency(link.dpid1, link.port1, link.dpid2,
    ↪ link.port2)
67              self.add_adjacency(link.dpid2, link.port2, link.dpid1,
    ↪ link.port1)
68          elif event.removed:
69              self.remove_adjacency(link.dpid1, link.port1)
70              self.remove_adjacency(link.dpid2, link.port2)
71          log.info('link added is %s' %event.added)
72          log.info('link removed is %s' %event.removed)
73          log.info('switch1 %d' %link.dpid1)
74          log.info('port1 %d' %link.port1)
75          log.info('switch2 %d' %link.dpid2)
76          log.info('port2 %d' %link.port2)
77
```

```
    ↪ log.info("--------------------------------------------------------")
78
79      def _handle_ConnectionUp(self, event):
80          log.debug("Connection %s" %(event.connection,))
81
82          msg = of.ofp_flow_mod()
83          msg.match.dl_dst = ETHER_BROADCAST
84          msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
85          event.connection.send(msg)
86
87          msg = of.ofp_flow_mod()
88          msg.match.dl_type = pkt.ethernet.IPV6_TYPE
89          event.connection.send(msg)
90
91          msg = of.ofp_flow_mod()
92          msg.match.dl_type = pkt.ethernet.ARP_TYPE
93          msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
94          event.connection.send(msg)
95
96      @staticmethod
97      def print_msg(msg):
98          print "++++++++++++++++++++++++++++++++++++++++++++"
99          print msg
100         print "++++++++++++++++++++++++++++++++++++++++++++"
101
102     def fill_arp_table(self):
103         entry = self.host_tracker.getMacEntry(self.addr_dst)
104         if entry is None:
105             log.info("HOST TRACKER COULD NOT FIND ENTRY DST")
106             return
107         self.arp_table[self.addr_dst] = {
108             "dpid": entry.dpid,
109             "port": entry.port
110         }
111
112     def print_adjacents(self):
113         msg = ""
114         for dpid in self.adjacency:
115             msg += "dpid: %s: [" %dpid
116             for port in self.adjacency[dpid]:
117                 msg += "%s, " %self.adjacency[dpid][port]["dpid"]
118             msg += "]"
119             log.info(msg)
120             msg = ""
121
122     def has_discovered_the_entire_topology(self):
123         if len(self.adjacency.keys()) != 7:
124             return False
125         for dpid in self.adjacency:
126             size = len(self.adjacency[dpid])
127             if dpid in [4, 5, 6, 7] and size < 2:
128                 return False
```

```
129             if dpid in [2, 3] and size < 4:
130                 return False
131             if dpid == 1 and size < 1:
132                 return False
133         return True
134 
135     def _handle_PacketIn(self, event):
136         self.host_tracker._handle_PacketIn(event)
137         if not self.has_discovered_the_entire_topology():
138             log.info("Please wait... learning the topology")
139             return
140         self.event = event
141         self.dpid = event.connection.dpid
142 
    ↪ log.info("--------------------------------------------------------------------------------")
143         self.print_adjacents()
144         log.info("SWITCH %s" % self.dpid)
145         self.in_port = event.port
146         self.packet = event.parsed
147         log.info("ports: %s" % event.connection.ports)
148         log.info("ports: %s" % event.connection.ports)
149         log.info("in port: %s" % self.in_port)
150         if not self.packet.parsed:
151             log.warning("%i %i ignoring unparsed packet" %
152                         (self.dpid, self.in_port))
153             return
154         log.info("HOST SRC %s" % self.packet.src)
155         log.info("HOST DST: %s" % self.packet.dst)
156         self.eth_packet = self.packet.find(pkt.ethernet)
157         self.addr_dst = self.packet.dst
158         self.fill_arp_table()
159         self.ip_packet = self.packet.find(pkt.ipv4)
160         self.arp_packet = self.packet.find(pkt.arp)
161         self.icmp_packet = self.packet.find(pkt.icmp)
162         self.tcp_packet = self.packet.find(pkt.tcp)
163         self.udp_packet = self.packet.find(pkt.udp)
164 
165         if not self.validate_protocols():
166 
167             return
168         if not self.validate_net_packets():
169             return
170 
171         if self.addr_dst not in self.arp_table:
172             log.warning("Could not find dst")
173             return self.flood()
174 
175         entry = self.arp_table[self.addr_dst]
176         self.dst_dpid = entry["dpid"]
177         if self.dpid == self.dst_dpid:
178             log.info("Current switch is destination")
179             self.out_port = entry["port"]
```

```
180         else:
181             if self.packet.dst.is_multicast:
182                 self.print_msg("MULTICAST")
183                 return self.flood()
184             log.info("Finding minimun paths from %s to %s"
185                      %(self.dpid, self.dst_dpid))
186             minimun_paths = self.get_minimun_paths()
187             log.info("finding out port")
188             self.out_port = self.get_out_port(minimun_paths)
189             if self.out_port is None:
190                 log.info("Could not find out port")
191                 return
192         log.info("out port: %s" %self.out_port)
193         log.info("Updating flow table")
194         self.update_table()
195         log.info("Sending packet")
196         self.send_packet()
197 
198     def flood(self):
199         log.info("FLOODING PACKET")
200         msg = of.ofp_packet_out()
201         msg.buffer_id = self.event.ofp.buffer_id
202         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
203         msg.data = self.event.ofp
204         msg.in_port = self.in_port
205         self.event.connection.send(msg)
206 
207     def validate_protocols(self):
208         if self.udp_packet is not None:
209             log.info("UDP packet!")
210             self.protocol = "UDP"
211             self.protocol_packet = self.udp_packet
212             return True
213         elif self.tcp_packet is not None:
214             log.info("TCP packet!")
215             self.protocol = "TCP"
216             self.protocol_packet = self.tcp_packet
217             return True
218         elif self.icmp_packet is not None:
219             log.info("ICMP packet!")
220             self.protocol = "ICMP"
221             self.protocol_packet = self.icmp_packet
222             return True
223         else:
224             log.warning("icmp, tcp and udp packets are None!")
225             return False
226 
227     def validate_net_packets(self):
228         if _ethtype_to_str[self.packet.type] == "IPV6":
229             log.warning("DROP IPV6 packet")
230             return False
231         if self.eth_packet is None:
```

```
232             log.warning("ETHERNET packet is None!")
233             return False
234         if self.ip_packet is not None:
235             log.info("IP packet!")
236             self.is_ip = True
237             self.net_packet = self.ip_packet
238         elif self.arp_packet is not None:
239             log.info("ARP packet!")
240             self.is_ip = False
241             self.net_packet = self.arp_packet
242         else:
243             log.warning("ARP and TCP packets are None!")
244             return False
245         return True
246 
247     def match_protocol_packets(self, msg):
248         if self.is_ip:
249             msg.match.nw_src = self.net_packet.srcip
250             msg.match.nw_dst = self.net_packet.dstip
251             msg.match.nw_proto = self.net_packet.protocol
252             return msg
253         msg.match.nw_src = self.net_packet.protosrc
254         msg.match.nw_dst = self.net_packet.protodst
255         msg.match.nw_proto = self.net_packet.prototype
256         return msg
257 
258     def match_packet(self, msg):
259         if not self.is_ip:
260             return msg
261         msg.match.nw_src = self.ip_packet.srcip
262         msg.match.nw_dst = self.ip_packet.dstip
263         msg.match.nw_proto = self.ip_packet.protocol
264         return msg
265 
266     def update_table(self):
267         msg = of.ofp_flow_mod()
268         msg.match.dl_type = self.eth_packet.type
269         msg = self.match_packet(msg)
270         msg.buffer_id = self.event.ofp.buffer_id
271         if self.protocol != "ICMP":
272             msg.match.tp_src = self.protocol_packet.srcport
273             msg.match.tp_dst = self.protocol_packet.dstport
274         msg.actions.append(of.ofp_action_output(port=self.out_port))
275         self.event.connection.send(msg)
276         self.balance_of_charges()
277 
278     def get_minimun_paths(self):
279         adjacents = self.get_adjacents(self.dpid)
280         if not adjacents:
281             log.warning("NO ADJACENTS FOUND")
282             return []
283         paths = [[neighbour] for neighbour in adjacents]
```

```
284         while not self.has_found_a_path(paths, self.dst_dpid):
285             last_paths = paths[:]
286             for path in last_paths:
287                 adjacents = self.get_adjacents(path[-1]["dpid"])
288                 for an_adjacent in adjacents:
289                     if an_adjacent["dpid"] != self.dpid:
290                         if not self.node_belongs_path(an_adjacent,
    ↪ path):
291                             paths.append(path + [an_adjacent])
292         return self.filter_paths(paths, self.dst_dpid)
293 
294     def node_belongs_path(self, node, path):
295         dpid = node["dpid"]
296         for a_node in path:
297             if a_node["dpid"] == dpid:
298                 return True
299         return False
300 
301     def get_out_port(self, paths_to_dst):
302         if len(paths_to_dst) == 0:
303             return None
304         ports = self.get_all_ports(paths_to_dst)
305         data = (
306             ports,
307             self.dpid,
308             self.dst_dpid,
309             self.protocol,
310             self.packet.src,
311             self.packet.dst
312         )
313         return self.table.get_port_applying_ecmp(data)
314 
315     def balance_of_charges(self):
316         log.info("saving (%s, %s, %s): %s" %
317                  (self.dpid, self.dst_dpid, self.protocol,
    ↪ self.out_port))
318         data = (
319             self.dpid,
320             self.dst_dpid,
321             self.protocol,
322             self.packet.src,
323             self.packet.dst,
324             self.out_port
325         )
326         self.table.save_port(data)
327 
328     @staticmethod
329     def get_all_ports(paths_to_dst):
330         return [a_path[0]["port"] for a_path in paths_to_dst]
331 
332     @staticmethod
333     def filter_paths(paths, dpid):
```

```
334         paths_to_dst = []
335         for path in paths:
336             if path[-1]["dpid"] != dpid:
337                 continue
338             paths_to_dst.append(path)
339         return paths_to_dst
340 
341     def has_found_a_path(self, paths, dpid):
342         for path in paths:
343             if path[-1]["dpid"] == dpid:
344                 log.info("FOUND A PATH!")
345                 return True
346         return False
347 
348     # @staticmethod
349     def get_adjacents_last(self, dpid):
350         adjacents = []
351         for an_adjacent in core.openflow_discovery.adjacency:
352             if an_adjacent.dpid1 == dpid:
353                 adjacents.append({
354                     "dpid": an_adjacent.dpid2,
355                     "port": an_adjacent.port1
356                 })
357             elif an_adjacent.dpid2 == dpid:
358                 adjacents.append({
359                     "dpid": an_adjacent.dpid1,
360                     "port": an_adjacent.port2
361                 })
362         return self.filter_repeated(adjacents)
363 
364     def get_adjacents(self, dpid):
365         adjacents = []
366         if dpid not in self.adjacency:
367             return adjacents
368         for port in self.adjacency[dpid]:
369             adjacents.append({
370                 "dpid": self.adjacency[dpid][port]["dpid"],
371                 "port": port
372             })
373         return adjacents
374 
375     def filter_repeated(self, adjacents):
376         filtered = []
377         belongs = False
378         for an_adjacent in adjacents:
379             for final_adjacent in filtered:
380                 dpid_1 = final_adjacent["dpid"]
381                 dpid_2 = an_adjacent["dpid"]
382                 port_1 = final_adjacent["port"]
383                 port_2 = an_adjacent["port"]
384                 if dpid_1 == dpid_2 and port_1 == port_2:
385                     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:
```

```
37              body = of.ofp_flow_stats_request()
38              connection.send(of.ofp_stats_request(body=body))
39
40      def _handle_flowstats_received(self, event):
41          log.info("handle denial of service")
42          self.dpid = event.connection.dpid
43          self.total_udp_flow_packets = {}
44          for flow in event.stats:
45              self.dst_ip = flow.match.nw_dst
46              if self.dst_ip is None:
47                  log.info("DST IP IS NONE. COULD NOT HANDLE DoS")
48                  continue
49              if not self.get_udp_flow(flow):
50                  continue
51              self.evaluate_blocking()
52              current = self.total_udp_flow_packets[self.dst_ip]
53              self.last_udp_flow_packets[self.dpid] = {}
54              self.last_udp_flow_packets[self.dpid][self.dst_ip] =
    ↪ current
55
56      def get_udp_flow(self, flow):
57          if self.dst_ip is None or flow.match.nw_proto !=
    ↪ UDP_PROTOCOL:
58              return False
59          if self.dst_ip not in self.total_udp_flow_packets:
60              self.total_udp_flow_packets[self.dst_ip] =
    ↪ flow.packet_count
61          else:
62              self.total_udp_flow_packets[self.dst_ip] +=
    ↪ flow.packet_count
63          return True
64
65      def get_last_udp_flow_packets(self, dst_ip):
66          if self.dpid not in self.last_udp_flow_packets:
67              self.last_udp_flow_packets[self.dpid] = {}
68              self.last_udp_flow_packets[self.dpid][dst_ip] = 0
69              return 0
70          elif dst_ip not in self.last_udp_flow_packets[self.dpid]:
71              self.last_udp_flow_packets[self.dpid][dst_ip] = 0
72          return self.last_udp_flow_packets[self.dpid][dst_ip]
73
74      def evaluate_blocking(self):
75          for dst_ip in self.total_udp_flow_packets:
76              current = self.total_udp_flow_packets[dst_ip]
77              last = self.last_udp_flow_packets.get(self.dpid,
    ↪ {}).get(dst_ip, 0)
78              if (current - last) > self.MAX_UDP_PACKETS:
79                  self.block_udp_packet(dst_ip)
80              else:
81                  self.unblock_udp_packet(dst_ip)
82
83      def block_udp_packet(self, dst_ip):
```

```
84          log.info("BLOCKING UDP PACKET IN %s" % dst_ip)
85          if dst_ip not in self.blocked_udp_packets:
86              log.info("Blocking ip: %s" % dst_ip)
87              msg = of.ofp_flow_mod()
88              msg.match.nw_proto = UDP_PROTOCOL
89              msg.match.dl_type = IP_TYPE
90              msg.priority = of.OFP_DEFAULT_PRIORITY + 1
91              msg.match.nw_dst = dst_ip
92              self.send_message_to_all(msg)
93              self.blocked_udp_packets[dst_ip] = time()
94
95      def unblock_udp_packet(self, dst_ip):
96          if dst_ip not in self.blocked_udp_packets:
97              return
98          time_passed = time() - self.blocked_udp_packets[dst_ip]
99          if time_passed < self.MAX_UDP_TIME:
100             return
101         del self.blocked_udp_packets[dst_ip]
102         log.info("UNBLOCKING UDP PACKET IN %s" % dst_ip)
103         log.info("unblocking ip: %s" % dst_ip)
104         msg = of.ofp_flow_mod()
105         msg.match.nw_proto = UDP_PROTOCOL
106         msg.match.dl_type = IP_TYPE
107         msg.command = of.OFPFC_DELETE
108         msg.match.nw_dst = dst_ip
109         self.send_message_to_all(msg)
110
111     @staticmethod
112     def send_message_to_all(msg):
113         for a_connection in core.openflow.connections:
114             a_connection.send(msg)
```

Listing 3: Firewall

```
1  def launch():
2      import pox.log.color
3      pox.log.color.launch()
4      import pox.log
5      pox.log.launch(format="[@@@bold@@@level%(name)-22s@@@reset] " +
6                             "@@@bold%(message)s@@@normal")
7      from pox.core import core
8      import pox.openflow.discovery
9      pox.openflow.discovery.launch()
10     core.getLogger("openflow.spanning_tree").setLevel("INFO")
11     import pox.openflow.spanning_tree
12     pox.openflow.spanning_tree.launch()
13     from controller import Controller
14     core.registerNew(Controller)
15     from firewall import Firewall
16     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