MEMORY

NETWORK DEVICES

Network Laboratory

Authors:

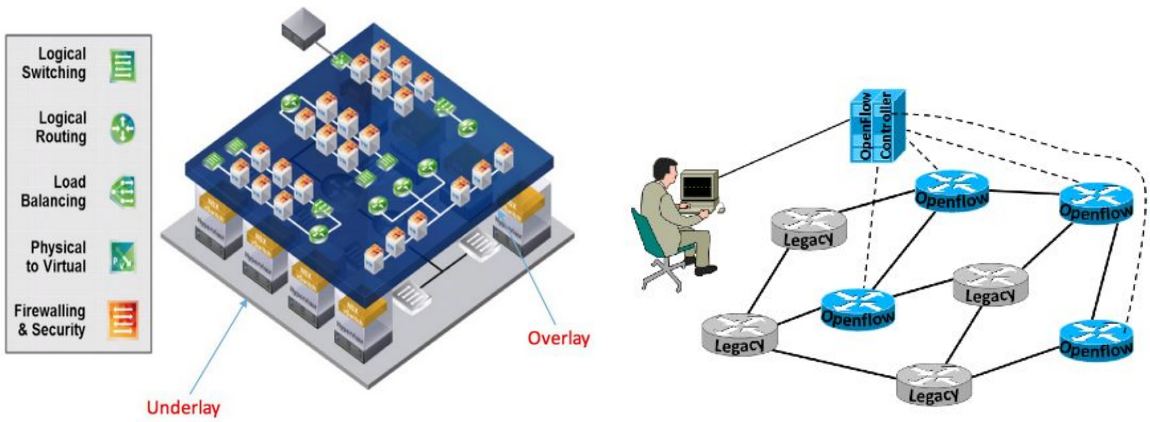
Eduardo Borges Fernández (alu0100885613@ull.edu.es)
Jonathan Eliot Viera Rivas (alu0100831863@ull.edu.es)
José Santiago Padilla Álvarez (alu0101069937@ull.edu.es)

- **Introduction.**

This memory tries to be an introduction to the configuration of network devices through the paradigm of 'software-defined networks' (SDN).
The use of SDN has been favored by the need to use WAN links and metropolitan links efficiently. The current routing protocols that are routinely used are insufficient to support the current large amounts of traffic between clients and application / content servers.
An approach to the solution through SDN is optimal because the telematic equipment can react in real time to peaks of traffic and adapt for a very efficient use of the WAN / MAN links. In this way all telecommunication equipment that is part of the same network reports the capacity and instantaneous use of all its links to a "brain" (controller). The controller makes the appropriate routing decisions of the data flows (whether TCP, UDP or others) and lets it know all the communications equipment involved. The topology that we will use in this case for the router controller check will be the following:

- **SDN and Openflow concept.**

SDN is a paradigm in computer networks that aims to implement network services dynamically and scalable.



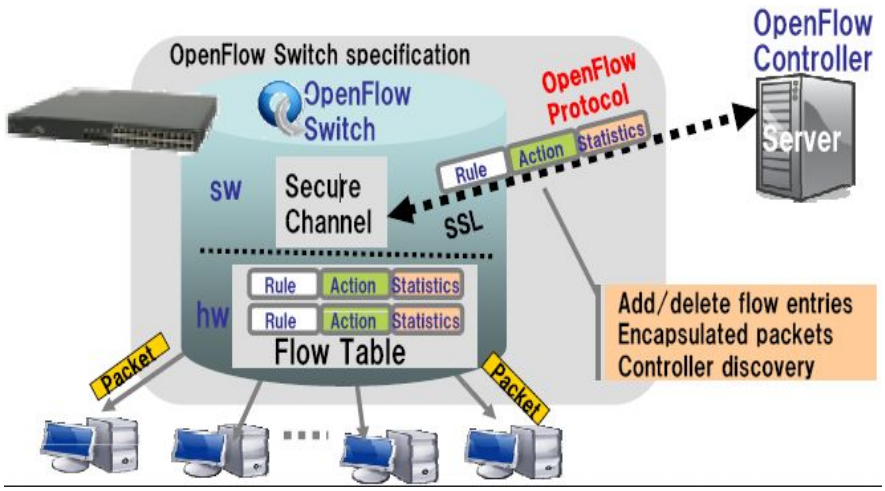
SDN separates the control layer (software) from the data layer (hardware). The control is centralized in one or more external equipment (controllers) that executes a software for that purpose.
Openflow is an open communications program that allows a server to determine the path of packet forwarding that is recommended to follow a network of switches. With the OpenFlow protocol, a network can be managed centrally, regardless of the number of switches.
The forwarding instructions are based on flow tables, defined by several parameters: source ports, destination, IP addresses and other series of packet characteristics.
The flow table is a set of inputs with several fields that are compared to the received packet and an associated action or set of actions.

- **How an Openflow Switch Works.**

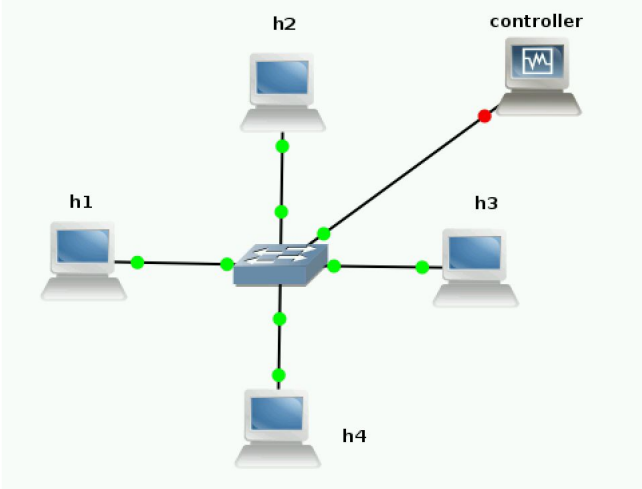
An Openflow switch has one or more flow tables, each with a set of inputs. The switch pipeline defines how packets interact with flow tables.
When a packet enters the switch it is compared to the pattern of each of the table entries, starting with the first table. When match fields match, the counters are updated and the instruction, or set of instructions associated with the input is executed. In case the instruction set is empty, the packet is normally forwarded to some output port on the switch.
If no compatible input exists in the flow tables, the switch sends the packet to the controller for a decision. The driver will either delete the package or insert a new entry in the flow table.

-Flow Table Entries:

Pattern Fields: *Contains parameters such as ports, addresses, packet headers, or metadata from another previous flow table to compare. If the value of the input is 'ANY', the field is omitted.*
Priority: to order the entry in the table.
Counters: updated when a match is found.
Instructions: allows you to modify the set of associated actions
Timeouts: maximum time before the switch deletes the table entry.
Cookie: controller value, which is not used for packet processing.



- **Implementation of a switch L2.**




```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    self.mac_to_port = dict()  # Creacion diccionario vacio
    msg = ev.msg               # Objeto que representa la estuctura de datos PacketIn.
    datapath = msg.datapath    # Identificador del datapath correspondiente al switch.
    ofproto = datapath.ofproto # Protocolo utilizado que se fija en una etapa
                               # de negociacion entre controlador y switch

    ofp_parser=datapath.ofproto_parser # Parser con la version OF
                                       # correspondiente

    in_port = msg.match['in_port'] # Puerto de entrada.

    # Ahora analizamos el paquete utilizando las clases de la libreria packet.
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
```

```
    # Extraemos la MAC de destino

    dst = eth.dst
    src = eth.src
    self.mac_to_port[src] = in_port


    if dst in self.mac_to_port.keys():
         match = ofp_parser.OFPMatch(eth_dst=dst)
         actions = [ofp_parser.OFPActionOutput(self.mac_to_port[dst])]
         self.add_flow(datapath, 0, match, actions, msg.buffer_id)
    elif haddr_to_bin(dst) == mac.BROADCAST or mac.is_multicast(haddr_to_bin(dst)):
         # Ahora creamos el match
         # fijando los valores de los campos
         # que queremos casar.
         match = ofp_parser.OFPMatch(eth_dst=dst)

         # Creamos el conjunto de acciones: FLOOD
         actions = [ofp_parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
         self.add_flow(datapath, 0, match, actions, msg.buffer_id)
    else:
         self.send_packet(datapath, ofproto.OFPP_FLOOD, pkt)
```

The principal objetive of a switch L2 is commute the ethernet packets when they arrive, then we had to analyze what to do and implement it, extracting the ethernet header, to know the destination and source direction.

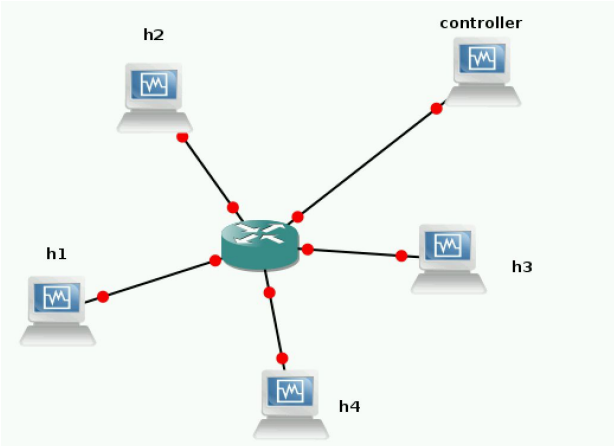
When a packet reaches the switch, there are three cases:

- The first case is that the controller knows the port of the mac destination, then the match will be that every packet which went into the switch and has this destination mac will be thrown out of the port that the controller knows. And the rule (match-action) will be added to the flows table.
- The second case is that the packet destination is multicast or broadcast, then every packet which has this destination will flood all the ports and will be added to the flows table to be faster the next time.
- The third case is that the controller doesn't know the port of the destination mac. Then it will flood all ports with the packet, but the controller won't add anything on the flows table, because it will add to the internal dictionary the port of this destination mac when the destination host response is received `(self.mac_to_port[src]=in_port)`

The function "add_flow(datapath,0,match,action, msg.buffer_id)" add a entry (match-actions) on the flows table.

The function "send_packet(datapath, port, packet)" send the packets for the select port.

- **Implementation of a simple router**



The difference between a router and a switch L2 is that the router works at level 3 with ip directions inter networks and can response packets like another host. To obtain the MAC direction of every PC or router interface we'll use some arp functions, arp is used to obtain the MAC direction of a IP direction, and the MAC is needed because a IP packet has encapsulated a ethernet packet which contains required fields like source MAC and destination MAC.

An example, if a host (h1) want to ping router, the steps are the next:

- Send an arp request (Because h1 doesn't know the destination MAC)
- The router will receive an arp request and will respond with an arp reply.
- host (h1) now knows the destination MAC. Then he proceed to send the packet.

```
# Tabla de enrutamiento
class RoutingTable:
  def __init__(self, ports, filename):
    self.table = []
    for p in ports.ports.keys():
      (ip, mask, mac) = ports.ports[p]
      ip = IPNetwork("%s/%s" % (ip, mask))
      self.add_route(str(ip.network), ip.prefixlen, p, None)

    import csv
    try:
       with open(filename, 'rb') as csvfile:
         reader = csv.reader(csvfile, delimiter=' ', quotechar='|')
         for row in reader:
            LOG.info(row)
            self.add_route(row[0], int(row[1]), int(row[2]), row[3])

    except:
       LOG.debug("File not found")

    LOG.info(self.table)

  def add_route(self, network, mask, port, next_hop):

    self.table.append((network, mask, port, next_hop))
```

```
def search(self, ip):
    current_route = None
    current_mask = 0

    for row in self.table:
      str_mask = str(row[1])
      aux = IPNetwork("%s/%s" % (ip, str_mask))
      result = aux.ip & aux.netmask

      if result == IPAddress(row[0]) and current_mask < row[1]:
          current_route = (row[0],row[1],row[2],row[3])
          current_mask = row[1]

    return current_route
```

One of the elements of the router is the Routing table, which has all the destination networks with every interface, netmask and gateway associated to obtain the correct routing of a packet with an IP direction. In code it's just a class which contains an attribute table/list, and the info to mount the routing table comes on the constructor from a .csv file (for indirectly connected networks) and from a class ports (for directly connected networks).

We had to implement the `search(self,ip)` function, which is needed to find the correct entry of the routing table for a packet with a destination IP. The function receive as parameter the destination IP and for each row in the table apply and logic operation AND with the mask and compare the result with the destination network, if there are the same it takes this entry like current result, but it finally will select the correct destination network with the longest mask. The tactic of that implementation is that we used an IPNetwork where the IP is the analyzed IP and the mask is changing constantly depending of the current row, then just had to magic an AND between that ip and the current mask.

```
class Router(app_manager.RyuApp):

    def __init__(self,*args, **kwargs):
        super(Router, self).__init__(*args, **kwargs)
        name = "r1"
        LOG.info("Configuring %s" % (name))
        self.name = name
        self.ports = Ports('%s_ports.csv' % (name))
        self.arpcache = ARPCache()
        self.routing = RoutingTable(self.ports, '%s_routing.conf' % (name))
        self.pending = PendingPackets()
        self.firewall = FirewallingTable("%s_firewalling.csv" % (name))
```

That function is the initialize of the Router class, which contain a name, ports with assigned direction from a .csv file, an arpcache which is a list of the associated IP-MAC, a routing table, a queue of pending packets (packets waiting for arp) and a firewall to filter packets following some rules.

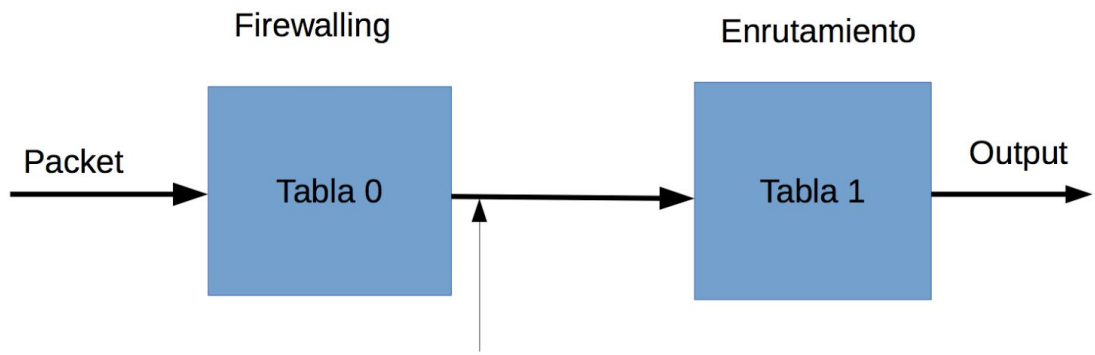
```
# Gestion del reenvío de paquetes.
def forwarding(self, msg):
    packet = Packet(msg.data)
    ip_packet = packet.get_protocol(ipv4.ipv4)

    entry = self.routing.search(ip_packet.dst)

    if (entry[3] == None):
        next_hop = ip_packet.dst
        dir_ip = ip_packet.dst
    else:
        next_hop = entry[3]
        str_mask = str(entry[1])
        dir_ip = IPNetwork("%s/%s" % (entry[0], str_mask))

    next_mac = self.arpcache.get_mac(next_hop,entry[2])

    if(next_mac != None):
        self.add_forwarding_flow(msg, entry[2], dir_ip, next_mac, 32)
    else:
        self.arp_request(msg.datapath, next_hop, entry[2])
        self.pending.addPendingPacket(next_hop, msg, dir_ip, entry[1])

    None
```

The `forwarding` function is called when a packet arrives, has a message parameter. First we extract the IP header, with the destination IP we looking for the destination in the RoutingTable with the function search, if doesn't exist a gateway, the next hop will be the destination, else the next hop will be the gateway and the destination is another network. The next step is ask the arpcache if the associated MAC of that IP is known. If this is right, we call `add_forwarding_flow` function to add the entry to the flows table, else we send an arp request to obtain the direction MAC of the next hop and putting on the packet in queue to be sended.

```
# Añade una entrada a la tabla de flujo para reenvío.
 def add_forwarding_flow(self, msg, port, dst_ip, dst_mac, priority):
    datapath = msg.datapath

    packet = Packet(msg.data)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    (ip,mask,eth_src) = self.ports.get_port(port)

    match = parser.OFPMatch(ipv4_dst = dst_ip, eth_type = ether.ETH_TYPE_IP)

    actions = [parser.OFPActionSetField(eth_dst=dst_mac),
    parser.OFPActionSetField(eth_src=eth_src),
    parser.OFPActionOutput(port=port)]

    self.add_flow(datapath,priority, match, actions, buffer_id=msg.buffer_id)

    None
```

The `add_forwarding_flow` function is called when is needed to add in the flow table, defining the match, which in this case is that the packet must be an IP packet, and the IP destination equal to the IP set by controller, and the actions, changing fields of every packet that arrives, here just it changes the destination mac because was empty, upload the new source MAC and throw out the packet for the selected port. Finally we call `add_flow` to apply the instructions on the flows table.

- **Turn the router on a router-firewall**



The idea of the firewall is having two flows tables. The first one with id = 0 is the firewalling table with the firewall match+actions, the second table with id = 1, is the previous table 0 in a router without firewall.

```
# Tabla de firewall
class FirewallingTable:
  def __init__(self,filename):
    self.table = []

    import csv
    try:
       with open(filename, 'rb') as csvfile:
          reader = csv.reader(csvfile, delimiter=' ', quotechar='|')
          for row in reader:
             LOG.info(row)
             self.add_firewall(int(row[0]), int(row[1]), row[2], row[3], row[4], row[5])

    except:
       LOG.debug("File not found")

    LOG.info(self.table)

  def add_firewall(self, port_src, port_dst, ip_src, ip_dst, protocol, action):

    self.table.append((port_src, port_dst, ip_src, ip_dst, protocol, action))
```

First, to add firewall on router, we need to create a second table that it have the firewall configuration by a file with extension ".csv", the created table will have six columns that contains the source port, destination port, source ip, destination ip, the protocol (udp or tcp) and the action (accept or drop). By default, we assume that all packets that don’t match with the archive rules “.csv” will be accepted.

```
def firewarding(self, datapath):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    actions = []

    for row in self.firewall.table:
        if(row[4] == "tcp"):
          match = parser.OFPMatch(tcp_src=row[0], tcp_dst=row[1], ipv4_src=IPAddress(row[2]),
                  ipv4_dst = IPAddress(row[3]), ip_proto = 6 , eth_type = ether.ETH_TYPE_IP)
        else:
          match = parser.OFPMatch(udp_src=row[0], udp_dst=row[1], ipv4_src=IPAddress(row[2]),
                  ipv4_dst = IPAddress(row[3]), ip_proto = 17 , eth_type = ether.ETH_TYPE_IP)

        if(row[5] == "Drop"):
          self.add_flow_drop(datapath, 1, match, actions)
        else:
          self.add_flow_goto(datapath, 1, match, actions)

    None
```

Now we must compare the packet with our firewall table, because we should know if the packet should pass or not. First of all, we will compare the protocol, then we can determine if we will set on match the udp or tcp source and destination ports, with others parameters that must be equal to the defined, like IP's. Second, we will compare the action, if the action is accept we call `add_flow_goto` ,else we call add_flow_drop. The actions list is empty because in both cases we won't do any action.

```
#  Inserta una entrada a la tabla de flujo.
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
    if buffer_id:
       mod = parser.OFPFlowMod(datapath=datapath,table_id=1,buffer_id=buffer_id,
             priority=priority, match=match,
             instructions=inst, idle_timeout=30,command=ofproto.OFPFC_ADD)
    else:
       mod = parser.OFPFlowMod(datapath=datapath,table_id=1,priority=priority,
             match=match, instructions=inst, idle_timeout=30,command=ofproto.OFPFC_ADD)

    datapath.send_msg(mod)

def add_flow_drop(self, datapath, priority, match, actions, buffer_id=None):
    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)

def add_flow_goto(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionGotoTable(1)]

    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)
```

The function `add_flow` is the function to add the match+action to the table 1.
The function add_flow_drop has the same code as `add_flow` but in the table 0 and the instruction is execute the actions, but add_flow_drop will be called always with a list of empty actions, then it will drop the packet from the router.
The `add_flow_goto` function, pass the packet to the table 1 because was accepted by the firewall.

This code with the arp functions is the needed to get the router-firewall up.

Testing

```
vagrant@ryubox:/vagrant/router$ sudo ryu-manager ryu_router.py
loading app ryu_router.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu_router.py of Router
Configuring r1
['1', '10.0.1.1', '24', '00:00:00:01:01:00']
['2', '10.0.2.1', '24', '00:00:00:01:02:00']
['3', '10.0.3.1', '24', '00:00:00:01:03:00']
['4', '10.0.4.1', '24', '00:00:00:01:04:00']
[('10.0.1.0', 24, 1, None), ('10.0.2.0', 24, 2, None), ('10.0.3.0', 24, 3, None), ('10.0.4.0', 24, 4, None)]
['443', '443', '10.0.1.2', '10.0.2.2', 'tcp', 'drop']
['500', '500', '10.0.1.2', '10.0.3.2', 'udp', 'accept']
[(443, 443, '10.0.1.2', '10.0.2.2', 'tcp', 'drop'), (500, 500, '10.0.1.2', '10.0.3.2', 'udp', 'accept')]
```

First, we load the controller, with the routing table, the ports and the firewall rules.

```
root@ryubox:/vagrant/router# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5.421s, table=0, n_packets=4, n_bytes=280, idle_age=2, pri
ority=0 actions=CONTROLLER:65535
 cookie=0x0, duration=5.420s, table=0, n_packets=0, n_bytes=0, idle_age=5, prior
ity=1,udp,nw_src=10.0.1.2,nw_dst=10.0.3.2,tp_src=500,tp_dst=500 actions=drop
 cookie=0x0, duration=5.421s, table=0, n_packets=0, n_bytes=0, idle_age=5, prior
ity=1,tcp,nw_src=10.0.1.2,nw_dst=10.0.2.2,tp_src=443,tp_dst=443 actions=drop
```

This is the state of the flows table when we load the controller. As we can see it has loaded the two rules of the firewall, both rules action seems like drop when the real action of the udp rule is accept, but this is a bug of the program.

```
#!/bin/bash

func(){
for((i=0; i<= 10; i++))
do
  hping3 -s 443 -p 443 -c 1 10.0.2.2
done
}

func2(){
for((i=0; i<= 5; i++))
do
  hping3 -s 500 -p 500 -c 1 10.0.3.2
done
}

script(){
$(func)
$(func2)
}

script
```

This is an script that we made to test the firewall, the first func() send to the selected port (-p) and IP and from the selected port (-s) 1 packet(-c) 11 times, its because with hping after the first packet the destination port is being increased. Then, it's going to send 10 packets, and all of them must be dropped by the firewall.
The func2() is a test of accepted packets, we will send 6 of them.

```
--- 10.0.2.2 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

--- 10.0.2.2 hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
./script.sh: line 19: HPING: command not found

--- 10.0.3.2 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 6.5/6.5/6.5 ms

--- 10.0.3.2 hping statistic ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 5.2/5.2/5.2 ms
```

On execution time we can see that drop packets are lost and the accept packets are received. It's a good signal.

```
root@ryubox:/vagrant/router# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=96.674s, table=0, n_packets=23, n_bytes=1334, idle_age=25,
 priority=0 actions=CONTROLLER:65535
 cookie=0x0, duration=96.674s, table=0, n_packets=0, n_bytes=0, idle_age=96, pri
ority=1,udp,nw_src=10.0.1.2,nw_dst=10.0.3.2,tp_src=500,tp_dst=500 actions=drop
 cookie=0x0, duration=96.674s, table=0, n_packets=11, n_bytes=594, idle_age=31,
priority=1,tcp,nw_src=10.0.1.2,nw_dst=10.0.2.2,tp_src=443,tp_dst=443 actions=dro
p
```

Finally on the flows table we can see how n_packets = 11, the 11 drop packets that we sent, and the 6 accept packet doesn't appear because they passes the table.

- **Bibliografía.**

IP addresses, subnets and ranges:
http://netaddr.readthedocs.io/en/latest/tutorial_01.html

hping3:
https://linux.die.net/man/8/hping3