SAN JOSE STATE UNIVERSITY

Charles. W .Davidson college of Engineering

CMPE-210 PROJECT

SDN & NFV

Counter Acting UDP Flooding Attacks in SDN

Submitted By:
Shruthi kondeti - 14
(012455690)

Submitted to:
Prof. Younghee Park
Department of Computer Engineering

ABSTRACT:

In this project we address one serious SDN-specific attack that is a data-to-control plane saturation attack, which overloads the infrastructure of SDN networks. In this, an attacker produces a large amount of table-miss packet_in messages by flooding the network with UDP packets to consume resources in both control plane and data plane. Our prototype provides an efficient way to reduce DoS (denial-of-service) attacks by maintaining constant communication bandwidth of the queues in OVS (open virtual switch) using RYU controller. QoS regulation techniques can be used to reduce the impact of DoS (denial-of-service) attacks on end host. This can also be mitigated by dropping the packets when it exceeds a particular threshold obtained by port stats.

INTRODUCTION:

Now-a-days increasing popularity of web over WAN has gained significant importance. Protecting these network infrastructures has become essential. SDN which is Software defined networking presents a logically centralized controlled network framework. Decoupling of Control plane and data plane distinguishes it from traditional network. Control and data plane are connected together with southbound protocols like OpenFlow where as control and data plane are connected together by northbound protocols like REST API. When new flow passes through a switch is not match with the flow rule installed in flow table then it is consider as table miss entry and data plane will ask controller to take necessary actions. Table miss entries consume resource in both control plane as well as data plane.

In our project, we address serious SDN-specific attacks i.e data-to-control plane saturation attack, which overloads the SDN network. In this attack, an attacker produces large amount of traffic (table-miss packet_in messages) . There are two ways to deal with this attack. One is attack specific and other is from a resource regulation perspective. First perspective includes attack on specific kinds of protocol behaviour like TYP SYN flood, ICMP flood, DNS flood etc. TCP SYN attacks are handle by TCP SYN cookies where as ICMP are handles by turning off ICMP echo reply. Second one is Resource control perspective. Resources could include network bandwidth, memory buffer, memory or CPU. Resource control and QOS are tied to each other. This reduces the opportunities for DoS attack to succeed.

Our solution is to implement QoS regulation techniques to reduce the impact of DoS attacks on end host. Along with this, our framework is efficient to reduce DoS attacks by maintaining constant communication bandwidth of the queues in OVS. Another way of mitigating this attack is to drop the packets when it crosses a certain threshold level.

PROJECT FLOW:

1) We create a topology in Mininet environment.
2) RYU controller is started and queue settings and rules are added to reserve a constant Bandwidth to queues.
3) The flow entries are then added to redirect the packets to queues.
4) Flood one of the hosts from other host using hping3 command.
5) Check whether the flooded host is reachable or not by pinging from other hosts in the network
6) Measuring the Bandwidth of the queues using Iperf Command.
7) Also mitigating the attack by dropping the excess packets using the port stats of the switch. This flow entry is added to drop all the packets obtained from the port.

SDN HUB:

SDN hub is an all-in-one pre-built tutorial VM, built by SDN Hub. This is a 64-bit Ubuntu 14.04 image (3GB) that has a number of SDN software and tools installed.
SDN software and tools installed are:

- SDNControllers: OpenDaylight, ONOS, RYU, Floodlight, Floodlight-OF1.3, POX, and Trema.
- Example code for a hub, L2 learning switch, traffic tap, and other applications
- Open vSwitch 2.3.0 with support for OpenFlow 1.2, 1.3 and 1.4, and LINC switch
- Mininet to create and run example topologies
- Pyretic
- Wireshark 1.12.1 with native support for OpenFlow parsing
- JDK 1.8, Eclipse Luna, and Maven 3.3.3

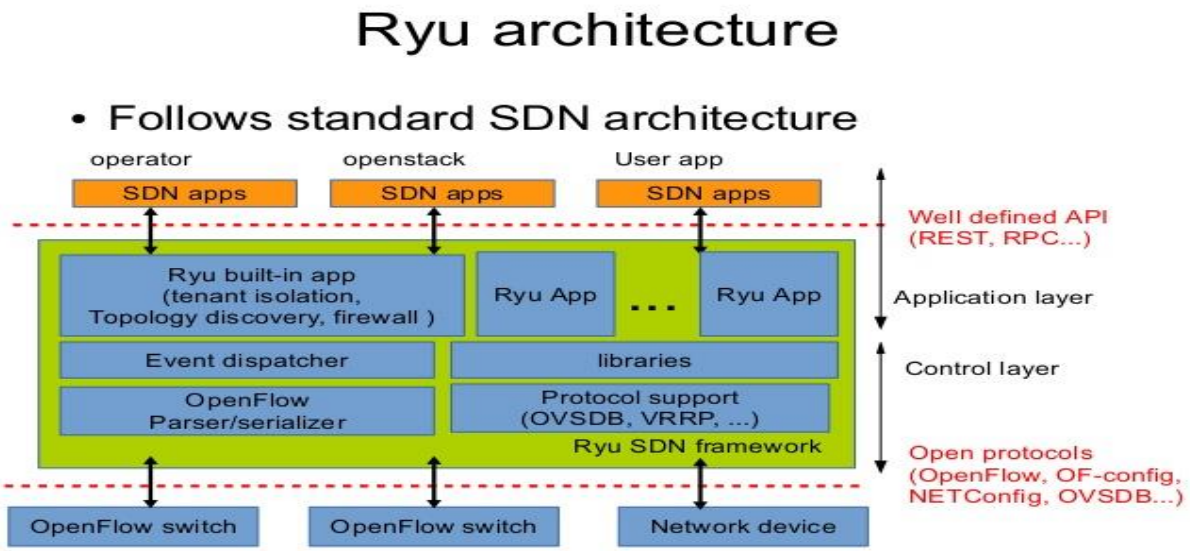
RYU:

Ryu Controller is an open, software-defined networking (SDN) Controller designed to increase the agility of the network by making it easy to manage and adapt how traffic is handled. In general, the SDN Controller is the brains of the SDN environment, communicating information down to the switches and routers with southbound APIs, and up to the applications and business logic with northbound APIs. The Ryu Controller is supported by NTT and is deployed in NTT cloud data centers as well.

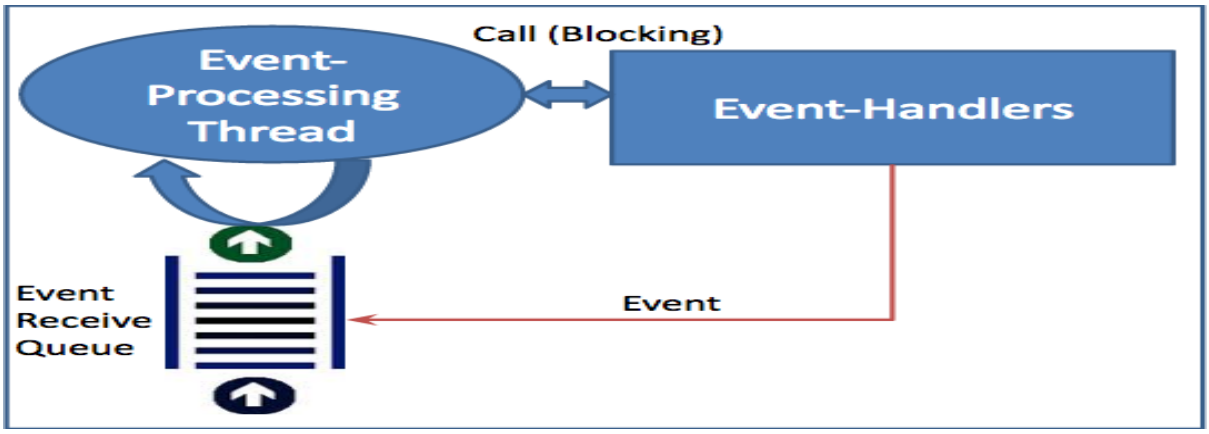
The Ryu Controller provides software components, with well-defined application program interfaces (APIs) that make it easy for developers to create new network management and control applications. This component approach helps organizations customize deployments to meet their specific needs; developers can quickly and easily modify existing components or implement their own to ensure the underlying network can meet the changing demands of their applications.

RYU supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc. About OpenFlow, RYU supports fully 1.0, 1.2, 1.3, 1.4, 1.5 and Nicira Extensions. RYU is fully written in Python and it is easy to introduce REST API.

RYU Architecture:



RYU Manager Process:



RYU Modules:

The Ryu code has six main components which are essentials while writing the application.

Those are as follows:

Controller

Base

App

Lib

Ofproto

Topology

Controller Module:

This module contains set of required files to deal with the OpenFlow functions such as packets from switches, handling network events, etc. The controller contains many important files such as controller.py, event.py, ofp_handler.py dpset.py, etc. Controller.py is the main component of the OpenFlow Controller. It handles connections from switches, generate and route events to appropriate entities like Ryu applications. For example, event.py defines the base of all event classes. The dpset.py file are defined for the switch to describe and operate the switch. This module contains the required set of files to handle OpenFlow functions. e.g packets from switches, generating flows, handling network events. It is used to observe events as well as act when an event occurs. This component also happens to have the base class for all the events in the entire controller.

Base Module:

This module contains the base class for Ryu applications. The app manager class which resides in app_manager.py is inherited by every Ryu application. It handles the app initialization and communication with the other components of the controller. The management is centered for Loading Ryu applications, Providing contexts to Ryu applications and Routing messages among them. This App is a collection of Contexts, event handlers and event observers. The lifetime of an application is decided by the number of events it must process. Based on that, it dispatches the event to many observers. These observers carry out further computation to manage the network.

Ryu is not just limited to a single application. It can execute multiple application with different features at the same time. The base module in this case provides context to each of these applications. The context is an integral part of the application and one can understand the purpose of the application from it.

App Module:

This Module contains the set of applications provided by the Ryu. Examples include simple_switch_13.py, simple_monitor.py, rest_qos.py. simple_switch_13.py performs basic routing functionalities. simple_monitor.py take note of statistics, the total number of bytes exchanged. rest_qos.py to set and access the queues of the switches, etc.

Lib Module:

This module contains set of packet libraries to parse different protocol headers and a library for ofconfig. For example: Take OVS bridge for example, the lib module contains the implementation of an OVS Bridge which allows the developer to handle the QoS parameters specific to that bridge without having to worry about the internal details of how it works.

Ofproto Module:
This module contains the OpenFlow protocol specific information and related parsers to support different versions of OpenFlow protocol. The details about which features from the
specific versions of OpenFlow are incorporated in the controller can be found here.

Topology Module:
This Module contains the code for performing topology discovery related to the OpenFlow switches and handles the information about ports, links, etc. associated with the OpenFlow Switches. It internally uses LLDP protocol. It defines the basic behavior of the ports and switches, the attributes which a port has, the attributes of a switch etc.

MININET:

Mininet is one the important part of our project. It is a network emulator which basically runs a collection of switches, routers hosts and links on a single linux. A mininet host just behaves like a real machine and program which one runs can send packets through ethernet with speed and delay as like in real world. Mininet virtual host switches and controller are real things which are created using software in linux environment rather than hardware. Use of mininet is increasing as one can run edit debug loop very quickly, can create custom topologies, run real programs, customize packet forwarding, share and replicate results. Creating topologies in mininet environment is easy as we need to write few lines of python code to create one. In our project we are creating custom topology using command :sudomn –custom "PATH" --topomytopo –controller=remote,protocols=OpenFlow13 -x
Here – topo is short name for topology constructor and –controller is for controller constructor. We have set open flow protocol version as 1.3. Here path is the address of topology file.

REST API:

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data. REST APIs are used for retrieving the switch stats and Updating the switch stats. This application helps you debug your application and get various statistics. This application supports Open Flow version 1.0, 1.2, 1.3, 1.4 and 1.5 ryu.app.ofctl_rest provides REST APIs for the RYU controller for retrieving the switch statistics.

QUEUE:

OpenFlow has supported queues for rate-limiting packets egress a switch port and for QoS Implementation. Queues are designed to provide a guarantee on the rate of flow of packets placed in the queue. As such, different queues at different rates can be used to prioritize "special" traffic over "ordinary" traffic.

HPING3:

This is a free packet generator and analyser for TCP/IP protocols and also one type of a tester for network security. HPING uses TCl language and packets are sent via binary/string representation.Hping 3 is command line which is also compatible with hping2 but scripting core is not limited to hping2 packet generation. We are using HPING 3 in order to create a DoS attack using UDP packets. Standard command used for this is : hping3 -c 10000 -d 120 -S -w 64 -p 21 –flood –rand-source www.xxxxx.com➔ hping3 is name of application binary, -c 10000 is no. of packets to send, -d 120 is size of each packet thatwas sent to target machine. -s is sending SYN packets only, -w 64 is the TCP window, -p 21 is destination port 21 , --flood is used for flooding, --rand-source using some random source ip (one can also use -a to hide hostnames), www.xxx.com is destination IP.

Command used is our project is :hping3 10.0.0.X –flood –udp. Here –udp states that it is flooding using udp packets.

TOPOLOGY:

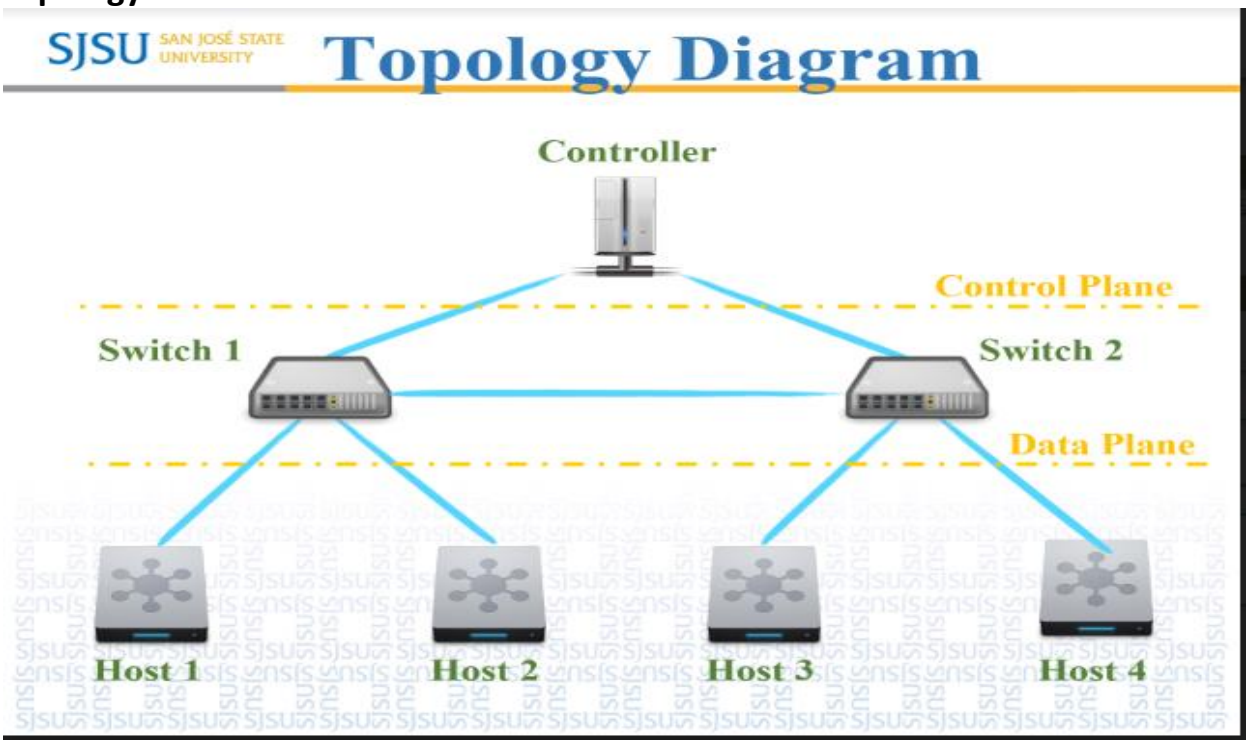
The word topology comes from the Greek words topos means place and logos means study. It is a description of any locality in terms of its layout. Topology is an arrangement of the elements (example: links, switches etc) in a communication network.

Topology Code:

```
from mininet.topo import Topo
class MyTopo( Topo ):
   "creating simple topology"

   def __init__( self ):
      "Creating custom topo"
      # Initialize topology
      Topo.__init__( self )
      #Adding hosts and switches
      leftHost = self.addHost( 'h1' )
      rightHost = self.addHost( 'h2' )
      leftHost1 = self.addHost( 'h3' )
      rightHost1 = self.addHost( 'h4' )
      leftSwitch = self.addSwitch( 's1' )
      rightSwitch = self.addSwitch( 's2' )
      # Add links
      self.addLink( leftHost, leftSwitch )
      self.addLink( leftHost1, leftSwitch )
      self.addLink( leftSwitch, rightSwitch )
```

```
        self.addLink( rightSwitch, rightHost )
        self.addLink( rightSwitch, rightHost1 )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Topology view:



As we can see in the above figure there are 4 hosts, 2 switches and a controller. Two hosts h1 and h2 are connected to switch s1, other two hosts h3 and h4 are connected to switch s2. These two switches are connected to one Controller (RYU Controller).

Result of Topology in Mininet:
The topology is simulated in mininet by using a following command.

Command: ***sudo mn --custom /home/ubuntu/mininet/custom/topology.py --topo mytopo --controller=remote, protocols=OpenFlow13 -x***

- mn in command - We can choose which controller we want
- --custom in command - creating a custom topology
- /home/ubuntu/mininet/custom/topology.py - file path
- --topo mytopo - to obtain a topology created by us, not the default one.
- --controller=remote - to use an external OpenFlow controller
- protocols=OpenFlow13 - to use OpenFlow version 1.3
- -x - starts the xterm as soon as we execute the command

```
ubuntu@sdnhubvm:~[00:31]$ sudo mn --custom /home/ubuntu/mininet/custom/topology.
py --topo mytopo --controller=remote,protocols=OpenFlow13 -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h3, s1) (s1, s2) (s2, h2) (s2, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Running terms on :0.0
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet>
```

Result after typing 'Pingall' command in mininet:

In the Image below we obtain following attributes:

1) Packet In
2) Datapath Id
3) Ethernet source address
4) Ethernet destination address
5) In Port

```
packet in 2 b2:28:4d:00:68:5d ff:ff:ff:ff:ff:ff 1
packet in 2 b2:28:4d:00:68:5d ff:ff:ff:ff:ff:ff 1
packet in 1 e2:07:99:17:ea:e8 b2:28:4d:00:68:5d 2
packet in 1 e2:07:99:17:ea:e8 b2:28:4d:00:68:5d 2
packet in 1 b2:28:4d:00:68:5d e2:07:99:17:ea:e8 1
packet in 1 b2:28:4d:00:68:5d ff:ff:ff:ff:ff:ff 1
packet in 2 b2:28:4d:00:68:5d ff:ff:ff:ff:ff:ff 1
packet in 2 fa:1c:5a:c7:20:2d b2:28:4d:00:68:5d 3
packet in 1 fa:1c:5a:c7:20:2d b2:28:4d:00:68:5d 3
```

CODE ANALYSIS:

If you wish to manage network devices like switch and router at our way, we need to write a RYU application. The job of application is to tell RYU how to manage these networking devices. Then RYU configures these devices using open flow protocol.
Writing Ryu application is just a matter of writing a python code. Also we can save the file with any name and at any location. In our project we have named it SW_TM4.py and saved it in ryu/ryu/apps in SDN hub

Libraries Used:

```
1 from ryu.base import app_manager
2 from ryu.controller import ofp_event
3 from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_DISPATCHER
4 from ryu.controller.handler import set_ev_cls
5 from ryu.ofproto import ofproto_v1_3
6 from ryu.lib.packet import packet
7 from ryu.lib.packet import ethernet
8 from operator import attrgetter
9 from ryu.lib import hub
10 import requests,json
```

Initialization:

```
class Switching_hub(app_manager.RyuApp):
  #OpenFlow protocol version is specified
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Switching_hub, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.datapaths = {}
        self.switches = set()
        self.dp1 = True
        self.dp2 = True
```

```
#thread is initialized to periodically issue a request
# hub.spawn() is used to create threads
  self.monitor_thread = hub.spawn(self._monitor)
```

First job for writing a ryu application is to define a subclass of RyuAppin order to run python code as ryu application. Here we have named it as Switching_hub. This switching_hub class is subclassing ryu.base.app_mamnager.ryuapp. this class registers the application and is instantiated by ryu-manager. When it is passed to ryu-manager, all ryuapp subclasses imported will be started.
Second line of code basically tells a list of versions for openflow which is required by controller application. If this line is not specified in our code then application would consider all the versions that ryu supports. Reason to specify the openflow version as 1.3 is that it supports the table_miss flow entries.
Now we initiate this subclass using init method. Self._ _init__ accepts number of keyword arguments as we used *args, **kwargs and then this passes to super class Ryu.app.__init__ method which tells us that superclass is initialized. Then we initialize MAC-to Port table as an instance variable. This mac to port is a 2-D dictionary with first key as DPID of switch and second key as MAC address. We have also initiated datapaths and switches along with mac – to – port table.
Hub.spawn is used to create threads. We have used traffic monitor code in our project. In _monitor ( ) function gives us statistical information about the registered switches. In this when datapath = MAIN_DISPATCHER the switch is registered as monitor target and when datapath= DEAD_DISPATCHER then the registration is deleted.
In thread function _monitor(), issuance of a statistical information acquisition request for the registered switch is repeated infinitely every 10 seconds.

Switch Features Handler

Here we set a decorator which tells ryu when it should be called. First argument of this indicates an event. There are different types of events. In our project we have mentioned OFP switch feature, OFP queuegetconfig reply, packetIn, OFP state change and Port stats reply.Job of decorator is to register the below switch handler function for the event type(switch feature) within the specified dispatcher(CONFIG_DISPATCHER). We have mentioned config dispatcher because in this phase ryu asks switch for its feature via feature request. The switch responds with a feature reply message. Message received by switch results in an event. In switch_feature_handler function datapath is an object representing the switch that sends the message. Ofproto is for the version for openflow protocol used for communication between controller and switch. Parser ➔ parser the packet. Datapath has many attributes and we only need id to Ovsdb has information about switch so to access this we need to set address. Now we need to create a flow entry so switch creates a match condition. Here match is an empty parser which means that flow entry matches any packet. Here action is a list which contains a parser.OFPActionOutput which forwards the matched packet to a specific port. Here iaction is controller which means that switch will send the match packet to the

controller via packet-in message. Now ofproto.OFPCML_NO_buffer is used so that it will instruct switch to not buffer the packet, but will send entire packet to the controller.
With the use of helper instance method self.add_flow we asks for a flow entry to be added to the switch. In above line flow entry is added to the datapath with priority of 0 along with match and action elements.

```
#decorator to obtain the switch features
   @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
   def switch_features_handler(self, ev):
      datapath = ev.msg.datapath
      ofproto = datapath.ofproto
      parser = datapath.ofproto_parser

      self.switches.add(datapath)
      dpid= str(datapath.id)
      dp_Str = "000000000000000"+dpid
      #ovsdb_addr is set to access OVSDB
      connection = "tcp:127.0.0.1:6632"
      print "PUT the Request",requests.put(url="http://localhost:8080/v1.0/conf/switches/"+ dp_Str
+"/ovsdb_addr",data=json.dumps(connection))
      print dpid
      # install table-miss flow entry
      # We specify NO BUFFER to max_len of the output action
      match = parser.OFPMatch()
      actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                        ofproto.OFPCML_NO_BUFFER)]
      self.add_flow(datapath, 0, match, actions)
```

Add Flow:

Above function is used to add flow entries with predefined option to reduce the code required elsewhere in controller. Here datapath is instance from ryu.controller.datapath library, priority
Ofproto and ofproto_parserare extracted from the datapath and it is assigned to ofproto and parser.'Inst' is a list created in order to construct the instruction. As we are working with openflow version 1.3 we need to add some instructions along with actions. In this inst list, parser.OFPinstructionActions is the instance which is set to ofproto.OFPIT_APPLY_ACTIONS provided in actions
Now the below if else condition is used to construct the flow mod messages. Here depending on whether buffer_id is present are absent the flow mod message is constructed. If buffer id is present then flow mod messages contains datapathbuffer_id priority match instruction. Else if buffer_id is absent then it is passed with same datapath, same priority, match and instruction mentioned above. In order to send the flow mod message, datapath.send_msg(mod) is mentioned.

```
#to add the flow entry into switch
   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:
         flow_mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                      priority=priority, match=match,
                                      instructions=inst, table_id=1)
      else:
         flow_mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                      match=match, instructions=inst, table_id=1)
      datapath.send_msg(flow_mod)
```

Packet in Handler:

```
#decorator to handle packet in messages
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # If you hit this you might want to increase
        # the "miss_send_length" of your switch
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("packet truncated: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        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_protocols(ethernet.ethernet)[0]

        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

        # mac address learn to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD
        actions = [parser.OFPActionOutput(out_port)]
```

Here we see that the handler is defined for ryu.controller. ofp_event.EventOFPPacketIn. This is called every time when a switch sends a packet to the controller that is when a

packet in event is fired in MAIN_DISPATCHER. This happens when the table-miss entry is matched in the switch. This method is called with the current event as ev.

The below code does the sanity check. We check the message length to ensure that the entire message was received by the switch, otherwise a log is made that the packet was truncated, which may cause issues routing the package.

```
if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                ev.msg.msg_len, ev.msg.total_len)
```

The below code is used to obtain the vital data. Several attributes are assigned to a local variable for more compact code. The in_port variable retrieved from msg.match dictionary. This information is used to learn the location of the devices on the network.

```
    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_protocols(ethernet.ethernet)
```

The below code is to learn the MAC address and associated port. Here the Mac_to_Port table for the DPID of the current switch is created if itsn't already existed. The packet info is logged and the Mac_to_port table is finally updated with source address of the packet associated with the port it arrived on. The self.mac_to_port is updated by using the dpid and src variables as keys and setting it to the in_port.

```
    dst = eth.dst
    src = eth.src
    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
    # mac address learn to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port
```

The below code is to obtain mac_to_port and packet destination lookup. This check is perfformed by checking if the destination is in the self.mac_to_port[dpid] keys. If it does then the out_port is set to the physical port at the location in the table. Otherwise, the out_port is set ofproto.OFPP_Flood. Then actions are set to a list with an instance parser.OFPActionOutputgiven out_port to direct the packet.

```
if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD
    actions = [parser.OFPActionOutput(out_port)]
```

In the below screen shot we see that the queues and the flow rules can be setup manually too without using any QOS API's.

```
#setup of queue rules manually for switch 1 & switch 2
if   dpid==1 and src=="00:00:00:00:00:01":
    actions.append(parser.OFPActionSetQueue(1))
elif dpid==1 and src=="00:00:00:00:00:02":
    actions.append(parser.OFPActionSetQueue(0))
elif dpid==2 and dst=="00:00:00:00:00:01":
    actions.append(parser.OFPActionSetQueue(1))
elif dpid==2 and dst=="00:00:00:00:00:02":
    actions.append(parser.OFPActionSetQueue(0))
```

The below code is to set the bandwidth rate for queues 0 and 1 for both switches. In this code we install the flows to avoid packet_in messages from the switch. We define the queues for switches.

If the destination MAC address is known then the controller application needs to add a new flow entry to the switch so the packet can be directly forwarded rather than depending on the controller to forward. Therefore the check is porformed to ensure that the out_port is not set to ofproto.OFPP_FLOOD, which says that the destination physical port is known. The match conditions for a new flow are specified by setting match to an instance of parser.OFPMatch with arguments as in_port,eth_dst and eth_src. We see the max_rate and min_rate is set for two queues of port s1-eth1 where host h1 is connected and of port s1-eth2 where host h2 is connected. This is set by sending a request using the HTTP Post method. Then the queue request is sent for the datapath Id. The QOS setup for the switch s1 is done and then the flow entry is added to redirect any packet with UDP protocol to destination host h3 with IP 10.0.0.3 should be directed to queues 0 and 1 and the priority is set to 100 so this entry will be executed first. This is to prevent the host h3 from being flooded. By this we can see that the host h3 will always be available even if someone tries to attack by sending multiple UDP packets from different source IP address.

```
    # installing flows to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)

    #defining queues for switch 1 in topology
        if(dpid==1 and self.dp1):
            self.dp1=False
            print("Queues Set Up for",dpid)

            data={"port_name": "s1-eth1","max_rate": "1000000", "queues":[{"max_rate":"500000"}, {"min_rate": "800000"}]}
            url="http://localhost:8080/qos/queue/0000000000000001"
            print "Post the Request",requests.post(url=url,data=json.dumps(data))

            req = parser.OFPQueueGetConfigRequest(datapath, 1);
            print "Queue Req sent for ID",datapath.id
            datapath.send_msg(req)

            data={"port_name": "s1-eth2","max_rate": "1000000","queues": [{"max_rate":"300000"}, {"min_rate": "800000"}]}
            url="http://localhost:8080/qos/queue/0000000000000001"
            print "Post the Request",requests.post(url=url,data=json.dumps(data))

            req = parser.OFPQueueGetConfigRequest(datapath, 1);
            print "Queue Req sent for ID",datapath.id
            datapath.send_msg(req)

            #qos setup for switch s1,dest h3 flow entry
            url="http://127.0.0.1:8080/qos/rules/0000000000000001"
            data ={"match": {"nw_dst":"10.0.0.3","nw_proto":"UDP"},"actions":{"queue":0},"priority":100}
            print "Post the request",requests.post(url=url,data=json.dumps(data))
            print "GET Request",requests.get(url=url)

            #qos setup for switch s1,dest h3 flow entry
            url="http://127.0.0.1:8080/qos/rules/0000000000000001"
            data ={"match": {"nw_dst":"10.0.0.3","nw_proto":"UDP"},"actions":{"queue":1},"priority":100}
            print "Post the Request",requests.post(url=url,data=json.dumps(data))
            print "GET Request",requests.get(url=url)
```

The below code is same as explained for switch s1. Here we define it for the Switch s2.

```
    #defining queues for switch 2 in topology
      if(dpid==2 and self.dp2):
            self.dp2=False
            print("Queues Set Up for",dpid)

            data={"port_name":"s2-eth1","max_rate": "1000000", "queues": [{"max_rate":"500000"}, {"min_rate": "800000"}]}
            url="http://localhost:8080/qos/queue/0000000000000002"
            print "Post the Request",requests.post(url=url,data=json.dumps(data))

            req =parser.OFPQueueGetConfigRequest(datapath, 2);
            print "Queue Req sent for ID",datapath.id
            datapath.send_msg(req)

            data={"port_name":"s2-eth2","max_rate": "1000000", "queues": [{"max_rate":"200000"}, {"min_rate": "800000"}]}
            url="http://localhost:8080/qos/queue/0000000000000002"
            print "Post the request",requests.post(url=url,data=json.dumps(data))

            req = parser.OFPQueueGetConfigRequest(datapath, 2);
            print "Queue Req sent for ID",datapath.id
            datapath.send_msg(req)

            #qos setup for switch s2,dest h3 flow entry
            url="http://127.0.0.1:8080/qos/rules/0000000000000002"
            data = {"match":{"nw_dst":"10.0.0.3","nw_proto":"UDP"},"actions":{"queue":0},"priority":100}
            print "Post the Request",requests.post(url=url,data=json.dumps(data))
            print "GET Request",requests.get(url=url)

            #qos setup for switch s2, dest h3 flow entry
            url="http://127.0.0.1:8080/qos/rules/0000000000000002"
            data = {"match": {"nw_dst":"10.0.0.3","nw_proto":"UDP"},"actions":{"queue":1},"priority":100}
            print "Post the Request",requests.post(url=url,data=json.dumps(data))
            print "GET Request",requests.get(url=url)
```

```
            # verify if we have a valid buffer_id
            # Valid buffer id is used to avoid sending flow_mod &
            # packet_out messages

            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                self.add_flow(datapath, 2, match, actions, msg.buffer_id)
                return
            else:
                self.add_flow(datapath, 2, match, actions)

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

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

If the packet_in message contains buffer_id then the switch has buffered the packet. Therefore the flow entry must be added while referencing that buffer, causing the switch to immediately forward the buffered packet while the flow entry is added to the switch. A priority is set higher. If packet is not buffered then the normal flow entry is added without the buffer Id.

```
# verify if we have a valid buffer_id
        # Valid buffer id is used to avoid sending flow_mod &
        # packet_out messages

        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
           self.add_flow(datapath, 2, match, actions, msg.buffer_id)
           return
        else:
           self.add_flow(datapath, 2, match, actions)
```

The below code is executed when, either the destination is unknown or a buffer was not specified in the packet_in message. In either case the packet must be sent. The data argument for the later parser.OFPPacketOut call is prepared by setting data to None. If the message doesn't refer a buffer, then the switch did not buffer the packet so the data in the packet_in message sent to the controller must be supplied in the packet-out message. The parser.OFPPacketOut message is prepared with the variables prepared so far. The in_port argument is provided so the switch knows what port the packet should not be forwarded to, if the out_port is set to OFPP_FLOOD. The actions are then prepared both for this packet-out message and for the added flow entry, if one was added.

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

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

State Change Handler:

The below event is called for detecting the connection and disconnection of the switches to the controller that is to manage the state change handling. During this event the state change handler is called by passing the event ev. Here the MAIN DISPATCHER is used for the normal status check that is when the switch is connected to the controller. The DEAD DISPATCHER is used delete the switch registration when any un-correctable error occurs.

```
#event is used for detecting connection and disconnection
    @set_ev_cls(ofp_event.EventOFPStateChange,[MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        #switch is registered as the monitor target
        if ev.state == MAIN_DISPATCHER:
            if datapath.id not in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        #switch registration is deleted
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x', datapath.id)
                del self.datapaths[datapath.id]
```

Monitor and Request stats

The monitor method is called when we want to obtain the statistical info about the registered switch infinitely every 5 seconds. The request_stats method is called to handle the request sent by the controller to switch to obtain the switch statistics to perform certain operations or for any experimental purpose.

```
#to obtain statistical info about registered switch infinitely every 5 seconds
    def _monitor(self):
        while True:
            for dp in self.datapaths.values():
                self._request_stats(dp)
            hub.sleep(5)

#to request the port stats
    def _request_stats(self, datapath):
        self.logger.debug('send stats request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
        datapath.send_msg(req)
```

Port Stats Reply Handler

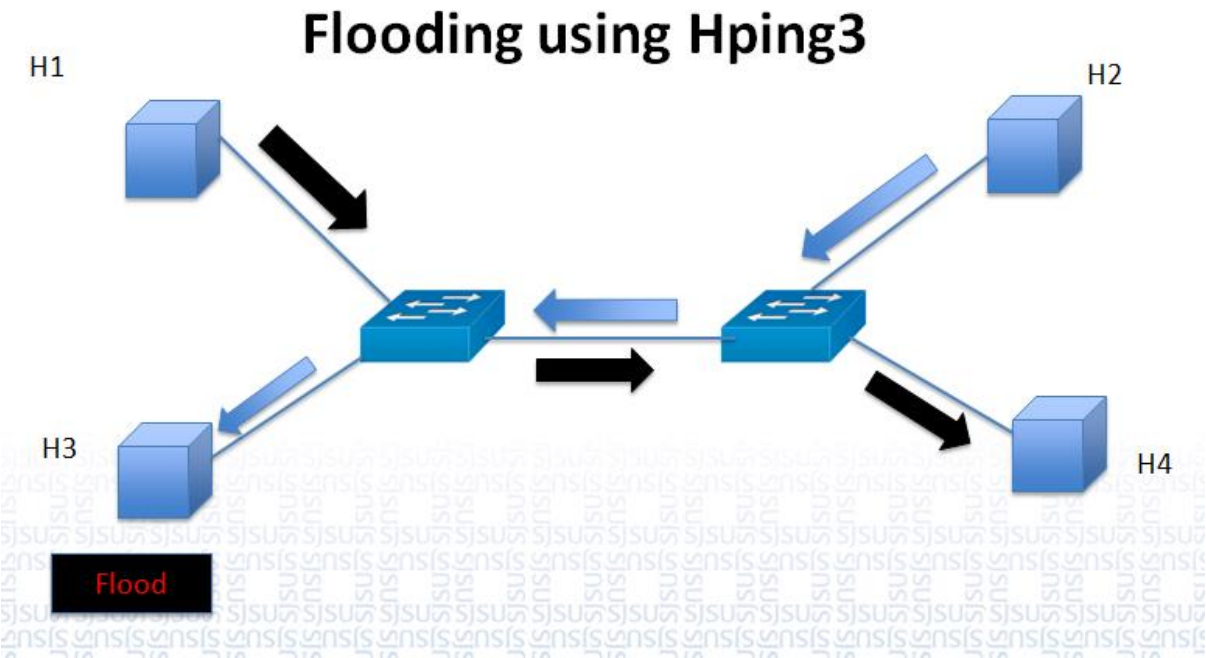
This decorator is called to handle the port stats reply obtained by the switch. Here we see the port stats reply about queue is handled by the event ofp_event.EventOFPQueueGetConfigReply. Here we obtain the datapath response for the queue.

```
#this decorator is to handle the stats reply about queue
    @set_ev_cls(ofp_event.EventOFPQueueGetConfigReply, MAIN_DISPATCHER)
    def port_stats_reply_handler(self, ev):
        print "datapath response for queue",(ev.msg.queues),ev.msg.datapath
```

Here we see the port stats reply about queue is handled by the event ofp_event.EventOFPPortStatsReply under MAIN DISPATCHER. Here we obtain the received bytes, packets and error count and transmitted bytes, packets and error count. We used this stats to mitigate the UDP attacking by dropping the packets if it exceeds than particular threshold. Here when the number of packets exceed then the action is set to drop with priority 10. Any packet to the port is then dropped. This flow entry is added to the switch by the controller.

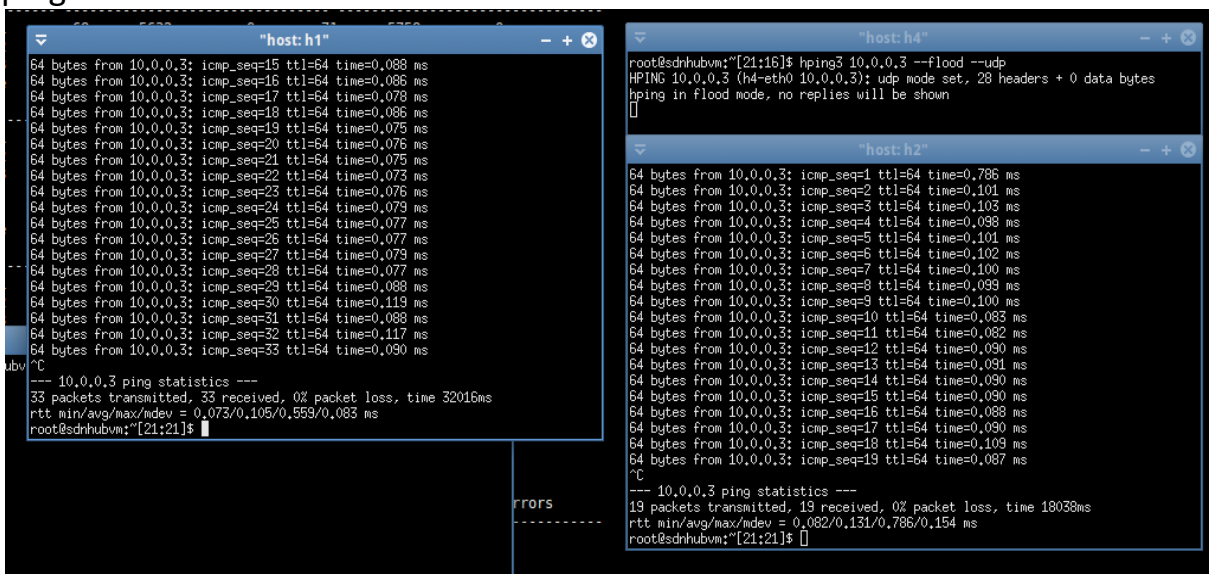
```
#handles the received response from the switch about it's port status
    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        body = ev.msg.body

        self.logger.info('datapath         port          '
                         'rx-pkts    rx-bytes   rx-errors '
                         'tx-pkts    tx-bytes   tx-errors ')
        self.logger.info('---------------------------------- '
                         '---------------------------------- '
                         '---------------------------------- ')

        for stat in sorted(body, key=attrgetter('port_no')):
            self.logger.info('%016x  %8x  %8d  %8d  %8d  %8d  %8d  %8d ',
                             ev.msg.datapath.id,  stat.port_no,
                             stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                             stat.tx_packets, stat.tx_bytes, stat.tx_errors)

            #condition to drop the packets from that port
            if (((stat.rx_packets-stat.tx_packets) > 20000) or ((stat.tx_packets-stat.rx_packets) > 20000)):
                print("Flooding Recognised....")
                print("Packet dropped")
                msg = ev.msg
                datapath = msg.datapath
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                action =[]
                match = parser.OFPMatch(in_port=stat.port_no)
                self.add_flow(datapath, 10 , match, action)
```

RESULT:



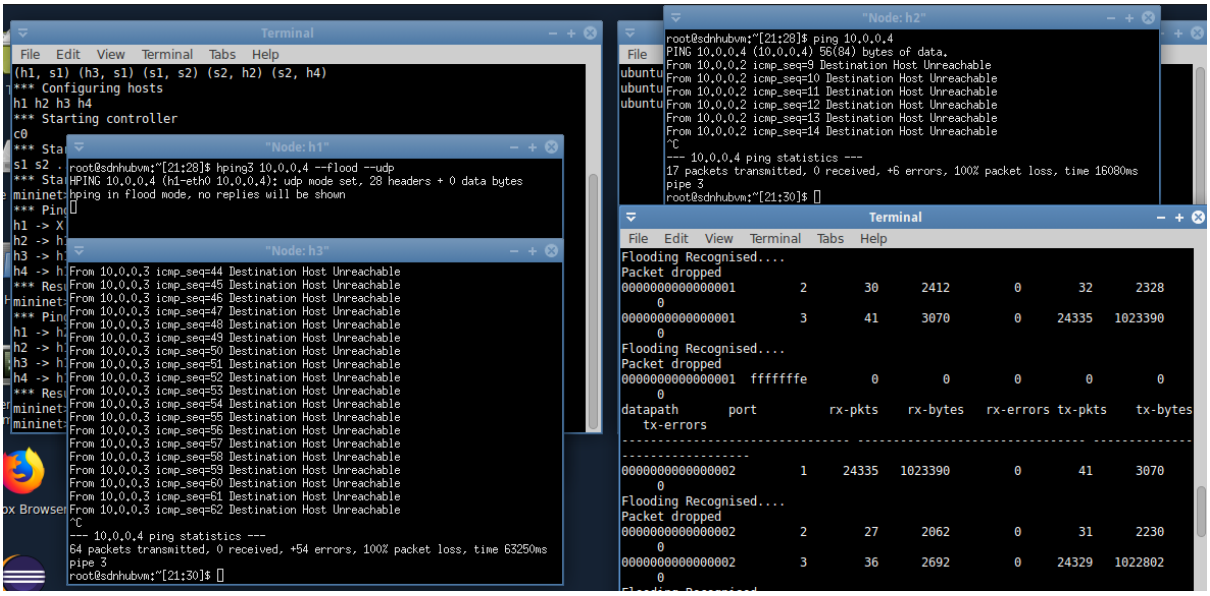
Test Scenario 1:

Here we flood the host h3 with UDP packets from host h2 using hping3 command. Whenever we send the UDP packets to destination host h3, the flow entry is made in such a way that the packets are directed to queue instead of directly passing it to the controller. This qos rules are added in both switch s1 and switch s2. Therefore, the host h3 is still available when we try to ping from host h1 and host h4.

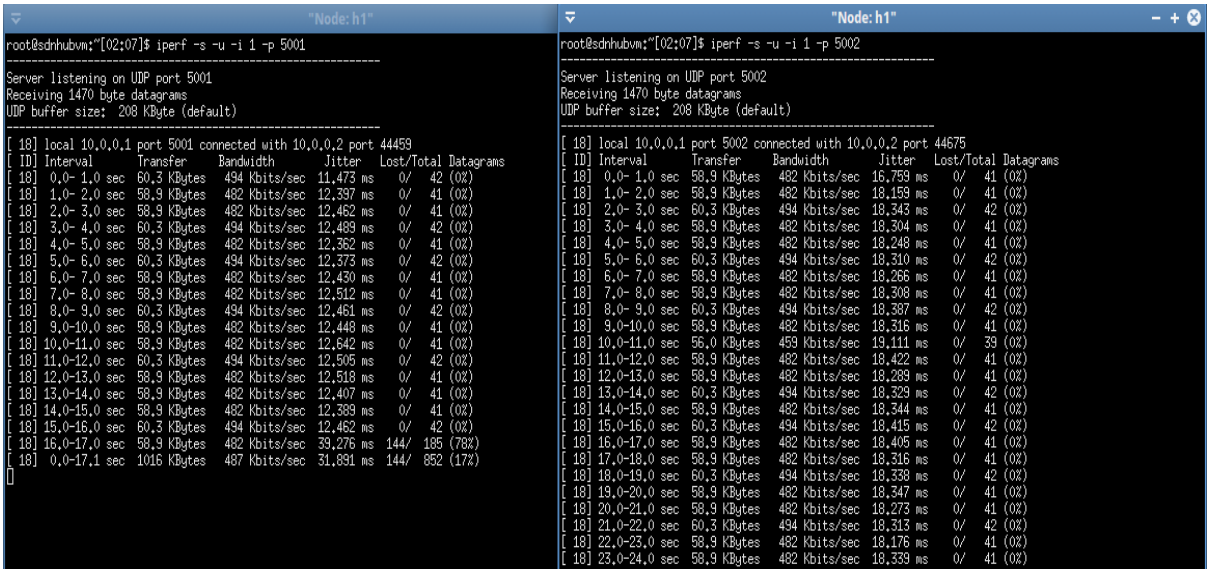


Test Scenario 2:

Here we flood the host h4 with UDP packets from host h1 using hping3 command. When we send the UDP packets to destination host h4, the flow entry is not made to direct the packets to queue instead it directly passes it to the controller. Therefore, the host h4 becomes unreachable when we try to ping from host h2 and host h3.



Measurement of Bandwidth of queues using Iperf command



CONCLUSION:

SDN is becoming an efficient one among the networking technology. But the challenges on the security side prevent the world from implementing the technology worldwide. Resource control and QoS are closely tied to each other. QoS regulation techniques can be used to reduce the impact of DoS attacks on end host. Our prototype provides an efficient way to reduce DoS attacks by maintaining constant communication bandwidth of the queues in OVS. This can also be mitigated by dropping the packets when it exceeds a particular threshold.

FUTURE SCOPE:

It can temporarily redirect the packets to cache and perform symbolic execution to analyze fake or normal packets and install flow rules for packets in cache and redirect back to switch. This cache can be controlled by using rate limiting and round robin scheduling algorithms.

REFERENCES:

1. https://ieeexplore.ieee.org/abstract/document/7266854/
2. https://osrg.github.io/ryubook/en/html/switching_hub.htmlNbkmsbcksb
3. https://osrg.github.io/ryu-book/en/html/traffic_monitor.html
4. https://osrg.github.io/ryu-book/en/html/rest_qos.html
5. **Lecture slides - CMPE210 by Prof. Younghee Park**