SimTP – Simple Tunneling Protocol

Project Website:

https://sites.google.com/ncsu.edu/tunnelprotocol

Instructor: Dr Rudra Dutta

Davis Polly Pynadath [dppynada@ncsu.edu]

Jalandhar Singh [jsingh8@ncsu.edu]

Suhaskrishna Gopalkrishna [sgopalk@ncsu.edu]

Contents

1 Introduction

1.1 Problem Statement

In an SDN environment, the SDN controller discovers the topology using LLDP and is aware where the host and its respective port on OVS(Open vSwitch) is present. By using a routing application module on the controller, it can determine the path from a source to destination and adds the respective flows onto the OVS.

A very common setting in many SDN environments would be as the figure below:

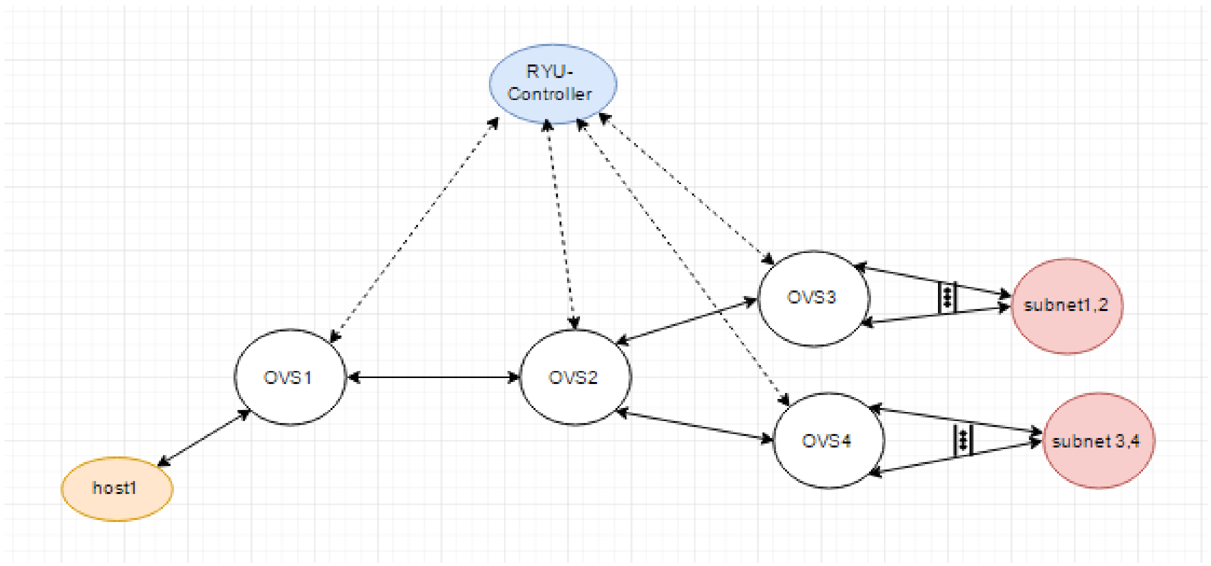


Fig-1

From the above fig, we notice many flows in the network follow the same path. Consider that the OVS-SW-1, SW-2, SW-3 and SW-4 are acting as routers and are forwarding the packets from source to destination based on their Flow table. Here, there would be multiple flows for each destination subnet prefix on intermediate OVS(SW-2), even though the packets are either forwarded through one of the two links. When a network scales, this could result in exhausting the Flow Table Memory. Another problem is that the match criteria is the Destination Prefix(IP) in the FlowTable and it would consume time because of 32 bit matching. There is also the additional overhead to the admin to manually configure the flows using OpenFlow control commands.

A possible solution for the problem is to have a match criteria having lesser than 32 bit for forwarding the frames from the source to destination and thereby improving the lookup time. Also, optimizing the number of flows entries i.e. on the Intermediary SW2 we should ideally have just 2 flows instead of flow to each and every destination, resulting in saving the FlowTable memory as the destination is the same for all the three flows. We could also write an application on the controller to automatically configure the flows and hence in this manner assist the admin.

1.2. Proposed Solution

The solution proposed for the problem is a new protocol – Simple Tunneling Protocol (SimTP) at layer L-2.5 which forwards packets based on a unique identifier (reusing the 20bit MPLS Label) assigned to each destination prefix called the Tunnel ID(TID). Using a 20 bit lookup rather a 32 bit is expected to improve performance. On a Note- by using the MPLS Label for the Tunnel ID we are only forwarding using the tunnel ID and there is no label switching involved.

Now in Fig-1 we could essentially encapsulate the packets with the tunnel ID at OVS1 so that at OVS2 we would only need two flows to forward packets. The match criteria now are not the destination prefix but rather the Tunnel-ID.

The protocol encapsulates specific flows based on the rule provided by the user. The rule can be a match-criteria to any field in the IP packet, but in the scope of the project, only destination address in the IP field will be considered. Now the forwarding of the packets which match the flow will be done using Tunnel ID, rather than the Destination IP. Hence, the forwarding becomes more based on the output port of the end OVS node rather than IP destination based.

Note: Many generic and wide ranged tunneling protocols like GRE Tunneling Protocol already exists. However, implementation of such a complex and wide ranged protocol is out of scope for the project, and hence a simpler alternative specific to the problem is proposed.

1.3. Platform

- ExoGeni
- OVS Open-Flow Version 1.3
- RYU SDN Controller – Open-Flow Version - 1.3
- Fedora 22 with kernel version-4.0.4

1.4. Project Area

Introducing a new protocol- SimTP protocol that will push the Tunnel ID onto the packet at ingress based on the destination and the forwarding of packets is based on this very Tunnel ID. This Tunnel ID will be popped at egress before it reaches the destination.

2. Design

2.1. High Level Design:

The HighLevel Design will get covered in two sections

1. Call Flow Steps
2. Call Flow Diagram

2.1.1. Call Flow Steps

• First user will do the configuration using curl or REST API like

```
curl -X POST -d ‘{ "tunnel_id":20, “tunnel_outport”:2, tunnel_type:”lsr”}’ http://localhost:8080/router/0000000000000001

curl -X POST -d ‘{ "prefix":"10.0.1.0/24", "tunnel_id":20}’ http://localhost:8080/router/0000000000000001
```

The curl command will send this data to the SimTP application over the WebSocket.

• Now SimTP Application will parse this information and then stores this information into internal data structure like:

Prefix	**Tunnel-ID**
10.0.1.0/24	20

Tunnel-ID	**Tunnel_type**	**Tunnel_outport**
20	Lsr	1

Note: LSR/LFR stands for “Label Forwarding Router”. (Intermediate Router)
LER stands for “Label Edge Router”. (Ingress or Egress Router)

• After storing the information, controller will push the flow to corresponding routers using Flow_Mod Message.

• Now If the one host tries to ping the another then it will trigger an ARP request to discover the gateway MAC address.

• OVS switch kernel module will checks the ether_type value.
If ether_type is MPLS then it will match the mpls_label in the flow entry and forwards the packet
Else it will forwards the packet to the controller.

Note: we are reusing MPLS ether_type and label for our implementation.

• As controller gets packet-in message, it will find the gateway destination MAC address and then sends back the packet-out message with dst_mac address and actions as “ARP REPLY”.

• Now OVS will forward the the packet_out to the host and once host receives the ARP reply, it will start sending an ICMP packet.

• The packet enters the default gateway (LER or ingress router) of the host.

• The packet matches one of the flows and is sent to the controller.

• The controller chooses a tunnel_id based on the destination IP.

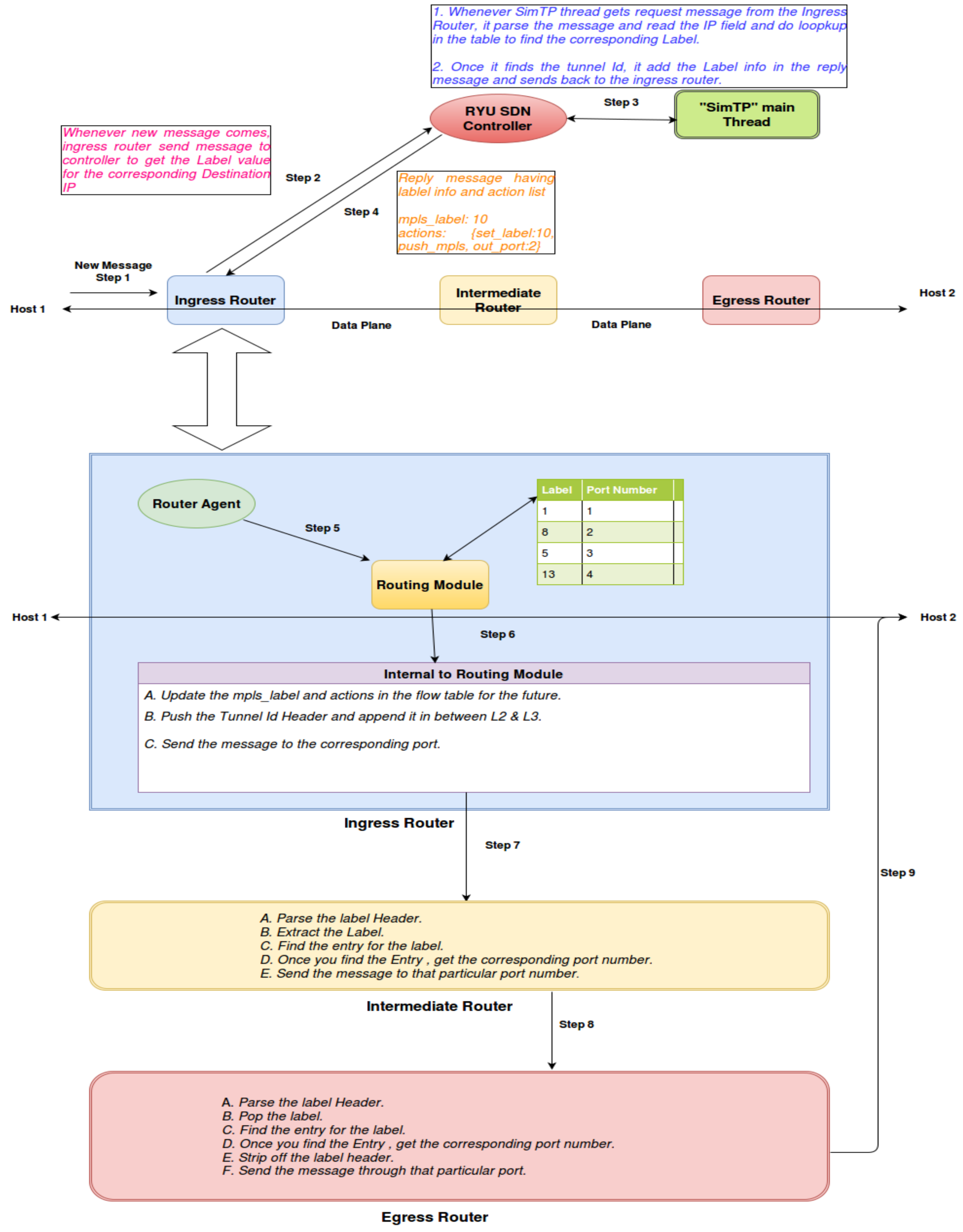
• The controller sends the flow-Mod message to OVS with match criteria “source and destination IP address” and actions as “push a MPLS label, set Soruce and Destination MAC for the next hop, Output port”.

As OVS receives flow-mod message, it creates a new flow entry into the flow table and sends the message based on the actions.

• Once LFR receives the packets, it will check for ether_type and in case of MPLS, it parses the Label Header and based on the label, it will do look up and get the out_port value. Then it will forward the message to Egress Router.

• Similarly once Egress router receives message, and it will check for the ether_type and based on the label lookup, it will forward the packet to the receptive out_port.

2.1.2. Call Flow Diagram



1. Whenever SimTP thread gets request message from the Ingress Router, it parse the message and read the IP field and do loopkup in the table to find the corresponding Label.

2. Once it finds the tunnel Id, it add the Label info in the reply message and sends back to the ingress router.

Whenever new message comes, ingress router send message to controller to get the Label value for the corresponding Destination IP

Reply message having lablel info and action list

mpls_label: 10
actions: {set_label:10, push_mpls, out_port:2}

Label	Port Number
1	1
8	2
5	3
13	4

Internal to Routing Module

A. Update the mpls_label and actions in the flow table for the future.
B. Push the Tunnel Id Header and append it in between L2 & L3.
C. Send the message to the corresponding port.

A. Parse the label Header.
B. Extract the Label.
C. Find the entry for the label.
D. Once you find the Entry , get the corresponding port number.
E. Send the message to that particular port number.

A. Parse the label Header.
B. Pop the label.
C. Find the entry for the label.
D. Once you find the Entry , get the corresponding port number.
E. Strip off the label header.
F. Send the message through that particular port.

2.2. Low-Level Design

Basically, there are two major Modules wr.r.t ovs and controller.

1. Application Development on top of RYU for SimTIP protocol
2. OVS Kernel Module

The Low-Level design will have the class level details and the flow diagrams to provide the call flow details for each module.

2.2.1. Application Development on top of RYU for SimTP protocol

To reduce the complexity, it takes advantage of Object Oriented Programming to encapsulate the different stakeholders of an openflow controller.

The main classes are the following

• **RestRouterAPI**: It's the main class, the one that inherits from RyuApp. This class contains four event handlers for the following events: EventOFPPacketIn, EventOFPFlowStatsReply, EventOFPStatsReply and EventDP which is a non-OpenFlow event contaied in the ryu.controller.dpset module.The OpenFlow controller part of Ryu automatically decodes OpenFlow messages received from switches and send these events to Ryu applications which expressed an interest using ryu.controller.handler.set_ev_cls.

• **RouterController**: This class inherits from ryu.controller.ControllerBase class, and aims to extend its functionality, focused on the scenario of an IP router. Implements the methods to add, delete and access to the routers listed by the Ryu OpenFlow Controller. Also, implements the methods for REST commands handling to set/get/delete the user configured router specific information.

• **Router**: Dictionary class. Contains at least one VlanRouter object. An instance of this class corresponds directly to one of the OpenFlow Switches of the network. This class implements several methods for writting, reading or deleting data on the VlanRouters, such as routes or Vlan tags.

• **VlanRouter**: This class implements the intelligence of an Vlan Router for a Vlan tag (vlan_id). If it's the case, as it is ours, that no Vlan tags are being used, the vlan_id is zero. This is the most important class from control perspective. It contains information about the router's IP addresses, static and default routes, and implements all the methods for handling that data. Also, implements a packetin method that discriminates packets by tipe (ARP, ICMP or TCP/UDP), and all the methods to handle those kinds of packets.

• **OfCtl**: This class implements methods to write and delete flows with different OpenFlow versions. Also, implements methods to send ARP and ICMP messages.

• **Data structure classes:** Classes that encapsulate IP addresses, routing tables, Ports, etc. These classes are Route, RoutingTable, Address, AddressData, TunnelData, Port and PortData. The module also contains several useful functions such as string to number transformations or mask application to IP addresses.

AddressData	Stores the list of IP Addresses of each port of the switch
PortData	List of {port_no, MAC Address} of each port of the switch where port number is the key.

TunnelData	List of the { tunnel_id, tunnel_type, tunnel_port } where tunnel_id is a key.
PrefixData	List of the {prefix, tunnel_id} where prefix is the key.

Next diagram will be showing the relationship between the classes and it will list down the data members and Methods.

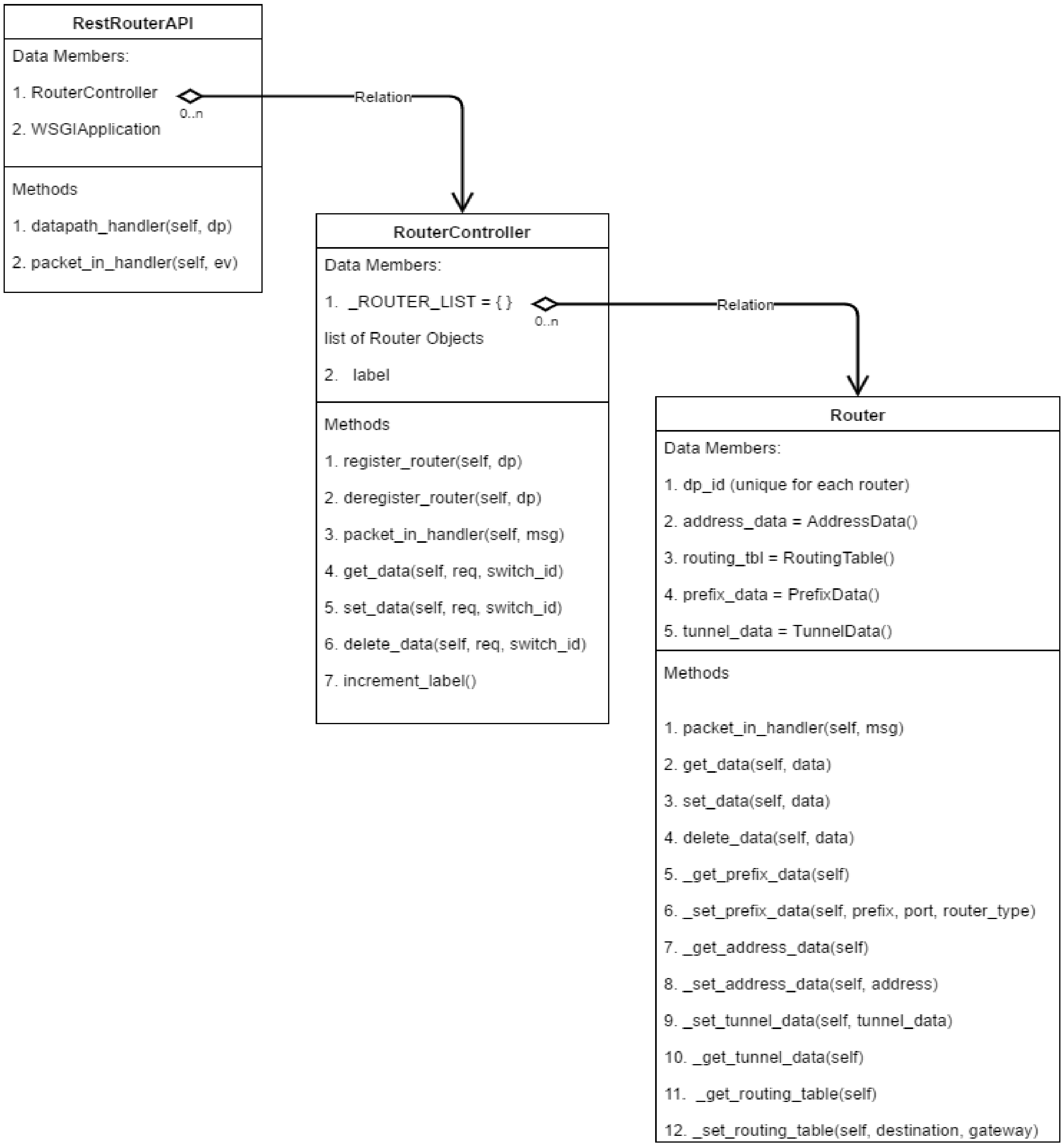

RestRouterAPI
Data Members:
1. RouterController
2. WSGIApplication
Methods
1. datapath_handler(self, dp)
2. packet_in_handler(self, ev)
0..n
Relation
RouterController
Data Members:
1. _ROUTER_LIST = { }
list of Router Objects
2. label
Methods
1. register_router(self, dp)
2. deregister_router(self, dp)
3. packet_in_handler(self, msg)
4. get_data(self, req, switch_id)
5. set_data(self, req, switch_id)
6. delete_data(self, req, switch_id)
7. increment_label()
0..n
Relation
Router
Data Members:
1. dp_id (unique for each router)
2. address_data = AddressData()
3. routing_tbl = RoutingTable()
4. prefix_data = PrefixData()
5. tunnel_data = TunnelData()
Methods
1. packet_in_handler(self, msg)
2. get_data(self, data)
3. set_data(self, data)
4. delete_data(self, data)
5. _get_prefix_data(self)
6. _set_prefix_data(self, prefix, port, router_type)
7. _get_address_data(self)
8. _set_address_data(self, address)
9. _set_tunnel_data(self, tunnel_data)
10. _get_tunnel_data(self)
11. _get_routing_table(self)
12. _set_routing_table(self, destination, gateway)

Figure: class diagram

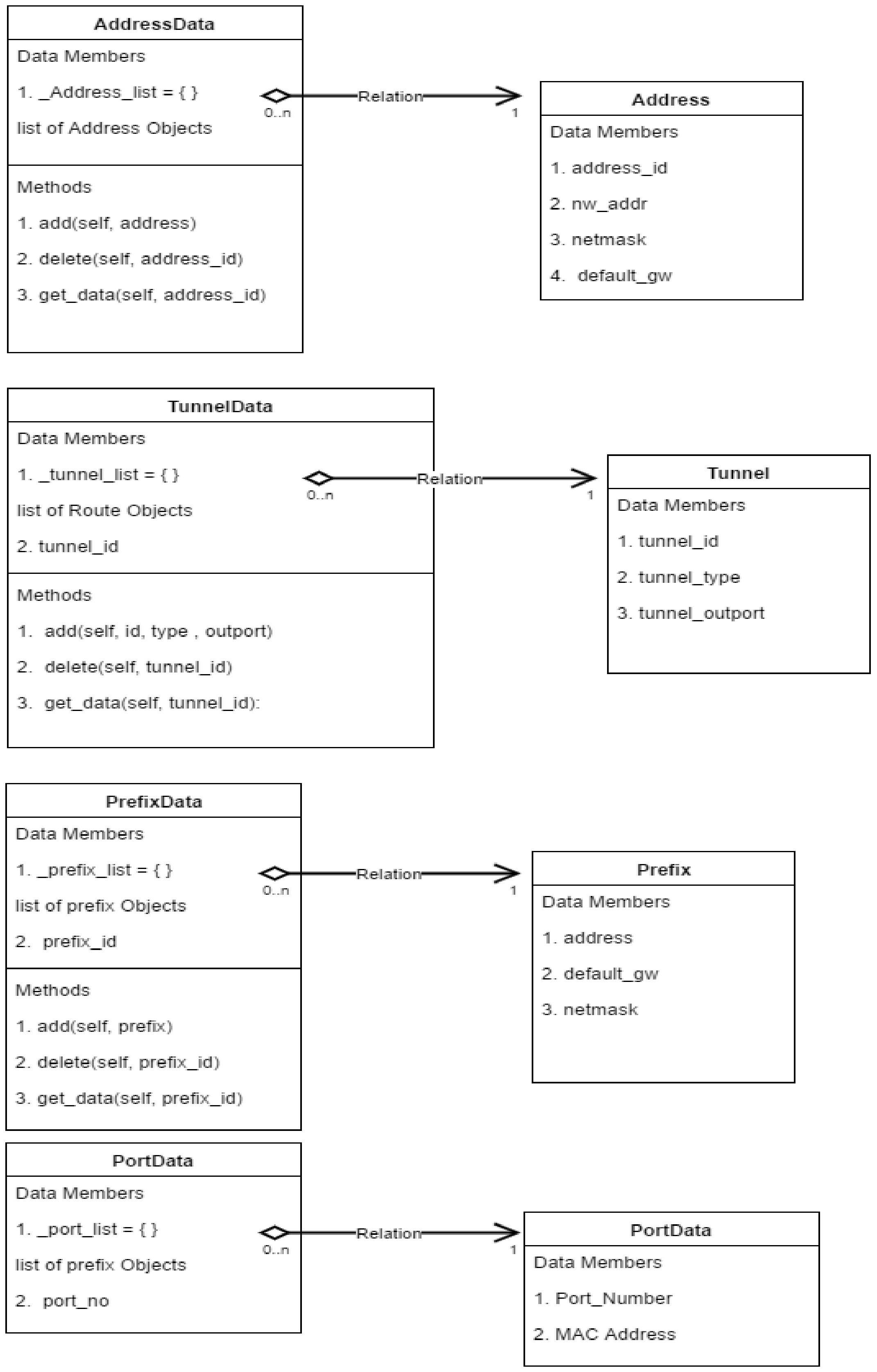


Figure: class diagram.

The application has three main components

1. Peer or OVS registration with the SimTP Application

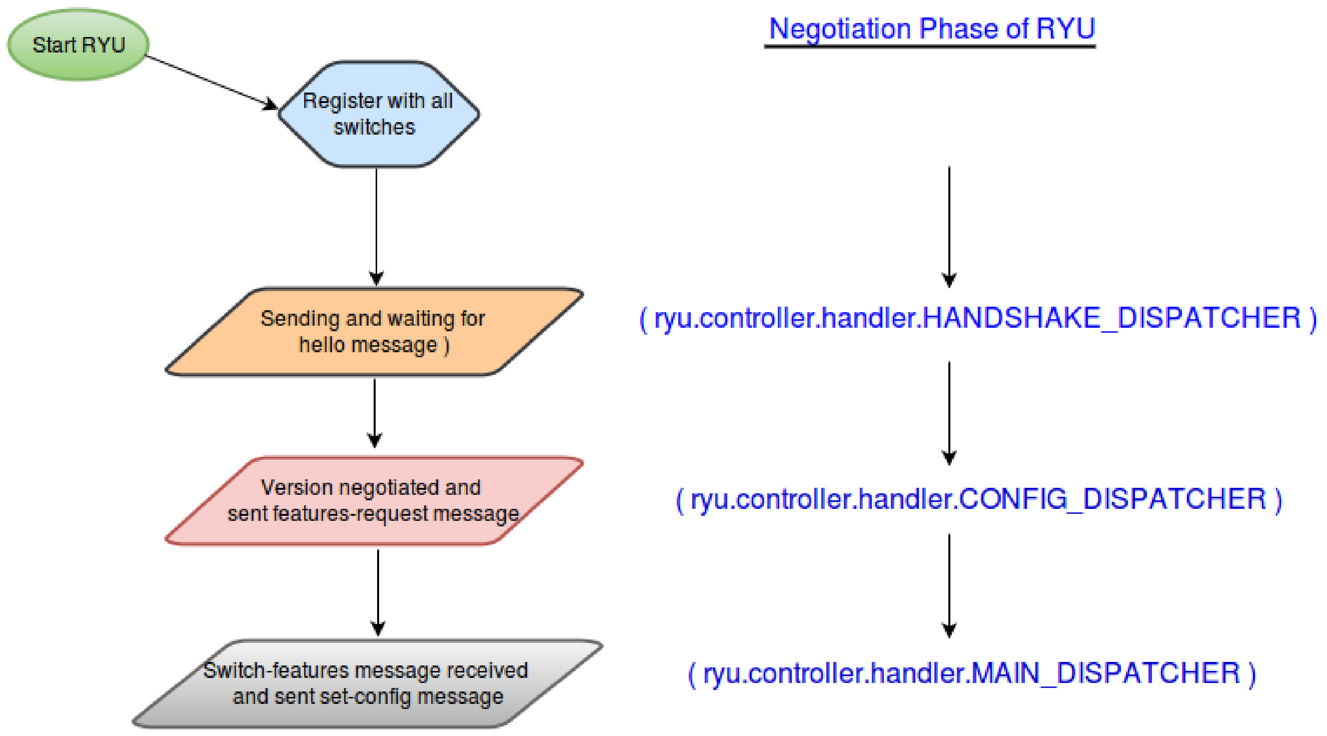
2. User configuration using REST API

3. SimTP Ryu Application

2.2.1.1. Peer or OVS registration with the SimTP Application

To run the application, user must run the command on controller

```
ryu-manager --verbose ryu/app/rest_simTP.py
```


Start RYU
Register with all switches
Sending and waiting for hello message)
Version negotiated and sent features-request message
Switch-features message received and sent set-config message
Negotiation Phase of RYU
(ryu.controller.handler.HANDSHAKE_DISPATCHER)
(ryu.controller.handler.CONFIG_DISPATCHER)
(ryu.controller.handler.MAIN_DISPATCHER)

SimTP Application goes into different state while connecting to the switches. Initially during “hello” message exchanges, it will be in the HANDSHAKE state. Once handshake is done, ryu-manager main thread raises an EventDP event to the SimTP Application.

While SimTP Application initialization, SimTP Application registers for the event “EventDP”. And “ryu.controller.dpset.EventDP” is an event class to notify connect/disconnect of a switch. An instance has at least the following attributes.

Attribute	Description
dp	A ryu.controller.controller.Datapath instance of the switch
enter	True when the switch connected to our controller. False for disconnect.

As SimTP receives the event, it calls register or unregister function based on the “enter” value shared by the event.

```
@set_ev_cls(dpset.EventDP, dpset.DPSET_EV_DISPATCHER)
        def datapath_handler(self, ev):
                if ev.enter:
                       RouterController.register_router(ev.dp, …....)
                else:
                        RouterController.unregister_router(ev.dp)
```

Note: ev.dp is a datapath object which will have the “switch id”.

So now SimTP Application has the list of registered switches with there respective ids.

RouterController class is the main class which maintains the list of routers using list object called “_ROUTER_LIST”.

```
class RouterController(ControllerBase):
                _ROUTER_LIST = {}
```

Once switches got registered and negotiated for the OpenFlow Protocol version , controller will request to the switch for its features like port number and respective MAC address. As the switches shares there features in the reply message, SimTP Application stores this data into the PortData list. i.e. PortData is a list of the Port : {port_number , MAC Address}.

Port Number	MAC Address
1	12:13:14:15:16:17
2	11:22:33:44:55:66

Figure: PortData Data Structure

Router Class will have the PortData class object which means, it will be per Router.

```
            class Router(dict):
                    # STP
                    def __init__(self, dp, logger, mpls_data):
                            super(Router, self).__init__()
                            self.sw_id = dp.id
                            self.logger = logger
                            self.port_data = PortData(dp.ports)



            class PortData(dict):
                    def __init__(self, ports):
                            super(PortData, self).__init__()
                            for port in ports.values():
                                    data = Port(port.port_no, port.hw_addr)
                                    self[port.port_no] = data

            class Port(object):
                    def __init__(self, port_no, hw_addr):
                            super(Port, self).__init__()
                            self.port_no = port_no
                            self.mac = hw_addr
```

Till this stage, SimTP knows all the registered switches with there respective "switch id" and port level information per switch.

2.2.1.2. User configuration using REST API

Networks administrators can get, set and delete data for each router using the REST API that is defined in the controller. Administrators can use HTTP commands GET, POST and DELETE to do this.

We want to pass the following messages to the controller as tunnel information:

- **Network address/Mask and Output port for a given tunnel:** This information will help the controller to decide that out port.

- **Tunnel type (LER/LFR):** we need tunnel type to distinguish between Edge router and intermediate router for a flow.

To achieve this, we must add four new keywords in the REST API

a. prefix
b. tunnel_id
c. tunnel_type
d. tunnel_ouport

Ryu has a Web server function corresponding to WSGI which listens oo an "address:port" for WebSocket connections. By using this function, it is possible to create a REST API, which is useful to link with other systems or browsers.

Note: WSGI means a unified framework for connecting Web applications and Web servers in Python.

SimTP has the main RestRouterAPI Class which has variable _CONTEXT to specify Ryu's WSGI-compatible Web server class. By doing so, WSGI's Web server instance can be acquired by a key called the wsgi key.

```
            class RestRouterAPI(app_manager.RyuApp):

                  _CONTEXTS = {'dpset': dpset.DPSet,
                  'wsgi': WSGIApplication}

                  def __init__(self, *args, **kwargs):
                          super(RestRouterAPI, self).__init__(*args, **kwargs)
```

```
                         wsgi = kwargs['wsgi']
```

RestRouterAPI Class Constructor acquires the instance of WSGIApplication to register the controller class. For registration, the registory method is used. When executing the registory method, the dictionary object is passed in the key name as “RouterController” so that the “ryu-manager” main thread can access the instance of the RouterController class of SimTP Application Thread.

```
        wsgi.registory['RouterController'] = self.data
```

After the RouterController registration, RestRouterAPI class initialization maps each possible HTTP commands to a functions like get_data with GET , set_data with POST etc.

```
        mapper = wsgi.mapper
        path = '/router/{switch_id}'
        mapper.connect('router', path, controller=RouterController,
                             requirements=requirements,
                             action='get_data',
                             conditions=dict(method=['GET']))

        mapper.connect('router', path, controller=RouterController,
                             requirements=requirements,
                             action='set_data',
                             conditions=dict(method=['POST']))

        mapper.connect('router', path, controller=RouterController,
                             requirements=requirements,
                             action='delete_data',
                             conditions=dict(method=['DELETE']))
```

As we know, WSGI opens a WebSocket for the web applications to interact with the running application to set or get the data from the application.

When user sends the josn formatted file to an url using curl command like mentioned below

```
curl -d '{ "prefix":"10.0.1.0/24","tunnel_id":"20"}'  http://127.0.0.1:8080/router/0000000000000001
```

Based on the actions mentioned in the curl command, WSGI module will call the mapped functions of the RouterController Application.

For Method type “POST” , it calls set_data and based on the key value mentioned in the curl command like ”prefix”, “tunnel_id” etc , it addeds the data into the respective data structure.

```
      def set_data(self, data):
          details = None
          try:
              # Set address data
              if REST_ADDRESS in data:
                  address = data[REST_ADDRESS]
                  address_id = self._set_address_data(address)
                  details = 'Add address [address_id=%d]' % address_id
```

```
            if REST_GATEWAY in data:
               gateway = data[REST_GATEWAY]
               self.send_arp_request(address, gateway)
               details = 'Sending ARP from %s to %s', address, gateway

        elif REST_TUNNELID in data:
             tunnel_id = data[REST_TUNNELID]
            if REST_TUNNEL_TYPE in data:
                if REST_TUNNEL_OUTPORT in data:
                   tunnel_type = data[REST_TUNNEL_TYPE]
                   tunnel_outport=data[REST_TUNNEL_OUTPORT]
                   self.tunnel_data[tunnel_id] =
                            [tunnel_id, tunnel_outport, tunnel_type]

                   details = 'Add tunnel data- type: %s ' % tunnel_type

        elif REST_PREFIX in data:
             prefix = data[REST_PREFIX]
             prefix = self.prefix_data.add(prefix,
                          self.tunnel_data[tunnel_id])

               self.ofctl.send_flow_flow(cookie,priority,
                          dl_type=ether.ETH_TYPE_IP,
                          dst_ip=prefix.address,
                          dst_mask=prefix.netmask)
```

Note:
1. After storing the address_data, it will send the ARP message to the “gateway” for the host/neighbor address learning.

2. When user configures tunnel information, it sends the flow mod messages for each prefix to the switch so that whenever new IP message comes with dst_ip which matches the prefix, can send to controller. And controller will decide about the actions.

In this section, get_data and delete_data related detailed information is not mentioned because it works similarly to the set_data.

At this stage, SimTP Application would have populated below mentioned data structures

1. AddressData: List of Addresses assigned to each port of the switch.

2. PrefixData: list of prefixes present in the network.

3. Tunnel_Data: list of tunnel objects {tunnel_id, tunnel_outport, tunnel_type}

4. PortData : list of ports related information {port number, MAC Address}

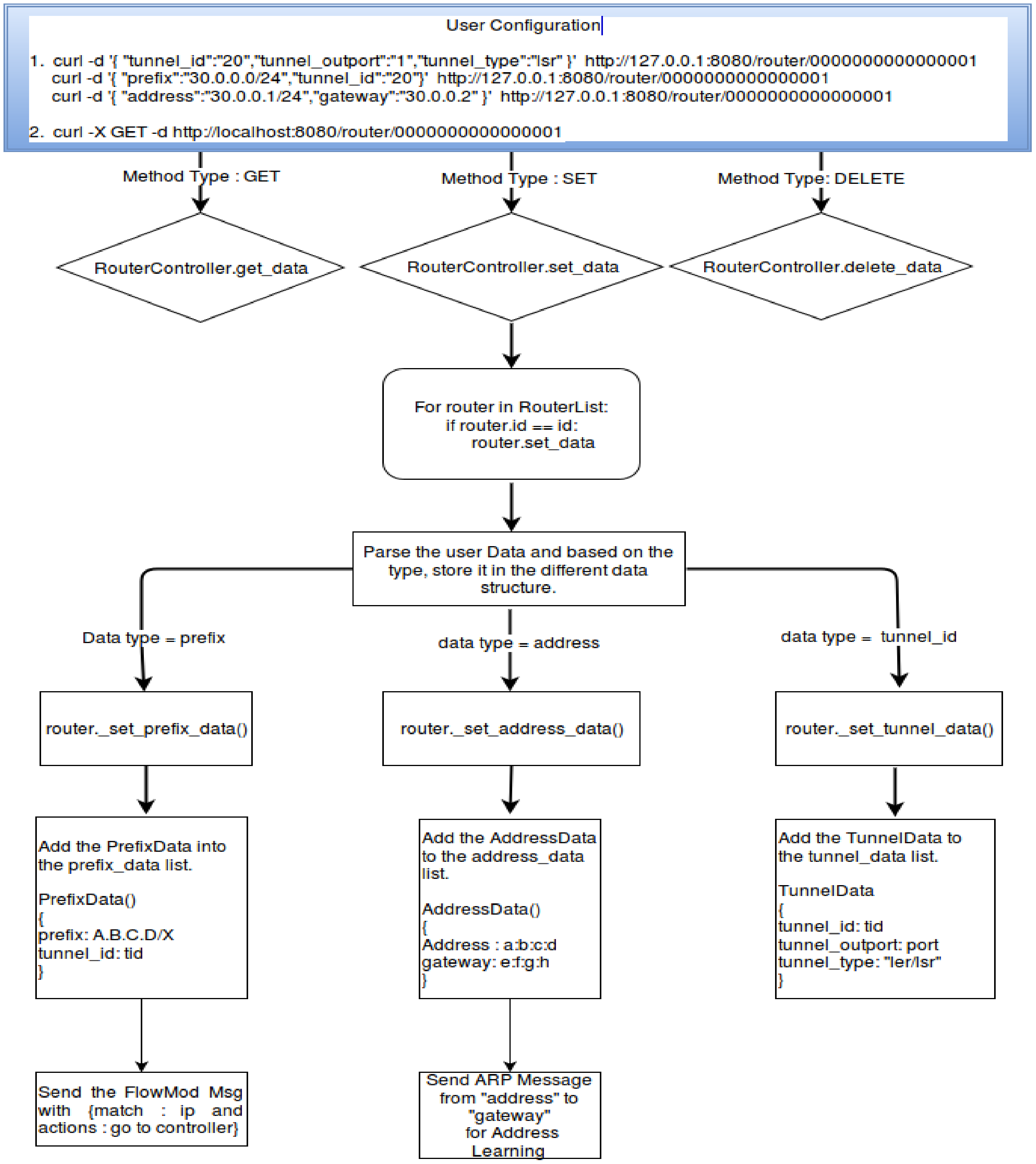
5. Hosts: It is Dictionary which keeps the information related to its neighbor {output_port, host_ip, host_mac}. It will get populated whenever OVS sends ARP/IP message to SimTP. This is how, SimTP application learns addresses of the hosts.

```
      pseudo code:
                while [ waiting for new packet-in ]
                        packet_in_handler(msg)
                        in_port = msg.in_port
                        src_ip = msg.src_ip
                        src_mac = msg.src_mac
                        hosts.add{in_port, src_ip, src_mac}
```

These are the data structure which helps in determining the actions for each packet_in message like

1. what should be the tunnel_id for new ICMP message?
2. what actions {pop/push} should it take?
3. What will be the dst_mac. src_mac of the next hop?

2.2.2.2. Call Flow Diagram for user configuration using REST API



User Configuration
1. curl -d '{ "tunnel_id":"20","tunnel_outport":"1","tunnel_type":"lsr" }' http://127.0.0.1:8080/router/0000000000000001
curl -d '{ "prefix":"30.0.0.0/24","tunnel_id":"20"}' http://127.0.0.1:8080/router/0000000000000001
curl -d '{ "address":"30.0.0.1/24","gateway":"30.0.0.2" }' http://127.0.0.1:8080/router/0000000000000001
2. curl -X GET -d http://localhost:8080/router/0000000000000001
Method Type : GET
Method Type : SET
Method Type: DELETE
RouterController.get_data
RouterController.set_data
RouterController.delete_data
For router in RouterList: if router.id == id: router.set_data
Parse the user Data and based on the type, store it in the different data structure.
Data type = prefix
data type = address
data type = tunnel_id
router._set_prefix_data()
router._set_address_data()
router._set_tunnel_data()
Add the PrefixData into the prefix_data list. PrefixData() { prefix: A.B.C.D/X tunnel_id: tid }
Add the AddressData to the address_data list. AddressData() { Address : a:b:c:d gateway: e:f:g:h }
Add the TunnelData to the tunnel_data list. TunnelData { tunnel_id: tid tunnel_outport: port tunnel_type: "ler/lsr" }
Send the FlowMod Msg with {match : ip and actions : go to controller}
Send ARP Message from "address" to "gateway" for Address Learning

2.2.1.3. SimTP Ryu Application

2.2.1.4. Call flow steps of the SimTP Application Code

1. whenever flow miss happens at OVS, it will send the packet_in message to controller.

2. Once controller main thread receives the message, it will raise the packet_in event.

3. _packet_in_handler API (RouterController class) will get called because it is registered with packet_in event.

```
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def packet_in_handler(self, ev):
                RouterController.packet_in_handler(ev.msg)
```

4. Now packetin_handler API has the msg object. So it will parse the dp_id from the message and dp_id is a unique id for each router.

5. SimTP Application will iterate through the Router_list (which is a data member of RouterController class) and will extract the Router which matches with the dp_id.

```
     @classmethod
        def packet_in_handler(cls, msg):
                dp_id = msg.datapath.id
                if dp_id in cls._ROUTER_LIST:
                        router = cls._ROUTER_LIST[dp_id]
                        router.packet_in_handler(msg)
```

6. Once SimTP Application find a router then it will call _packet_in_handler API of the Router class.

7. "_packet_in_handler(msg)" API will have the received msg object and it will extract the "message type". Based on the message type, SimTP calls different API's .

```
if message_type == ARP
      _packet_in_arp(msg)
else if message_type == MPLS
      # comment: lsr stands for intermediate router.
      if tunnel_type = "lsr"
            _packet_in_transit_tunnel(msg)
      else
            _packet_in_exit_tunnel(msg)
else if message_type == IPV4
      _packet_in_enters_tunnel(msg)
```

Pseudo code for each API:

a._packet_in_arp(msg)

```
def _packet_in_arp(msg)
     dst_ip = msg.dst_ip          // dst_ip will be the gateway IP.
     for prefix in PerfixData:
         if dest_ip == prefix:
               # comment: prefix: {address, tunnel_data}
               tunnel_data = prefix[2]

               # comment: tunnel_data:
                       {tunnel_id, tunnel_out_port, tunnle_type}
               out_port = tunnel_data[2]

               dst_mac = PortData[port].mac

            break

      send_packet_out(msg with dst_mac, actions: ARP_REPLY, )
```

Note: SimTP finds the respective prefix data entry from the PrefixData data structure for the dst_ip of the received message. Once it finds prefix data, it will get to know the tunnel_data. Based on the tunnel_data, it will do lookup for the output port. And once it gets the output port, it will find the Gateway MAC address of that out_port from PortData Data structure. At the end, it will send the packet_out with actions ARP_REPLY with dst_mac address of the gateway.

b. _packet_in_enters_tunnel(msg):

```
def _packet_in_enters_tunnel(msg):
      dst_ip = msg.dst_ip          // dst_ip will be the gateway IP.
      in_port = msg.in_port
      for prefix in PerfixData:
           if dest_ip == prefix:
            #comment: prefix: {address, tunnel_data}
            tunnel_data = prefix[2]

            #comment: tunnel_data: {tunnel_id, tunnel_outport, tunnle_type}
            out_port = tunnel_data[2]

            tunnel_id = tunnel_data[1]

            #comment: portData: {port number, MAC Address} of the switch.
            src_mac = PortData[out_port].mac

            #comment:  host:  {out_port,  host_ip,  host_mac}  Address  info  of
neighbor.
            dst_mac = hosts[out_port].mac

            break

      send_flow_mod("match" : ether_type: mpls,
                    "actions": push_mpls:tunnel_id, "set_src_mac" : src_mac,
                    "set_dst_mac": dst_mac)
```

Note: Similarly, to the packet_in_arp API, it will find the Source MAC address based on the out_port from PortData Data structure. Now it needs to find the destination MAC address, to achieve that it will do lookup in hosts data structure because hosts maintain the address information of the neighbor which is reachable through that port. At the end, send the flow_mod with match criteria as "ether_type:mpls" and actions as {"push_label", label:tunnel_id, set_src_mac, set_dst_mac, out_port}

c. _packet_in_transit_tunnel(msg):

```
def _packet_in_transit_tunnel(msg):
      #Comment: msg will have the mpls_label
      label_in = msg.mpls_label
      out_port = tunnel_data[label_in].tunnel_outport

      #comment: port: {port number, MAC Address} of the switch.
      src_mac = PortData[out_port].mac
      # comment: host: {out_port, host_ip, host_mac } Address info of neighbor.
      dst_mac = hosts[out_port].mac

      send_flow_mod("match" : ether_type: mpls, "mpls_label" : label_in
                     "actions":push_mpls:tunnel_id,"set_src_mac", "set_dst_mac")
```

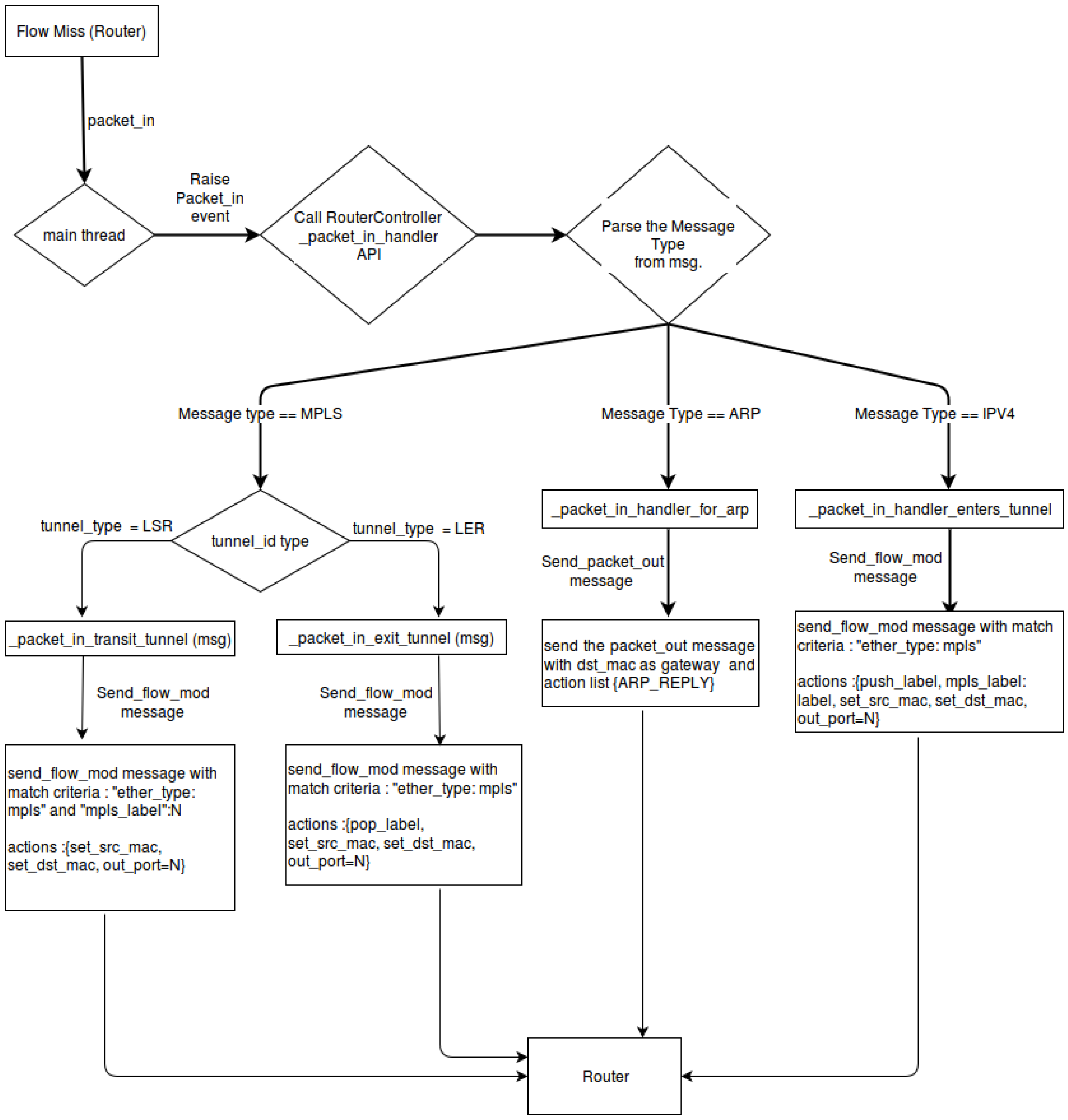
Note: packet-in message will have the mpls_label. Based on this mpls_label, SimTP will do lookup in the tunnel_data data structure and find the corresponding out_port. Once it gets out_port then it can easily get the source and destination mac address from the "PortData" and "Hosts" data structure. At the end, it will send the flow mod message with match criteria as {ether_type: mpls , mpls_label: label_in } and actions as {set_src_mac, set_dst_mac, out_port:n }

d._packet_in_exit_tunnel(msg)

```
def _packet_in_exit_tunnel(msg)
      # Comment: msg will have the mpls_label
      label_in = msg.mpls_label
      out_port = tunnel_data[label_in].tunnel_outport

      #comment: port: {port number, MAC Address} of the switch.
      src_mac = PortData[out_port].mac

      #comment: host: {out_port, host_ip, host_mac } Address info of neighbor.
      dst_mac = hosts[out_port].mac

      send_flow_mod("match" : ether_type: mpls, "mpls_label" : label_in
                    "actions": pop_mpls, set_src_mac, set_dst_mac)
```

Note: similarly, like above API, it will find the src and dst_mac. But in this case, the actions will be pop because message is exiting the tunnel.

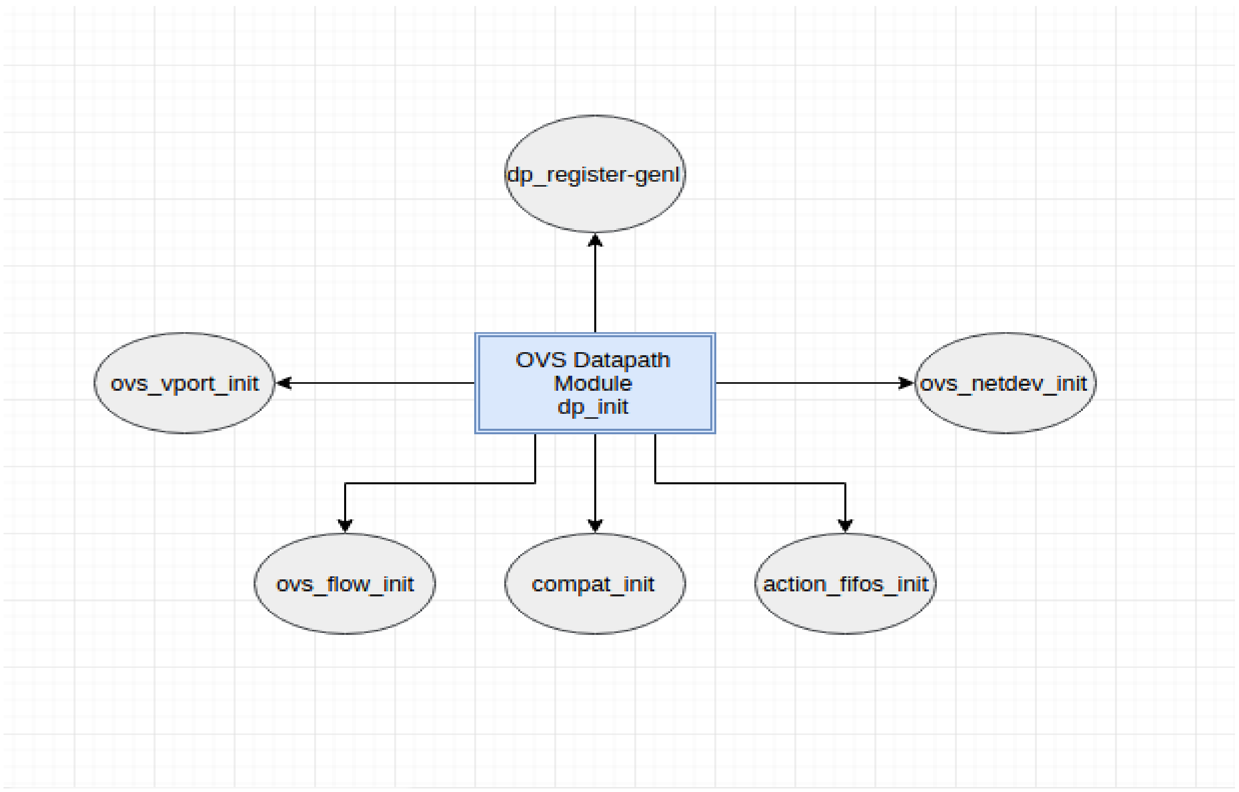
2.2.2.5. *Call Flow Diagram* of the SimTP Application Code



Flow Miss (Router)
packet_in
main thread
Raise Packet_in event
Call RouterController _packet_in_handler API
Parse the Message Type from msg.
Message type == MPLS
Message Type == ARP
Message Type == IPV4
tunnel_type = LSR
tunnel_id type
tunnel_type = LER
_packet_in_handler_for_arp
_packet_in_handler_enters_tunnel
Send_packet_out message
Send_flow_mod message
_packet_in_transit_tunnel (msg)
_packet_in_exit_tunnel (msg)
send the packet_out message with dst_mac as gateway and action list {ARP_REPLY}
send_flow_mod message with match criteria : "ether_type: mpls"
actions :{push_label, mpls_label: label, set_src_mac, set_dst_mac, out_port=N}
Send_flow_mod message
Send_flow_mod message
send_flow_mod message with match criteria : "ether_type: mpls" and "mpls_label":N
actions :{set_src_mac, set_dst_mac, out_port=N}
send_flow_mod message with match criteria : "ether_type: mpls"
actions :{pop_label, set_src_mac, set_dst_mac, out_port=N}
Router

2.2.2 OVS Kernel Module

The heart of the OVS kernel is the datapath module. This is the main module which registers all the handlers to different events and initializes other sub-modules.



dp_init() is the init function of the datapath module. The init function further calls the init functions of the supporting modules.

- compat_init() - This is to initialize modules required for backward compatibility to older versions of the linux kernel.
- action_fifos_init() - This allocates the action queues per cpu.
- ovs_flow_init() - This initializes the flow module and creates flow cache to store flows.
- ovs_vport_init() - This initializes the vport sub-system which includes sub-modules like lisp, vxlan, gre modules etc.
- ovs_netdev_init() - This initializes the ovs netdev module and registers handlers to create, and destroy ovs netdev, and also send packets on the netdev.

It also registers handlers to different netlink messages using dp_register_nl() function. There are 4 families of netlink messages to which handlers are initialized -

1. OVS_DATAPATH_FAMILY – Handles addition, deletion and modification of datapath in the kernel.

2. OVS_VPORT_FAMILY – Handles operations related to vport.

3. OVS_FLOW_FAMILY – Handles flow operations like add, del, set etc.

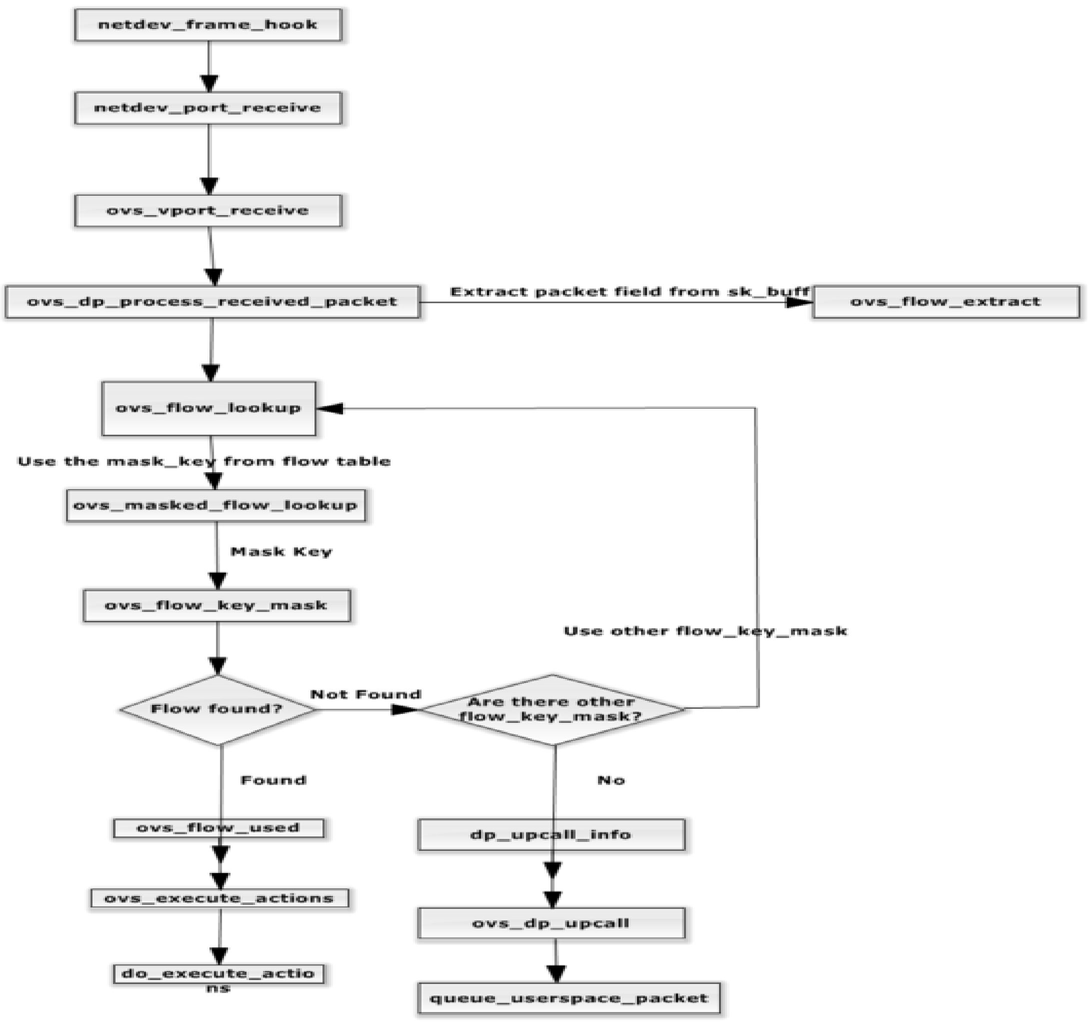
4. OVS_PACKET_FAMILY – Handles packet operations (usually packets received from userspace).

In addition to this, dp_init() also registers the netdevice notifier (register_netdevice_notifier), registers devices corresponding to the network namespace to a list for sending notifications (register_pernet_device), and registers handlers for various operations on internal dev links (ovs_internal_dev_rtnl_link_register).

2.2.2.1 Dataflow:

Initially, when user executes command to create a new datapath (new bridge) using vsctl commands, and adds a port (vport) to the datapath, the vswitchd parses the command, frames netlink messages to create the same in kernel space and signals the kernel by sending the netlink message. Netlink message type OVS_DATAPATH_FAMILY is used to handle datapath related changes like addition, deletion and modification of the datapaths. Netlink message type OVS_VPORT_FAMILY is used to handle vport related changes like addition, deletion and modification of vports to the respective datapaths.

Once a datapath is created and ports are added to it, any packet received on that port is handled by the particular datapath. Here is the complete flow of the packet in the datapath:



When a vport is created and added to the datapath, the netdev_link() command registers the corresponding physical port's receive handler to netdev_frame_hook(). So, when a packet is received on the port, netdev_frame_hook is called which further calls netdev_vport_receive(). This function further calls the ovs_vport_receive() which does the datapath packet processing. In older versions of the ovs, ovs_dp_process_received_packet() function would do the actual processing of the received packet. In the latest versions, this function has been replaced by ovs_dp_process_packet() function which does the same processing as such.

When processing the packet, the function first extracts the flow from the packet. A flow can be based upon various fields in the packet including but not limited to Ethernet header fields, IP header fields, TCP/UDP header fields etc.

Once the flow is extracted, a flow lookup is performed based on the flow key mask present in the table. If the packet matches the flow entry, ovs_execute_action() with the corresponding action is called which performs the action by calling do_execute_actions(). Here, the corresponding action is taken for the packet and the packet flow ends. However, if the flow is not matched with a particular flow entry, it is matched with other possible matching flow_key_masks. If a packet does not match any flow, then an exception is raised and the packet is sent to the user level for processing. The dp_upcall_info structure is populated and an ovs_dp_upcall() function is performed for the same. All packets to user level are usually queued using queue_userspace_packet().

The user level further processes this packet (normally frames an openflow request packet to the controller asking for information to handle this packet). When the user level has processed the packet and is ready to send the packet, it submits the packet to the kernel by sending a netlink message of type OVS_PACKET_FAMILY and passing the packet to the kernel. The datapath in the kernel runs ovs_packet_command_execute() function to process this packet. This function call is similar to ovs_dp_process_packet(). The function again looks up for a flow entry in the table to take the corresponding action except that a flow would be present in this case because the packet would be forwarded to the controller.

Note: Initially if the flow to the controller is missing, the packet is again fed back to user level. The user level would then add a flow to the controller (based on its configuration) in the datapath kernel and then the packet would be processed. This is however only for the first ever packet received by the datapath.

When the controller responds with a flow for the packet, the flow is added by the user space daemon using the netlink message of type OVS_FLOW_FAMILY. So, next time a packet of that flow is received, it is matched with the flow and corresponding action is taken. Control is not given to user level.

This dataflow is same irrespective of any packet inbound. The only change is extracting the various fields in the packet to match the corresponding entry in the flow table. For the implementation of SimTP protocol as such, the existing code of MPLS can be reused to achieve the final desired outcome. So, there is no need to change the code in the datapath.

3. Self-Study Plan

3.1. Description of Base Case:

The base case scenario for our system is to study IP based forwarding without implementation of the proposed SimTP protocol. The study of delay with respect to number of traffic flows, amount of traffic and the kind of traffic needs to be undertaken. For this, the complete packet flow in the ovs switch module needs to be understood. In the ovs module, regular destination based IP forwarding is done using the flow entries based on the destination address field in the IP header. The datapath module consists of flow tables, each consisting of multiple flow entries. Hence, the performance of the datapath forwarding is inversely proportional to the number of flow entries in the table. An incoming packet is hash mapped to an existing flow entry based on the headers it contains. Wildcard matching is also supported but the exact/longest matching flow is given higher priority. Understanding these factors and their impacts on our system is paramount.

3.2. Characteristics to Observe:

The main characteristics to be observed in the project environment in comparison to the base case environment are -

1. Delay improvement, if any, in the traffic.

2. Traffic drop, if any.

3. Jitter impact, if any.

4. Ease of configuration at the controller (though this can be a subjective parameter).

3.3. Range of Scenarios to Investigate:

1. Performance impact i.e. the traffic delay or traffic drop when forwarding high traffic of only one flow while having hundreds of entries in the flow table.

2. Traffic delay/drop when forwarding high traffic evenly/randomly distributed across hundreds of flows present in the flow table.

3. Traffic delay/drop when forwarding small amount of traffic equally distributed across hundreds of flows.

4. Measuring performance as a factor of increasing number of flows.

5. Measuring performance with different types of traffic – TCP, UDP, IPv4, IPv6.

3.4. Self-Study Results:

Multiple observations were done during the self study. The result and analysis of the same are as explained below:

The delay seen for packets on single flow/multiple flows with high traffic was more in SimTP based forwarding than regular IP lookup based forwarding. This is because latency is introduced while adding new tunnel(MPLS) header when a packet comes into the network, and also while removing the header when packet goes out of the network.

For TCP, there is poor performance w.r.t rate of transmission. TCP adjusts its transmission rate and sends lesser traffic because of the delay. Performance of UDP w.r.t transmission rate remains same though.

In case of ICMP, the average round trip time was higher for SimTP application, however we have seen tests where it results in lower value also as compared to the base test case of running rest_router.py. This is shown in the test cases in next section.

There was less packet loss(in SimTP Application) seen when pumping high traffic of ICMP with large packet frames on single/multiple flows on a regular router. This is perhaps because of large number of flows required on the regular router application and more intervention needed by the controller to provide information on every openvswitch node as compared to SimTP application.

When it came to number of flows, SimTP clearly had much lesser flows on the entire system. On intermediate nodes, there were flows only for the tunnels as against for every packet destination type in the case of regular router. Thus, the system memory consumed for SimTP protocol is better than regular router application

The ease of configuration was better in case of SimTP as there were fewer commands to be executed for the same topology. Also, once a config is done for a particular tunnel, adding new configurations for packets destined to the same edge port is very simple. In case of regular router, configuration is tedious as the number of flows increase. Extensibility of configuration is higher in SimTP.

Note- The results we mentioned above is obtained through the various test-cases we performed and this is stated in the Test Section.

4. Reflection

4.1 Per-member learning experiences

Suhas-

Learnt the packet flow and working of the openvswitch datapath - Packet handler functions, flow table handling, socket communication with vswitchd daemon.

Learnt about the RYU software architecture hierarchy. Learnt the packet handling in Ryu router application while writing the code for SimTP application.

Learnt about the practical implementation of networking concepts like Routing (static route) & Forwarding, MPLS and ARP.

Davis-

Understood the communication between the OVS-Vswitchd and the controller through the socket and the various kind of control data that is transmitted in between, so as to provide the end result of forwarding the packet such as flowmod(containing match criteria and actions) and handling of various ether-types through packet-In and Out.
Read the vswitchd block(the daemon ovs-vswitchd.c) of the OVS code to understand the communication between OVS and controller as we initially attempted to create our own label(ethertype) rather using the MPLS label, and to do so I had observed the functions defined within the code -bridge.c that calls functions defined in ofproto.c(it is this that handles the match and action part as it has functions that call the respective codes) that calls functions defined in Connmgr.c and it is this code that handles the socket communication between the RYU controller and OVS.

Had a beginner level of understanding of python and dwelling into the RYU controller code was a wonderful learning experience for me as I observed the powerful parsing tools with the use of dictionaries and its inherent structure for data serialization over the socket.

Jalandhar Singh-

When I started the project, it didn't know much about OpenFlow protocol. So, journey started from understanding the OpenFlow protocol by reading Specification and parallelly I started playing with ovs-controller setup. After getting some basic understanding, I started learning, How router application works on RYU controller by looking into ryu code and taking wireshark output at all the interfaces?. During this stage, I understood RYU controller based infrastructure w.r.t code. Then started with next task of REST API module because user interface plays very important role in the Application on the controller. After understanding REST API, I started writing the SimTP application code for normal tunnel based routing. Once I got success in the ping, I started integrating the concepts like "Auto learning of the Addresses of Neighbors". At the end of the project, I realized that how a controller plane application can makes dump OVS working as it want ?

4.2 What went as expected

1. User input through REST API module.

2. Tunnel ID (MPLS Label) push and/or pop flows on the LER node as per the configuration.
3. No label switching at LSR node.
4. With the SimTP application, on the LSR(Intermediate node) I expected a decrease in the number of flows introduced.
5. Auto Learning of the Neighbors Addresses.

4.3 What went different from as Expected

Initially, openvswitch didn't behave as expected though the openflow version (1.3) which supported MPLS and hence our application was integrated. Later found that the datapath supports MPLS only on kernel versions 3.19 or later. Hence, had to upgrade the ExoGeni VMs with Fedora images (only available OS in ExoGeni which supports higher versions of the kernel).

The base code of RYU router application did not support address addition per port of the switch. The router used to rely on the network it is connected to, to figure out the port for the particular address. Because of this our initial design which assumed the address to interface mapping to be present had to be modified.

Sending the SimTP(MPLS) tagged frame and observing IPerf initially gave no results/output. We realized that packet now is a mpls tagged and hence would be more than 1500Byte and hence before transmission from the end-host we had to specify the mtu size so as to ensure that as it gets tagged and sent out through OVS, it should be lesser than 1500B, else packet would be dropped.

5. Task Decomposition

5.1. High Level Tasks

1. Application Development on top of RYU for SimTP protocol.
2. Integrating control communication (actions) for SimTP using OpenFlow.
3. Integrating simTP with Kernel Module to process packets in the Data Plane.

5.2 Per Member Responsibility

Task	Name
Base class level coding	Jalandhar Singh
ARP message handling in the simTP Application	Suhaskrishna Gopalkrishna
ICMP message handling in the simTP Application	Suhaskrishna Gopalkrishna
simTP message handling in the simTP Application	Jalandhar Singh
Addition of new elements in the simTP Application's REST API	Jalandhar Singh
Integrating simTP with Kernel Module to process packets in the Data plane	Suhaskrishna Gopalkrishna
Integrating Control Communication(OpenFlow) for SimTP working	Davis Polly Pynadath
End to End testing	Davis Polly Pynadath
Stress testing	Davis Polly Pynadath
SimTP Application Design Document	Jalandhar Singh
Kernel Module Design Document	Suhaskrishna Gopalkrishna

Test cases and Demo plan	Davis Polly Pynadath

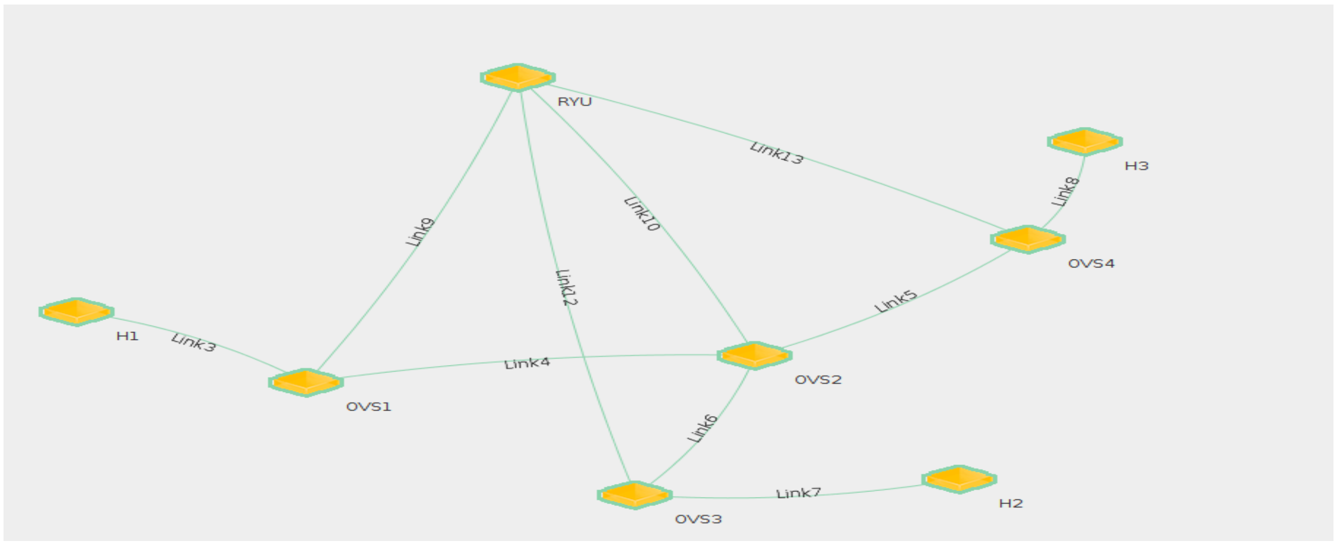
6. Timeline

Objective	**Start Data**	**End Date**	**Status**
Project Proposal	26th Sept	6th Oct	Completed
Getting the SDN (RYU and OVS) network UP	7th Oct	14th Oct	Completed
ARP message handling in the simTP Application	28th Oct	7th Nov	Completed
ICMP message handling in the simTP Application	29th Oct	9th Nov	Completed
simTP message handling in the simTP Application	2nd Nov	18th Nov	Completed
Base class level coding	26th Oct	29th Oct	Completed
Addition of new element in the simTP Application’s REST API	9th Nov	15th Nov	Completed
Integrating simTP with Kernel Module to process packets in the Data plane	7th Nov	18th Nov	Completed
Integrating Control Communication(OpenFlow) for SimTP working	7th Nov	18th Nov	Completed
End to End testing with development	4th Nov	18th Nov	Completed
Stress testing with development	8th Nov	18th Nov	Completed
Demo	28th Nov	28th Nov	

7. Verification and Validation Plan

7.1 Demo Topology

The SimTP application will be run on the demo setup shown below.

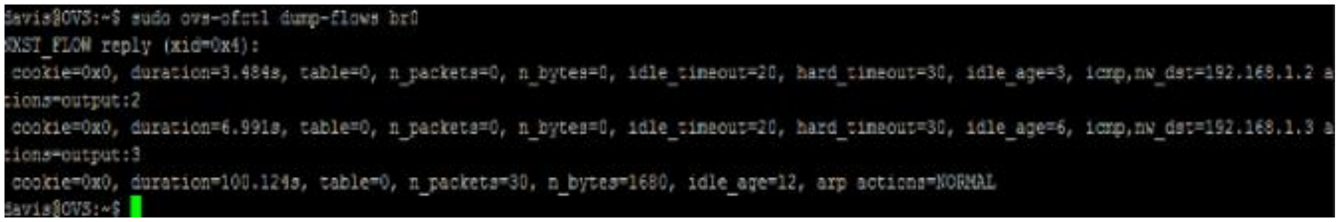


Tunnels will be configured between the end hosts- H1, H2 and H3 by using the SimTP application which is running on the RYU controller. Also, we would be using Wireshark/tcpdump/Iperf for analyzing the various output such as bandwidth consumption, packet-loss, delay and jitter.

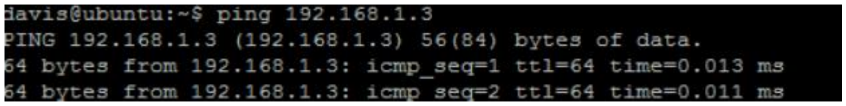
7.2 Test Plan

Topology Creation and Normal Function Verification (32 bit IPv4 destination lookup)			
Test ID	**Action**	**Expected Observation**	**Conclusion**
1.1	The topology shown in figure is created in Exogeni and additional user created to enable remote login of project-team	All users able to login to all the nodes	The topology is online
1.2	Configured ARP Flows on all switches and ping executed for H1, H2 and H3	H1, H2 and H3 MAC learnt on all OVS's and end hosts and ping unsuccessful	ARP learning established
1.3	Configured ICMP flows to ensure icmp packets reaches destination (h1, h2 and h3)	Ping Successful	The actual physical ports of the inter-switch OVS links are identified properly and normal forwarding function based on IP destination verified.

Result- Ping was successful and the arp and icmp flows are present in flow-tabel0



```
davis@OVS:~$ sudo ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=3.484s, table=0, n_packets=0, n_bytes=0, idle_timeout=20, hard_timeout=30, idle_age=3, icmp,nw_dst=192.168.1.2 a
tions=output:2
 cookie=0x0, duration=6.991s, table=0, n_packets=0, n_bytes=0, idle_timeout=20, hard_timeout=30, idle_age=6, icmp,nw_dst=192.168.1.3 a
tions=output:3
 cookie=0x0, duration=100.124s, table=0, n_packets=30, n_bytes=1680, idle_age=12, arp actions=NORMAL
davis@OVS:~$
```



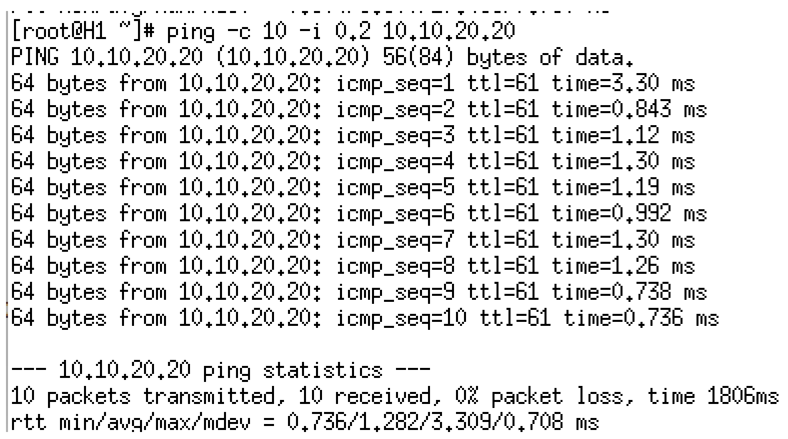
```
davis@ubuntu:~$ ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.013 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=0.011 ms
```

Performance Comparison with rest_router.py- Normal Routing Application			
Test ID	**Action**	**Expected Observation**	**Conclusion**
2.1	Run the rest_router.py code on the controller and add the configuration for all flows	All switches register their dpid with the controller	Rest_router.py successfully running on controller
2.2	Performance Test	The expected results are below	This is the base performance test case and is the benchmark for our comparison with respect to SimTP Application.

Result:-

Performance measurement with one flow

- **Ping Test-** we use the Ping utility to determine the **average RTT**-

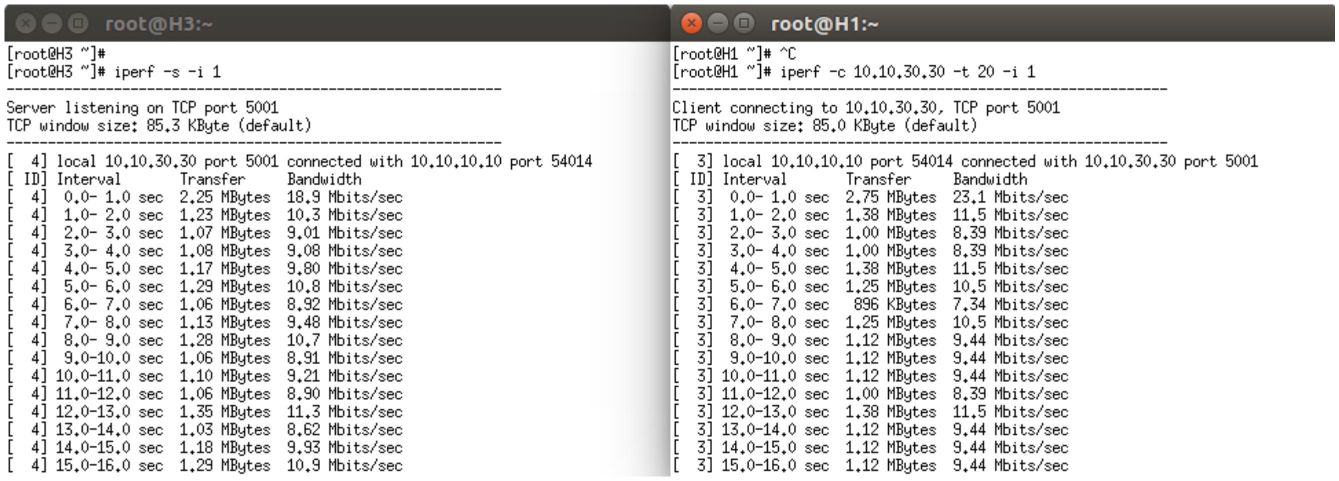


```
[root@H1 ~]# ping -c 10 -i 0.2 10.10.20.20
PING 10.10.20.20 (10.10.20.20) 56(84) bytes of data.
64 bytes from 10.10.20.20: icmp_seq=1 ttl=61 time=3.30 ms
64 bytes from 10.10.20.20: icmp_seq=2 ttl=61 time=0.843 ms
64 bytes from 10.10.20.20: icmp_seq=3 ttl=61 time=1.12 ms
64 bytes from 10.10.20.20: icmp_seq=4 ttl=61 time=1.30 ms
64 bytes from 10.10.20.20: icmp_seq=5 ttl=61 time=1.19 ms
64 bytes from 10.10.20.20: icmp_seq=6 ttl=61 time=0.992 ms
64 bytes from 10.10.20.20: icmp_seq=7 ttl=61 time=1.30 ms
64 bytes from 10.10.20.20: icmp_seq=8 ttl=61 time=1.26 ms
64 bytes from 10.10.20.20: icmp_seq=9 ttl=61 time=0.738 ms
64 bytes from 10.10.20.20: icmp_seq=10 ttl=61 time=0.736 ms

--- 10.10.20.20 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 1806ms
rtt min/avg/max/mdev = 0.736/1.282/3.309/0.708 ms
```

From above, we obtain an **average RTT-1.282ms**

Configuring H3 -10.10.30.30 as the TCP server and H1- 10.10.10.10 as the TCP client

- **TCP-**



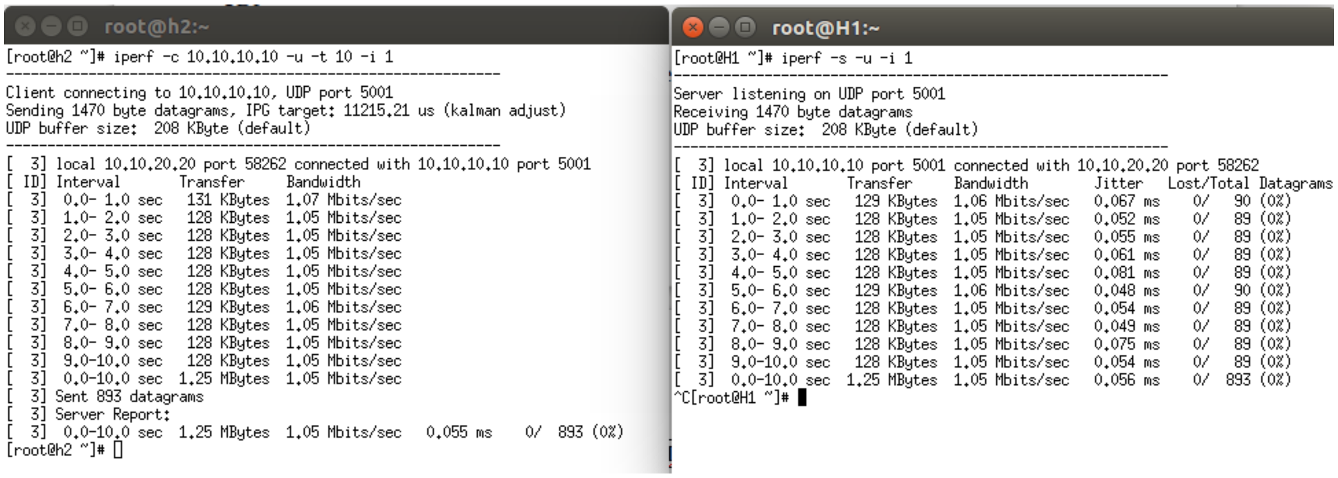
```
root@H3:~

[root@H3 ~]#
[root@H3 ~]# iperf -s -i 1
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.10.30.30 port 5001 connected with 10.10.10.10 port 54014
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 1.0 sec  2.25 MBytes  18.9 Mbits/sec
[  4]  1.0- 2.0 sec  1.23 MBytes  10.3 Mbits/sec
[  4]  2.0- 3.0 sec  1.07 MBytes  9.01 Mbits/sec
[  4]  3.0- 4.0 sec  1.08 MBytes  9.08 Mbits/sec
[  4]  4.0- 5.0 sec  1.17 MBytes  9.80 Mbits/sec
[  4]  5.0- 6.0 sec  1.29 MBytes  10.8 Mbits/sec
[  4]  6.0- 7.0 sec  1.06 MBytes  8.92 Mbits/sec
[  4]  7.0- 8.0 sec  1.13 MBytes  9.48 Mbits/sec
[  4]  8.0- 9.0 sec  1.28 MBytes  10.7 Mbits/sec
[  4]  9.0-10.0 sec  1.06 MBytes  8.91 Mbits/sec
[  4] 10.0-11.0 sec  1.10 MBytes  9.21 Mbits/sec
[  4] 11.0-12.0 sec  1.06 MBytes  8.90 Mbits/sec
[  4] 12.0-13.0 sec  1.35 MBytes  11.3 Mbits/sec
[  4] 13.0-14.0 sec  1.03 MBytes  8.62 Mbits/sec
[  4] 14.0-15.0 sec  1.18 MBytes  9.93 Mbits/sec
[  4] 15.0-16.0 sec  1.29 MBytes  10.9 Mbits/sec
```

```
root@H1:~

[root@H1 ~]# ^C
[root@H1 ~]# iperf -c 10.10.30.30 -t 20 -i 1
------------------------------------------------------------
Client connecting to 10.10.30.30, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.10.10 port 54014 connected with 10.10.30.30 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  2.75 MBytes  23.1 Mbits/sec
[  3]  1.0- 2.0 sec  1.38 MBytes  11.5 Mbits/sec
[  3]  2.0- 3.0 sec  1.00 MBytes  8.39 Mbits/sec
[  3]  3.0- 4.0 sec  1.00 MBytes  8.39 Mbits/sec
[  3]  4.0- 5.0 sec  1.38 MBytes  11.5 Mbits/sec
[  3]  5.0- 6.0 sec  1.25 MBytes  10.5 Mbits/sec
[  3]  6.0- 7.0 sec   896 KBytes  7.34 Mbits/sec
[  3]  7.0- 8.0 sec  1.25 MBytes  10.5 Mbits/sec
[  3]  8.0- 9.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3]  9.0-10.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3] 10.0-11.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3] 11.0-12.0 sec  1.00 MBytes  8.39 Mbits/sec
[  3] 12.0-13.0 sec  1.38 MBytes  11.5 Mbits/sec
[  3] 13.0-14.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3] 14.0-15.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3] 15.0-16.0 sec  1.12 MBytes  9.44 Mbits/sec
```

We observe the **average bandwidth** from client to server to be 11Mbps (we ignore the output at interval- 0.0-1 sec to determine average as the client transmits at high rate, but then both client and server decrease their respective rate of transmission, as they realize the network does not support the bandwidth due to congestion and drops)

- **UDP-**

Configuring H2 -10.10.20.20 as the UDP server and H1- 10.10.10.10 as the UDP client-



```
root@h2:~
[root@h2 ~]# iperf -c 10.10.10.10 -u -t 10 -i 1
------------------------------------------------------------
Client connecting to 10.10.10.10, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.10.20.20 port 58262 connected with 10.10.10.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   131 KBytes  1.07 Mbits/sec
[  3]  1.0- 2.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  2.0- 3.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  3.0- 4.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  4.0- 5.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  5.0- 6.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  6.0- 7.0 sec   129 KBytes  1.06 Mbits/sec
[  3]  7.0- 8.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  8.0- 9.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  9.0-10.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.055 ms    0/  893 (0%)
[root@h2 ~]# []
```

```
root@H1:~
[root@H1 ~]# iperf -s -u -i 1
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.10.10.10 port 5001 connected with 10.10.20.20 port 58262
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0- 1.0 sec   129 KBytes  1.06 Mbits/sec   0.067 ms    0/   90 (0%)
[  3]  1.0- 2.0 sec   128 KBytes  1.05 Mbits/sec   0.052 ms    0/   89 (0%)
[  3]  2.0- 3.0 sec   128 KBytes  1.05 Mbits/sec   0.055 ms    0/   89 (0%)
[  3]  3.0- 4.0 sec   128 KBytes  1.05 Mbits/sec   0.061 ms    0/   89 (0%)
[  3]  4.0- 5.0 sec   128 KBytes  1.05 Mbits/sec   0.081 ms    0/   89 (0%)
[  3]  5.0- 6.0 sec   129 KBytes  1.06 Mbits/sec   0.048 ms    0/   90 (0%)
[  3]  6.0- 7.0 sec   128 KBytes  1.05 Mbits/sec   0.054 ms    0/   89 (0%)
[  3]  7.0- 8.0 sec   128 KBytes  1.05 Mbits/sec   0.049 ms    0/   89 (0%)
[  3]  8.0- 9.0 sec   128 KBytes  1.05 Mbits/sec   0.075 ms    0/   89 (0%)
[  3]  9.0-10.0 sec   128 KBytes  1.05 Mbits/sec   0.054 ms    0/   89 (0%)
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.056 ms    0/  893 (0%)
^C[root@H1 ~]#
```

Here with respect to UDP traffic (Video, Voip), we also evaluate the jitter and the maximum UDP sending rate that results in almost no packet loss on the end-to-end path. We observe that the average transfer rate -1.05Mbytes with no packet loss and the average jitter is 55 microseconds (which is well under the jitter threshold for VoIP traffic)

Performance Measurement when large number of flows-

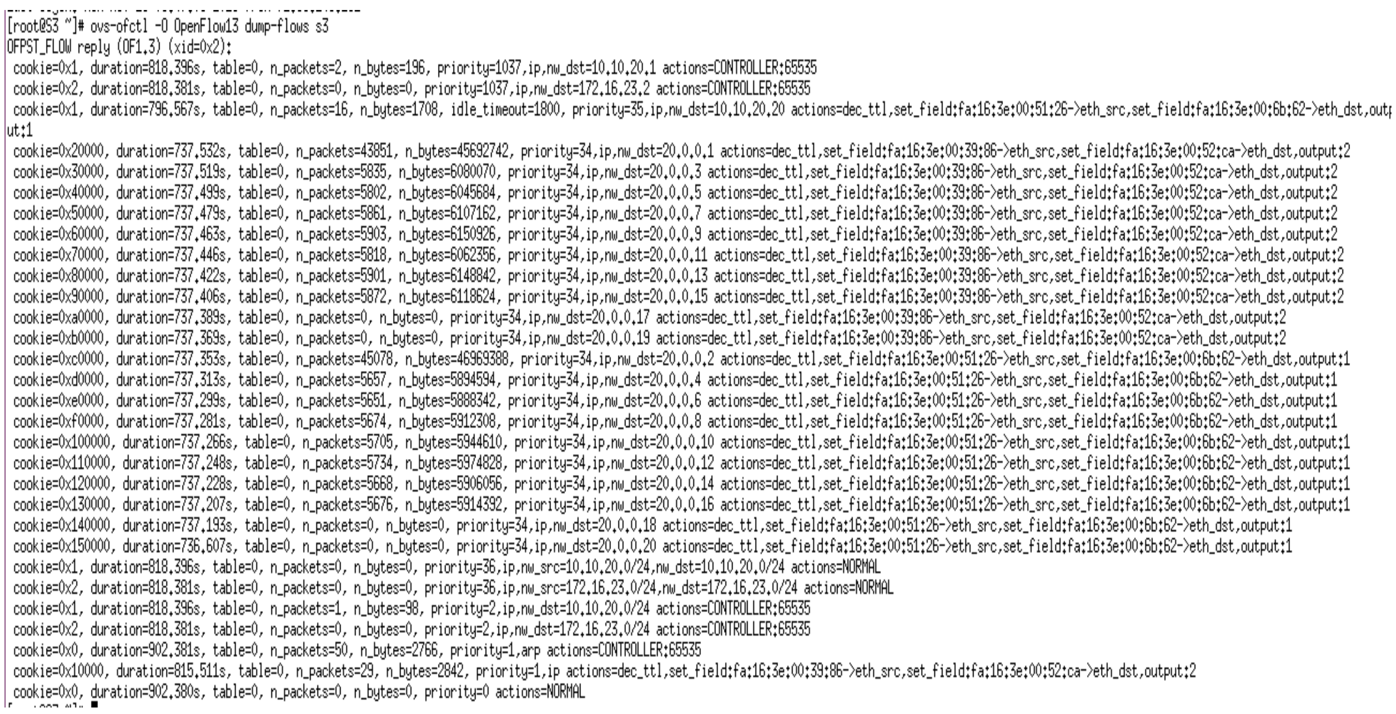
Now we introduce many flows on the OVS's by introducing 10 loopbacks on H1(20.0.0.1,20.0.03…) with odd IP addresses and 10 loopbacks on H2 with even IP addresses (20.0.0.2, 20.0.0.4….).

We wrote a bash script -pingtest.sh that executes ping on all source odd IP(H1) to all destination even IP(H2).

The ping command executed by the script is-

Ping -c 10000 -s 1000 -f -I <source loopback> <destination loopback>
i.e executing a flood ping with each echo_request to have a size of 1KB and repeat it 10000 times.

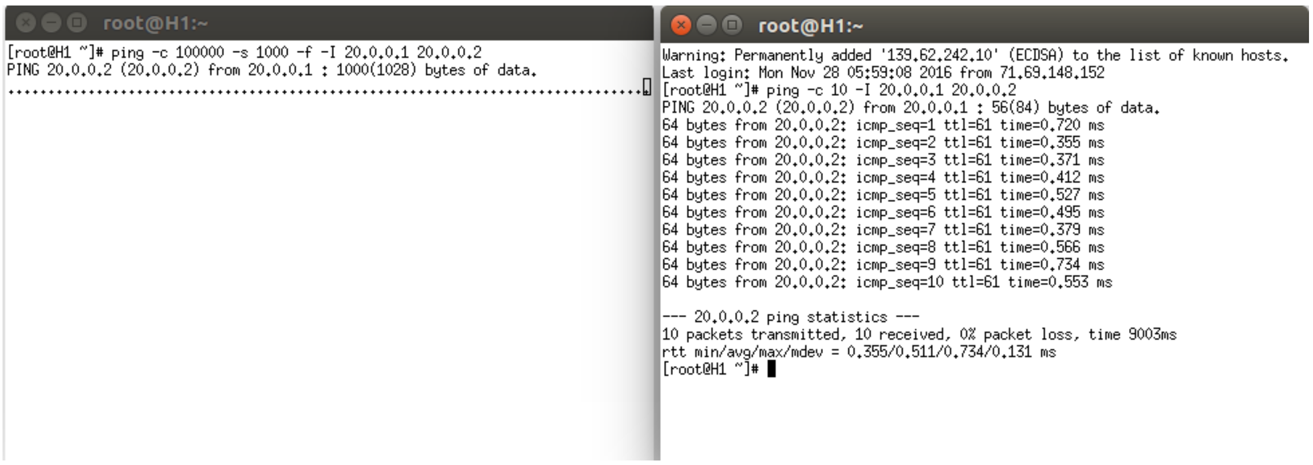
- **Number of flows:** We notice now the number of flows on the all the routers (LER/LSR) in case of normal router application.



```
[root@S3 ~]# ovs-ofctl -O OpenFlow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=818.396s, table=0, n_packets=2, n_bytes=196, priority=1037,ip,nw_dst=10.10.20.1 actions=CONTROLLER:65535
 cookie=0x2, duration=818.381s, table=0, n_packets=0, n_bytes=0, priority=1037,ip,nw_dst=172.16.23.2 actions=CONTROLLER:65535
 cookie=0x1, duration=796.567s, table=0, n_packets=16, n_bytes=1708, idle_timeout=1800, priority=35,ip,nw_dst=10.10.20.20 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,outp
ut:1
 cookie=0x20000, duration=737.532s, table=0, n_packets=43851, n_bytes=45692742, priority=34,ip,nw_dst=20.0.0.1 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x30000, duration=737.519s, table=0, n_packets=5835, n_bytes=6080070, priority=34,ip,nw_dst=20.0.0.3 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x40000, duration=737.499s, table=0, n_packets=5802, n_bytes=6045684, priority=34,ip,nw_dst=20.0.0.5 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x50000, duration=737.479s, table=0, n_packets=5861, n_bytes=6107162, priority=34,ip,nw_dst=20.0.0.7 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x60000, duration=737.463s, table=0, n_packets=5903, n_bytes=6150926, priority=34,ip,nw_dst=20.0.0.9 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x70000, duration=737.446s, table=0, n_packets=5818, n_bytes=6062356, priority=34,ip,nw_dst=20.0.0.11 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x80000, duration=737.422s, table=0, n_packets=5901, n_bytes=6148842, priority=34,ip,nw_dst=20.0.0.13 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x90000, duration=737.406s, table=0, n_packets=5872, n_bytes=6118624, priority=34,ip,nw_dst=20.0.0.15 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0xa0000, duration=737.389s, table=0, n_packets=0, n_bytes=0, priority=34,ip,nw_dst=20.0.0.17 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0xb0000, duration=737.369s, table=0, n_packets=0, n_bytes=0, priority=34,ip,nw_dst=20.0.0.19 actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0xc0000, duration=737.353s, table=0, n_packets=45078, n_bytes=46969388, priority=34,ip,nw_dst=20.0.0.2 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0xd0000, duration=737.313s, table=0, n_packets=5657, n_bytes=5894594, priority=34,ip,nw_dst=20.0.0.4 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0xe0000, duration=737.299s, table=0, n_packets=5651, n_bytes=5888342, priority=34,ip,nw_dst=20.0.0.6 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0xf0000, duration=737.281s, table=0, n_packets=5674, n_bytes=5912308, priority=34,ip,nw_dst=20.0.0.8 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x100000, duration=737.266s, table=0, n_packets=5705, n_bytes=5944610, priority=34,ip,nw_dst=20.0.0.10 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x110000, duration=737.248s, table=0, n_packets=5734, n_bytes=5974828, priority=34,ip,nw_dst=20.0.0.12 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x120000, duration=737.228s, table=0, n_packets=5668, n_bytes=5906056, priority=34,ip,nw_dst=20.0.0.14 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x130000, duration=737.207s, table=0, n_packets=5676, n_bytes=5914392, priority=34,ip,nw_dst=20.0.0.16 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x140000, duration=737.193s, table=0, n_packets=0, n_bytes=0, priority=34,ip,nw_dst=20.0.0.18 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x150000, duration=736.607s, table=0, n_packets=0, n_bytes=0, priority=34,ip,nw_dst=20.0.0.20 actions=dec_ttl,set_field:fa:16:3e:00:51:26->eth_src,set_field:fa:16:3e:00:6b:62->eth_dst,output:1
 cookie=0x1, duration=818.396s, table=0, n_packets=0, n_bytes=0, priority=36,ip,nw_src=10.10.20.0/24,nw_dst=10.10.20.0/24 actions=NORMAL
 cookie=0x2, duration=818.381s, table=0, n_packets=0, n_bytes=0, priority=36,ip,nw_src=172.16.23.0/24,nw_dst=172.16.23.0/24 actions=NORMAL
 cookie=0x1, duration=818.396s, table=0, n_packets=1, n_bytes=98, priority=2,ip,nw_dst=10.10.20.0/24 actions=CONTROLLER:65535
 cookie=0x2, duration=818.381s, table=0, n_packets=0, n_bytes=0, priority=2,ip,nw_dst=172.16.23.0/24 actions=CONTROLLER:65535
 cookie=0x0, duration=902.381s, table=0, n_packets=50, n_bytes=2766, priority=1,arp actions=CONTROLLER:65535
 cookie=0x10000, duration=815.511s, table=0, n_packets=29, n_bytes=2842, priority=1,ip actions=dec_ttl,set_field:fa:16:3e:00:39:86->eth_src,set_field:fa:16:3e:00:52:ca->eth_dst,output:2
 cookie=0x0, duration=902.380s, table=0, n_packets=0, n_bytes=0, priority=0 actions=NORMAL
```

We observe that there is a large number of flows, even though all packets for the flows require to go out through same port. It is because in router, flow entries are w.r.t destination address.

- Flood Ping(with packet size of 1KB) to only 1 host(1 flow gets matched) and execute normal Ping to same host with count 10



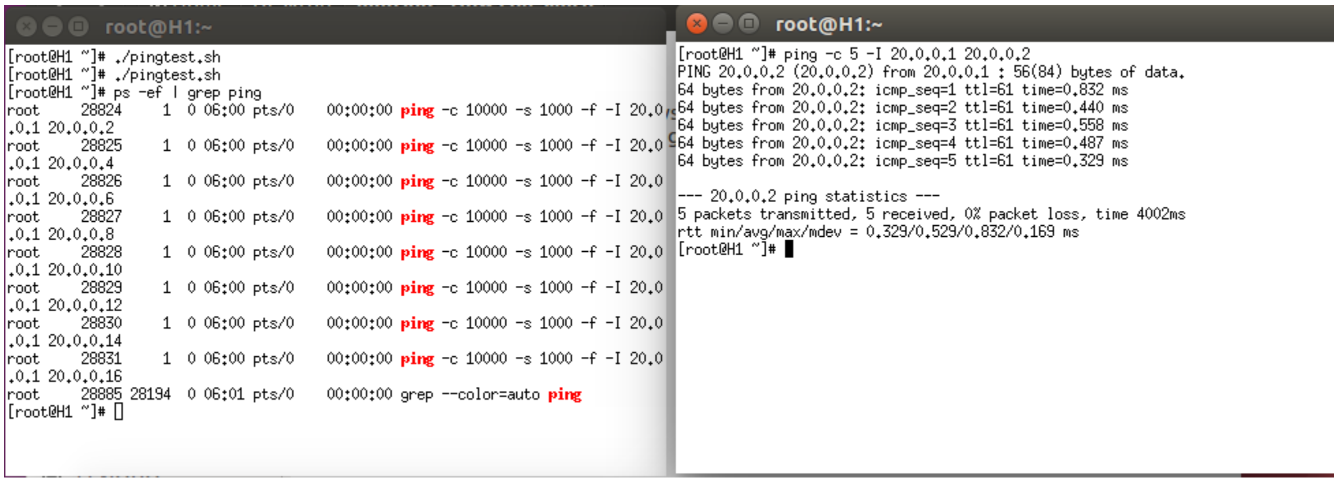
```
root@H1:~
[root@H1 ~]# ping -c 100000 -s 1000 -f -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 1000(1028) bytes of data.
..................................................................................
```

```
root@H1:~
Warning: Permanently added '139.62.242.10' (ECDSA) to the list of known hosts.
Last login: Mon Nov 28 05:59:08 2016 from 71.69.148.152
[root@H1 ~]# ping -c 10 -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 56(84) bytes of data.
64 bytes from 20.0.0.2: icmp_seq=1 ttl=61 time=0.720 ms
64 bytes from 20.0.0.2: icmp_seq=2 ttl=61 time=0.355 ms
64 bytes from 20.0.0.2: icmp_seq=3 ttl=61 time=0.371 ms
64 bytes from 20.0.0.2: icmp_seq=4 ttl=61 time=0.412 ms
64 bytes from 20.0.0.2: icmp_seq=5 ttl=61 time=0.527 ms
64 bytes from 20.0.0.2: icmp_seq=6 ttl=61 time=0.495 ms
64 bytes from 20.0.0.2: icmp_seq=7 ttl=61 time=0.379 ms
64 bytes from 20.0.0.2: icmp_seq=8 ttl=61 time=0.566 ms
64 bytes from 20.0.0.2: icmp_seq=9 ttl=61 time=0.734 ms
64 bytes from 20.0.0.2: icmp_seq=10 ttl=61 time=0.553 ms

--- 20.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9003ms
rtt min/avg/max/mdev = 0.355/0.511/0.734/0.131 ms
[root@H1 ~]#
```

We observe the average RTT to be **0.511 milliseconds**

- Flood Ping all end-hosts(all flows gets matched) and execute normal Ping to one of the flows with count 5

We execute the pingtest.sh script and do a normal ping with count as 5 with default echo_request size of 56 Bytes-



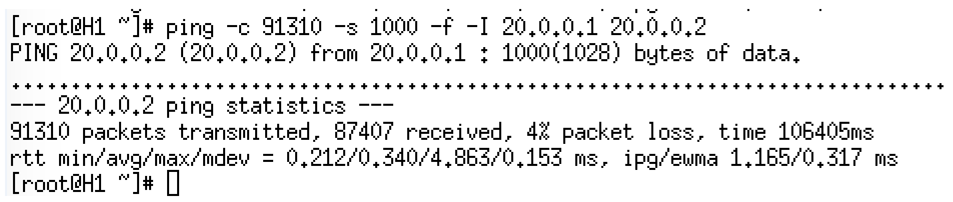
```
[root@H1 ~]# ./pingtest.sh
[root@H1 ~]# ./pingtest.sh
[root@H1 ~]# ps -ef | grep ping
root     28824     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.2
root     28825     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.4
root     28826     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.6
root     28827     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.8
root     28828     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.10
root     28829     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.12
root     28830     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.14
root     28831     1  0 06:00 pts/0    00:00:00 ping -c 10000 -s 1000 -f -I 20.0
.0.1 20.0.0.16
root     28885 28194  0 06:01 pts/0    00:00:00 grep --color=auto ping
[root@H1 ~]# 
```

```
[root@H1 ~]# ping -c 5 -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 56(84) bytes of data.
64 bytes from 20.0.0.2: icmp_seq=1 ttl=61 time=0.832 ms
64 bytes from 20.0.0.2: icmp_seq=2 ttl=61 time=0.440 ms
64 bytes from 20.0.0.2: icmp_seq=3 ttl=61 time=0.558 ms
64 bytes from 20.0.0.2: icmp_seq=4 ttl=61 time=0.487 ms
64 bytes from 20.0.0.2: icmp_seq=5 ttl=61 time=0.329 ms

--- 20.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4002ms
rtt min/avg/max/mdev = 0.329/0.529/0.832/0.169 ms
[root@H1 ~]# 
```

We observe the average RTT to be **0.529 milliseconds**

- Packet Loss Measurement-

Flood ping test with packets of size 1KB with count as 91310-



```
[root@H1 ~]# ping -c 91310 -s 1000 -f -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 1000(1028) bytes of data.
................................................................................
--- 20.0.0.2 ping statistics ---
91310 packets transmitted, 87407 received, 4% packet loss, time 106405ms
rtt min/avg/max/mdev = 0.212/0.340/4.863/0.153 ms, ipg/ewma 1.165/0.317 ms
[root@H1 ~]# 
```

We observe that out of 91310 packets 4% is lost i.e. only 87407 received.

This concludes the base-test with the normal rest_router.py application running on the controller.

Note: We will compare the stats which we got in this test case with the SimTP Application Performance test case 5.

Using Mpls Label for the SimTP Application (Manual Config)			
Test ID	**Action**	**Expected Observation**	**Conclusion**
3.1	Configure Flow that pushes a SimTP(MPLS) label at ingress of Ovs(LER) on the link between an OVS and end host. We manually configure the flow using ovs-ofctl.	This label should be present on the ICMP packet at the egress/output port of OVS for the flow	The Label is pushed onto the ICMP packets and this is verified by taking tcpdump and observing the packet at the egress port of OVS
3.2	Configure Flow that pops the SimTP(MPLS) label before transmission at egress of Ovs on the link between an OVS and end host. We manually configure the flow using ovs-ofctl.	No label should be present on the ICMP packet at the egress of OVS or at the ingress of the end host	The Label is popped from the ICMP packets before transmission onto the end host and this is verified by taking tcpdump and observing the packet at the ingress port of end host(h1,h2,h3)
3.3	Configure Flow on the intermediate OVS(LSR) that forward ICMP packets based on the SimTP(MPLS) label. We manually configure the flow using ovs-ofctl.	The ICMP packets towards a destination have the same SimTP Label on all Intermediate OVS(LSR's)	The ICMP packets are being forwarded based on the SimTP label and these labels are not switched at the Intermediate(LSR) OVS's
3.4	Execute ping between the end hosts	Ping Successful	Ping was successful implying that forwarding was solely based on the SimTP Label and we do not call it MPLS because it does not function as MPLS as there is no swapping of labels

Results-

The ping to the end-hosts are successful and it is label forwarded, where at the source LER we push a SimTP label, on the LSR- forwarded based on the label and at the destination LER- popping of the label and transmission of the packet to the destination device.

Ovs-ofctl commands is used to add the flows manually for the SimTP.

On Source LER-

For Host1 to Host2:

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=3,eth_type=0x800,nw_proto=1,actions=push_mpls:0x8847,set_field:12->mpls_label,output:1"
```

For Host2 to Host1:

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=1,eth_type=0x8847,mpls_bos=13,actions=pop_mpls:0x800,output:3"
```

LSR-

For Host1 to Host2:

```
sudo ovs-ofctl -O OpenFlow13 add-flow br0 "table=0,in_port=3,eth_type=0x8847,mpls_bos=12,
actions=output:2"
```

For Host2 to Host1:

```
sudo ovs-ofctl -O OpenFlow13 add-flow br0 "table=0,in_port=2,eth_type=0x8847,mpls_bos=13,
actions=output:3"
```

Destination LER-

For Host1 to Host2:

```
sudo ovs-ofctl -O OpenFlow13 add-flow br0
"table=0,in_port=1,eth_type=0x8847,mpls_bos=13,actions=pop_mpls:0x800,output:2"
```

For Hos2 to Host1:

```
sudo ovs-ofctl -O OpenFlow13 add-flow br0
"table=0,in_port=2,eth_type=0x800,nw_proto=1,actions=push_mpls:0x8847,set_field:13->mpls_label,output:
"
```

The above commands is useful only to ping from only one host to the other. We performed this test to see if we can forward based on mpls label without having to switch it at the LSR. The test on success shows us that we can use the MPLS Label header field for our SimTP Application.

Also, the wireshark captures showing the push and pop and forwarding based on MPLS label is demonstrated in the next test case where we use the SimTP application on the controller.

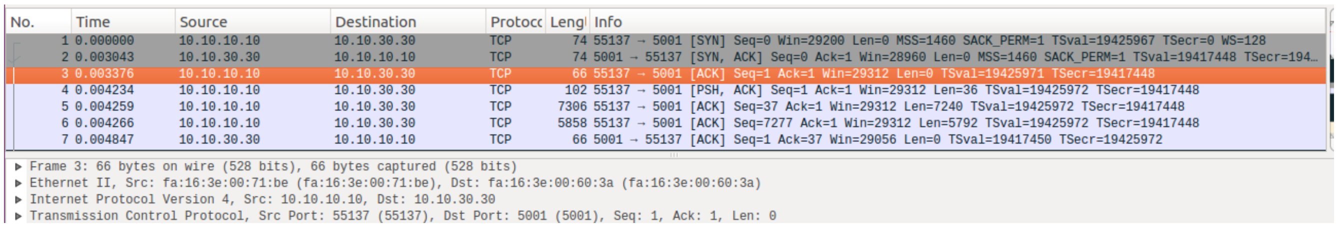
Automating the earlier test case by having Controller -RYU send the flows using FlowMod			
Test ID	**Action**	**Expected Observation**	**Conclusion**
4.1	Have the Controller send flows to push SimTP label for the ICMP Packets at the LER before transmission onto the link between OVS's	This label should be present on the ICMP packet at the egress/output port of OVS for the flow	The Label is pushed onto the ICMP packets and this is verified by taking tcpdump and observing the packet at the egress port of OVS
4.2	Have the Controller send flows to forward (and not switch label and forward) ICMP packets based on SimTP label at the LSR(or intermediate OVS)	The ICMP packets towards a destination have the same SimTP Label on all Intermediate OVS(LSR's)	The ICMP packets are being forwarded based on the SimTP label and these labels are not switched at the Intermediate(LSR) OVS's
4.3	Have the Controller send flows to pop SimTP label for the ICMP Packets at the LER before transmission to end-host	No label should be present on the ICMP packet at the egress of OVS or at the ingress of the end host	The Label is popped from the ICMP packets before transmission onto the end host and this is verified by taking tcpdump and observing the packet at the ingress port of end host(h1,h2,h3)
4.4	Execute ping between the end hosts	Ping Successful	Ping was successful implying that forwarding was solely based on the match criteria- SimTP Label.

Results-

We configured H1-10.10.10.10 as the client and H3-10.10.30.30 as the server and observing the tcpdump(through wireshark) to evaluate that the Tunnel Label is pushed at LER, forwarding based on Label at the LSR and POP again at the remote LER.

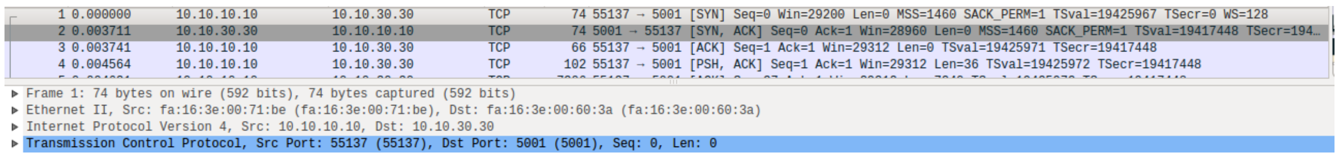
At the link between H1 and OVS1, we see below that there is no Tunnel-Label injected in both the request and reply-

Request (from 10.10.10.10 to 10.10.30.30)



```
No.  Time        Source        Destination   Protoco Lengl Info
   1 0.000000    10.10.10.10   10.10.30.30   TCP        74 55137 → 5001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19425967 TSecr=0 WS=128
   2 0.003043    10.10.30.30   10.10.10.10   TCP        74 5001 → 55137 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=19417448 TSecr=194…
   3 0.003376    10.10.10.10   10.10.30.30   TCP        66 55137 → 5001 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=19425971 TSecr=19417448
   4 0.004234    10.10.10.10   10.10.30.30   TCP       102 55137 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=36 TSval=19425972 TSecr=19417448
   5 0.004259    10.10.10.10   10.10.30.30   TCP      7306 55137 → 5001 [ACK] Seq=37 Ack=1 Win=29312 Len=7240 TSval=19425972 TSecr=19417448
   6 0.004266    10.10.10.10   10.10.30.30   TCP      5858 55137 → 5001 [ACK] Seq=7277 Ack=1 Win=29312 Len=5792 TSval=19425972 TSecr=19417448
   7 0.004847    10.10.30.30   10.10.10.10   TCP        66 5001 → 55137 [ACK] Seq=1 Ack=37 Win=29056 Len=0 TSval=19417450 TSecr=19425972
▶ Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
▶ Ethernet II, Src: fa:16:3e:00:71:be (fa:16:3e:00:71:be), Dst: fa:16:3e:00:60:3a (fa:16:3e:00:60:3a)
▶ Internet Protocol Version 4, Src: 10.10.10.10, Dst: 10.10.30.30
▶ Transmission Control Protocol, Src Port: 55137 (55137), Dst Port: 5001 (5001), Seq: 1, Ack: 1, Len: 0
```

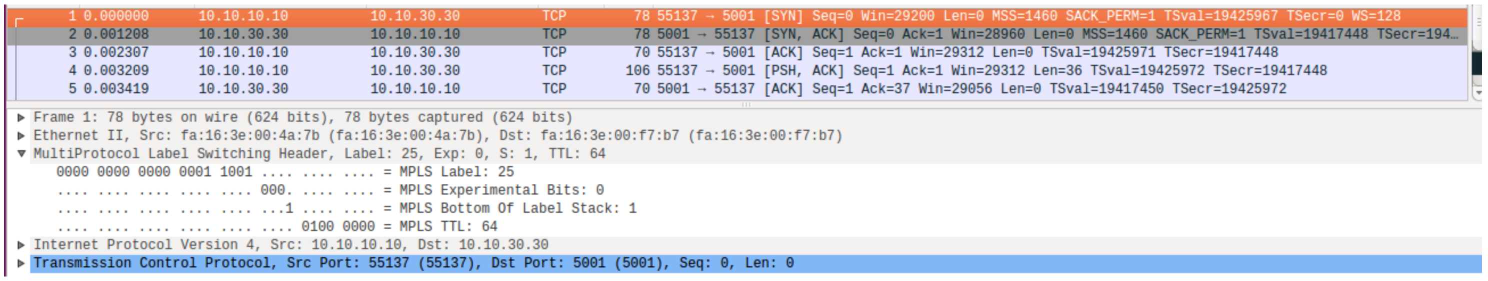
Reply (from 10.10.30.30 to 10.10.10.10)-



```
  1 0.000000    10.10.10.10     10.10.30.30     TCP      74 55137 → 5001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19425967 TSecr=0 WS=128
  2 0.003711    10.10.30.30     10.10.10.10     TCP      74 5001 → 55137 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=19417448 TSecr=194…
  3 0.003741    10.10.10.10     10.10.30.30     TCP      66 55137 → 5001 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=19425971 TSecr=19417448
  4 0.004564    10.10.10.10     10.10.30.30     TCP     102 55137 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=36 TSval=19425972 TSecr=19417448
▶ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
▶ Ethernet II, Src: fa:16:3e:00:71:be (fa:16:3e:00:71:be), Dst: fa:16:3e:00:60:3a (fa:16:3e:00:60:3a)
▶ Internet Protocol Version 4, Src: 10.10.10.10, Dst: 10.10.30.30
▶ Transmission Control Protocol, Src Port: 55137 (55137), Dst Port: 5001 (5001), Seq: 0, Len: 0
```

At the remote LER(that connects to the server H3) on the link towards the H3, we again observe that there is no label.

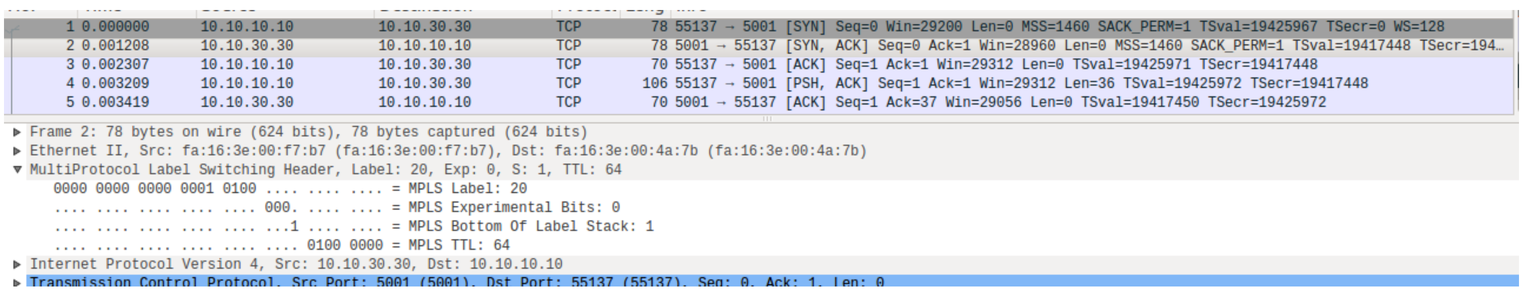
Now, we observe label on the intermediate **OVS2(LSR)**, on the link between OVS2 and OVS4-



```
  1 0.000000    10.10.10.10     10.10.30.30     TCP      78 55137 → 5001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19425967 TSecr=0 WS=128
  2 0.001208    10.10.30.30     10.10.10.10     TCP      78 5001 → 55137 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=19417448 TSecr=194…
  3 0.002307    10.10.10.10     10.10.30.30     TCP      70 55137 → 5001 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=19425971 TSecr=19417448
  4 0.003209    10.10.10.10     10.10.30.30     TCP     106 55137 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=36 TSval=19425972 TSecr=19417448
  5 0.003419    10.10.30.30     10.10.10.10     TCP      70 5001 → 55137 [ACK] Seq=1 Ack=37 Win=29056 Len=0 TSval=19417450 TSecr=19425972
▶ Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▶ Ethernet II, Src: fa:16:3e:00:4a:7b (fa:16:3e:00:4a:7b), Dst: fa:16:3e:00:f7:b7 (fa:16:3e:00:f7:b7)
▼ MultiProtocol Label Switching Header, Label: 25, Exp: 0, S: 1, TTL: 64
    0000 0000 0000 0001 1001 .... .... .... = MPLS Label: 25
    .... .... .... .... .... 000. .... .... = MPLS Experimental Bits: 0
    .... .... .... .... .... ...1 .... .... = MPLS Bottom Of Label Stack: 1
    .... .... .... .... .... .... 0100 0000 = MPLS TTL: 64
▶ Internet Protocol Version 4, Src: 10.10.10.10, Dst: 10.10.30.30
▶ Transmission Control Protocol, Src Port: 55137 (55137), Dst Port: 5001 (5001), Seq: 0, Len: 0
```

From above we observe the request(10.10.10.10 to 10.10.30.30) will be forwarded using label 25.

And the reply, which would be to a different destination(i.e. to 10.10.10.10) and would essentially have a different label i.e. Label 20-



```
  1 0.000000    10.10.10.10     10.10.30.30     TCP      78 55137 → 5001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19425967 TSecr=0 WS=128
  2 0.001208    10.10.30.30     10.10.10.10     TCP      78 5001 → 55137 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=19417448 TSecr=194…
  3 0.002307    10.10.10.10     10.10.30.30     TCP      70 55137 → 5001 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=19425971 TSecr=19417448
  4 0.003209    10.10.10.10     10.10.30.30     TCP     106 55137 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=36 TSval=19425972 TSecr=19417448
  5 0.003419    10.10.30.30     10.10.10.10     TCP      70 5001 → 55137 [ACK] Seq=1 Ack=37 Win=29056 Len=0 TSval=19417450 TSecr=19425972
▶ Frame 2: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▶ Ethernet II, Src: fa:16:3e:00:f7:b7 (fa:16:3e:00:f7:b7), Dst: fa:16:3e:00:4a:7b (fa:16:3e:00:4a:7b)
▼ MultiProtocol Label Switching Header, Label: 20, Exp: 0, S: 1, TTL: 64
    0000 0000 0000 0001 0100 .... .... .... = MPLS Label: 20
    .... .... .... .... .... 000. .... .... = MPLS Experimental Bits: 0
    .... .... .... .... .... ...1 .... .... = MPLS Bottom Of Label Stack: 1
    .... .... .... .... .... .... 0100 0000 = MPLS TTL: 64
▶ Internet Protocol Version 4, Src: 10.10.30.30, Dst: 10.10.10.10
▶ Transmission Control Protocol, Src Port: 5001 (5001), Dst Port: 55137 (55137), Seq: 0, Ack: 1, Len: 0
```

Also, we are using the MPLS header for the SimTP application, there is no label switching and only forwarding based on the label.

Below, we see a snip of the SimTP application running on the controller. We see the communication made by the OVS- through packet-in and the RYU responding with Packet-out.

```
[RT][INFO] switch_id=0000000000000004: Set L2 switching (normal) flow [cookie=0x
2]
port info :  fa:16:3e:00:f7:b7
port info :  fa:16:3e:00:f7:b7
port info :  fa:16:3e:00:f7:b7
port info :  fa:16:3e:00:f7:b7
port info :  fa:16:3e:00:f7:b7
port info :  fa:16:3e:00:f7:b7
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 228 0.006640
[RT][INFO] switch_id=0000000000000003: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000003: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000001: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000001: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000004: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000004: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000004: Receive ARP reply from [172.16.24.1] to r
outer port [172.16.24.2].
(2510) accepted ('127.0.0.1', 54648)
[RT][INFO] switch_id=0000000000000002: Launching Vlan_router PacketIn handler
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 234 0.003868
(2510) accepted ('127.0.0.1', 54650)
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 234 0.000712
(2510) accepted ('127.0.0.1', 54652)
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 234 0.000737
(2510) accepted ('127.0.0.1', 54654)
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 234 0.000732
(2510) accepted ('127.0.0.1', 54656)
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 234 0.002685
(2510) accepted ('127.0.0.1', 54658)
127.0.0.1 - - [26/Nov/2016 23:23:27] "POST /router/0000000000000004 HTTP/1.1" 20
0 234 0.000797
[RT][INFO] switch_id=0000000000000001: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000001: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000001: Receive ARP request from [10.10.10.10] to
 router port [10.10.10.1].
[RT][INFO] switch_id=0000000000000001: Send ARP reply to [10.10.10.10]
[RT][INFO] switch_id=0000000000000004: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000004: Set implicit routing flow [cookie=0x2]
[RT][INFO] switch_id=0000000000000004: Receive ARP request from [10.10.30.30] to
 router port [10.10.30.1].
[RT][INFO] switch_id=0000000000000004: Send ARP reply to [10.10.30.30]
[RT][INFO] switch_id=0000000000000004: Launching Vlan_router PacketIn handler
[RT][INFO] switch_id=0000000000000004: Set implicit routing flow [cookie=0x2]
[RT][INFO] switch_id=0000000000000004: Receive ARP request from [10.10.30.30] to
 router port [10.10.30.1].
[RT][INFO] switch_id=0000000000000004: Send ARP reply to [10.10.30.30]
[RT][INFO] switch_id=0000000000000001: Launching Vlan_router PacketIn handler
```

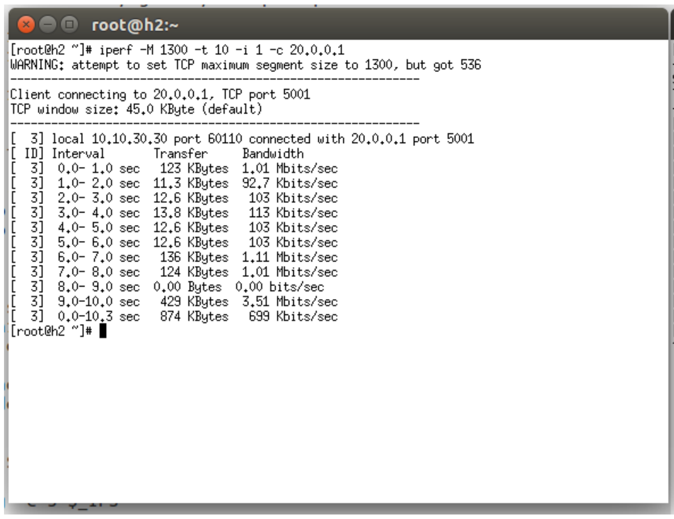
Delay and Performance			
Test ID	**Action**	**Expected Observation**	**Conclusion**
5.1	Configure H1 as client and H2 as server and source an infinite stream of TCP Packets using iperf	Packet gets labeled and are forwarded based on SimTP Label towards the server	Successful transmissions implying the SimTP application on controller adds the respective flows.
5.2	Now observe the the two way delay associated with the channel(H1 to H2 and H2 to H1)	We should observe the delay to be lesser as compared to the delay involved with forwarding based on ip destination prefix in the same topology.	This implies that the lookup is faster based on 20 bit SimTP(MPLS) label than the 32 bit IP destination prefix.
5.3	Now we introduce equal traffic on all flows and observe the delay	We expect lesser latency for traffic of each flow	The 20 bit lookup and the reduced number of flows improves performance in terms of decreased latency
5.4	Now we introduce high traffic on one flow and send few ICMP packets on one of the other flow	We expect the ICMP response to be quicker i.e. lesser latency	There is an improvement in performance due to faster lookup and lesser number of flows present in flow table

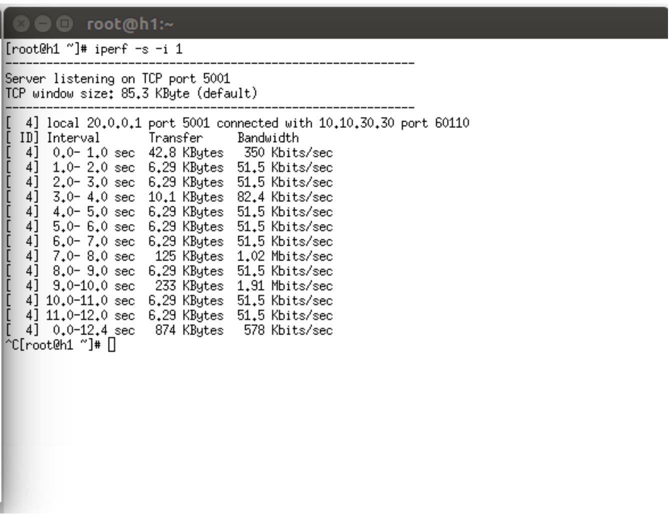
Results-

TCP-
H1-20.0.0.1 is the tcp server and H2- 20.0.0.2 is the client and we specify the MTU of 1300B, because using the 1500B MTU results in fragmentation of the SimTP(MPLS) packets sent and hence resulting in losses.

We notice a little decrease in performance as compared to the base test case. We believe this is because, our application code is not optimized as compared to the router.py controller application. Also, the support for jumbo frames on linux host links i.e. mpls is not as optimized.

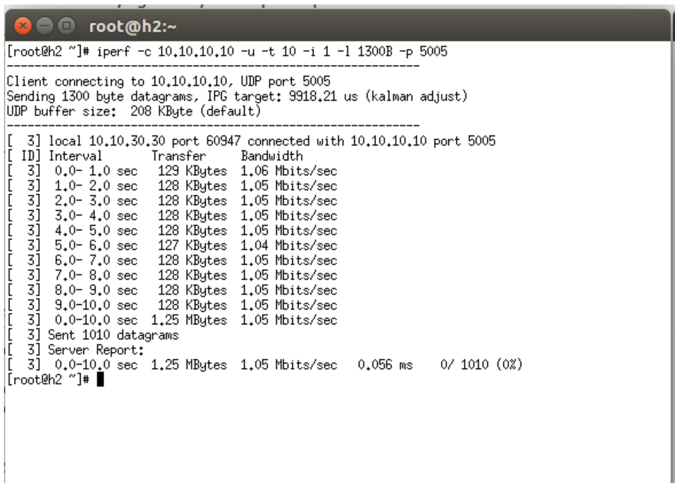
Another reason the default window size for the client is 45Kbytes and for the server is 85.3 Kbytes and this value is lesser as compared to the base test case where the window size was 83 Kbytes on both client and server.


```
root@h2:~
[root@h2 ~]# iperf -M 1300 -t 10 -i 1 -c 20.0.0.1
WARNING: attempt to set TCP maximum segment size to 1300, but got 536
------------------------------------------------------------
Client connecting to 20.0.0.1, TCP port 5001
TCP window size: 45.0 KByte (default)
------------------------------------------------------------
[  3] local 10.10.30.30 port 60110 connected with 20.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   123 KBytes  1.01 Mbits/sec
[  3]  1.0- 2.0 sec  11.3 KBytes  92.7 Kbits/sec
[  3]  2.0- 3.0 sec  12.6 KBytes   103 Kbits/sec
[  3]  3.0- 4.0 sec  13.8 KBytes   113 Kbits/sec
[  3]  4.0- 5.0 sec  12.6 KBytes   103 Kbits/sec
[  3]  5.0- 6.0 sec  12.6 KBytes   103 Kbits/sec
[  3]  6.0- 7.0 sec   136 KBytes  1.11 Mbits/sec
[  3]  7.0- 8.0 sec   124 KBytes  1.01 Mbits/sec
[  3]  8.0- 9.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  9.0-10.0 sec   429 KBytes  3.51 Mbits/sec
[  3]  0.0-10.3 sec   874 KBytes   699 Kbits/sec
[root@h2 ~]# 
```

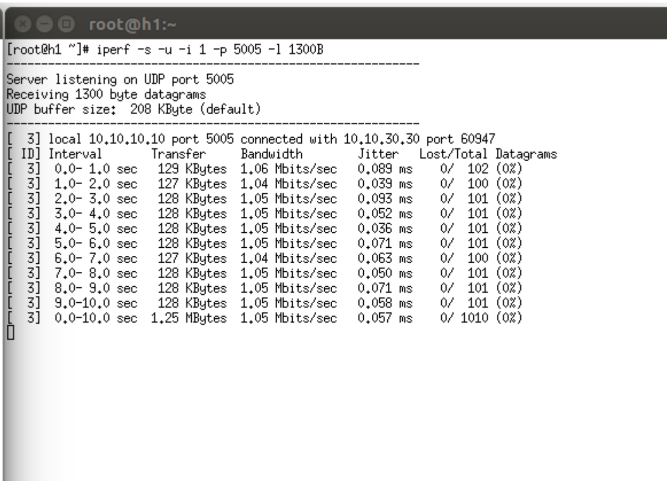

```
root@h1:~
[root@h1 ~]# iperf -s -i 1
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 20.0.0.1 port 5001 connected with 10.10.30.30 port 60110
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 1.0 sec  42.8 KBytes   350 Kbits/sec
[  4]  1.0- 2.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  2.0- 3.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  3.0- 4.0 sec  10.1 KBytes  82.4 Kbits/sec
[  4]  4.0- 5.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  5.0- 6.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  6.0- 7.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  7.0- 8.0 sec   125 KBytes  1.02 Mbits/sec
[  4]  8.0- 9.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  9.0-10.0 sec   233 KBytes  1.91 Mbits/sec
[  4] 10.0-11.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4] 11.0-12.0 sec  6.29 KBytes  51.5 Kbits/sec
[  4]  0.0-12.4 sec   874 KBytes   578 Kbits/sec
^C[root@h1 ~]# 
```

UDP-

H1-20.0.0.1 is the udp server and H2- 20.0.0.2 is the udp client.



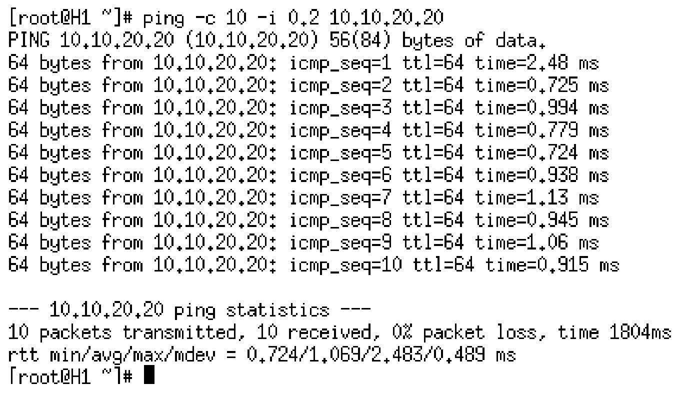
```
[root@h2 ~]# iperf -c 10.10.10.10 -u -t 10 -i 1 -l 1300B -p 5005
------------------------------------------------------------
Client connecting to 10.10.10.10, UDP port 5005
Sending 1300 byte datagrams, IPG target: 9918.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.10.30.30 port 60947 connected with 10.10.10.10 port 5005
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   129 KBytes  1.06 Mbits/sec
[  3]  1.0- 2.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  2.0- 3.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  3.0- 4.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  4.0- 5.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  5.0- 6.0 sec   127 KBytes  1.04 Mbits/sec
[  3]  6.0- 7.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  7.0- 8.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  8.0- 9.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  9.0-10.0 sec   128 KBytes  1.05 Mbits/sec
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 1010 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.056 ms    0/ 1010 (0%)
[root@h2 ~]#
```



```
[root@h1 ~]# iperf -s -u -i 1 -p 5005 -l 1300B
------------------------------------------------------------
Server listening on UDP port 5005
Receiving 1300 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.10.10.10 port 5005 connected with 10.10.30.30 port 60947
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0- 1.0 sec   129 KBytes  1.06 Mbits/sec   0.089 ms    0/  102 (0%)
[  3]  1.0- 2.0 sec   127 KBytes  1.04 Mbits/sec   0.039 ms    0/  100 (0%)
[  3]  2.0- 3.0 sec   128 KBytes  1.05 Mbits/sec   0.093 ms    0/  101 (0%)
[  3]  3.0- 4.0 sec   128 KBytes  1.05 Mbits/sec   0.052 ms    0/  101 (0%)
[  3]  4.0- 5.0 sec   128 KBytes  1.05 Mbits/sec   0.036 ms    0/  101 (0%)
[  3]  5.0- 6.0 sec   128 KBytes  1.05 Mbits/sec   0.071 ms    0/  101 (0%)
[  3]  6.0- 7.0 sec   127 KBytes  1.04 Mbits/sec   0.063 ms    0/  100 (0%)
[  3]  7.0- 8.0 sec   128 KBytes  1.05 Mbits/sec   0.050 ms    0/  101 (0%)
[  3]  8.0- 9.0 sec   128 KBytes  1.05 Mbits/sec   0.071 ms    0/  101 (0%)
[  3]  9.0-10.0 sec   128 KBytes  1.05 Mbits/sec   0.058 ms    0/  101 (0%)
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.057 ms    0/ 1010 (0%)
```

The average UDP sending rate is 1.05 Mbits/sec and the packet loss is 0 which is the same as the base test case. We observe an increase in the average jitter to 56 microseconds and we believe the reason for it is lack of optimized support on linux interfaces for jumbo frames.

- We use the Ping utility to determine the average RTT-

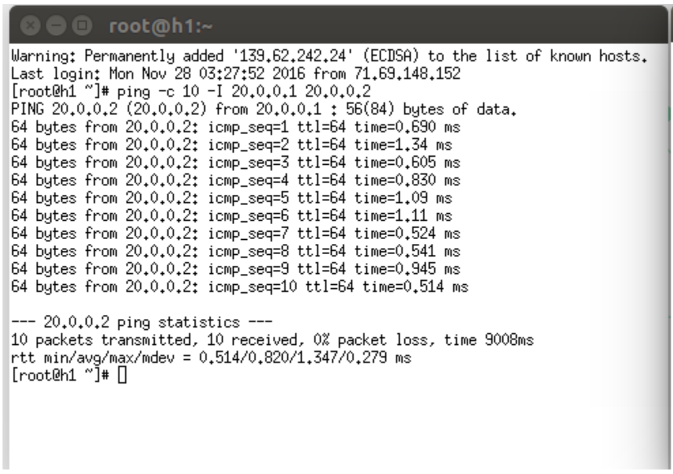


```
[root@H1 ~]# ping -c 10 -i 0.2 10.10.20.20
PING 10.10.20.20 (10.10.20.20) 56(84) bytes of data.
64 bytes from 10.10.20.20: icmp_seq=1 ttl=64 time=2.48 ms
64 bytes from 10.10.20.20: icmp_seq=2 ttl=64 time=0.725 ms
64 bytes from 10.10.20.20: icmp_seq=3 ttl=64 time=0.994 ms
64 bytes from 10.10.20.20: icmp_seq=4 ttl=64 time=0.779 ms
64 bytes from 10.10.20.20: icmp_seq=5 ttl=64 time=0.724 ms
64 bytes from 10.10.20.20: icmp_seq=6 ttl=64 time=0.938 ms
64 bytes from 10.10.20.20: icmp_seq=7 ttl=64 time=1.13 ms
64 bytes from 10.10.20.20: icmp_seq=8 ttl=64 time=0.945 ms
64 bytes from 10.10.20.20: icmp_seq=9 ttl=64 time=1.06 ms
64 bytes from 10.10.20.20: icmp_seq=10 ttl=64 time=0.915 ms

--- 10.10.20.20 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 1804ms
rtt min/avg/max/mdev = 0.724/1.069/2.483/0.489 ms
[root@H1 ~]#
```

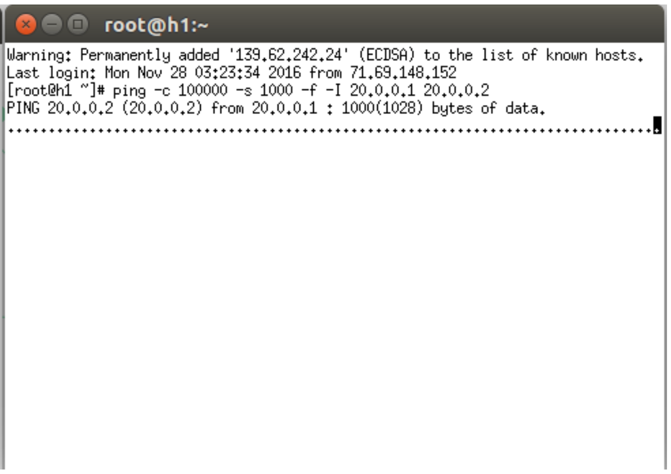
From above we observe an average RTT of 1.069 ms which as an improvement from the base test case of 1.282 ms.

Now we introduce many flows on the OVS's by introducing 10 loopbacks on H1(20.0.0.1,20.0.03…) with odd IP addresses and 10 loopbacks on H2 with even IP addresses (20.0.0.2, 20.0.0.4….). Then perform the below tests--

- Flood Ping(with packet size of 1KB) only 1 host(1 flow gets matched) and execute normal Ping to same host with count 10



```
Warning: Permanently added '139.62.242.24' (ECDSA) to the list of known hosts.
Last login: Mon Nov 28 03:27:52 2016 from 71.69.148.152
[root@h1 ~]# ping -c 10 -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 56(84) bytes of data.
64 bytes from 20.0.0.2: icmp_seq=1 ttl=64 time=0.690 ms
64 bytes from 20.0.0.2: icmp_seq=2 ttl=64 time=1.34 ms
64 bytes from 20.0.0.2: icmp_seq=3 ttl=64 time=0.605 ms
64 bytes from 20.0.0.2: icmp_seq=4 ttl=64 time=0.830 ms
64 bytes from 20.0.0.2: icmp_seq=5 ttl=64 time=1.09 ms
64 bytes from 20.0.0.2: icmp_seq=6 ttl=64 time=1.11 ms
64 bytes from 20.0.0.2: icmp_seq=7 ttl=64 time=0.524 ms
64 bytes from 20.0.0.2: icmp_seq=8 ttl=64 time=0.541 ms
64 bytes from 20.0.0.2: icmp_seq=9 ttl=64 time=0.945 ms
64 bytes from 20.0.0.2: icmp_seq=10 ttl=64 time=0.514 ms

--- 20.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9008ms
rtt min/avg/max/mdev = 0.514/0.820/1.347/0.279 ms
[root@h1 ~]# 
```

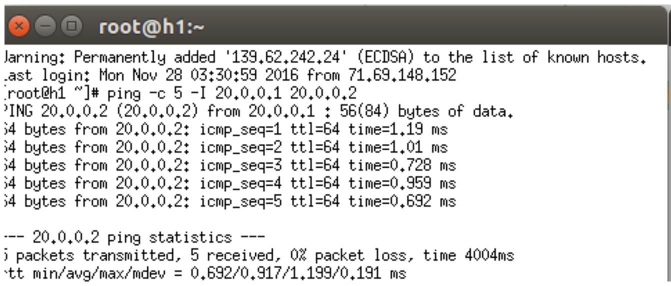


```
Warning: Permanently added '139.62.242.24' (ECDSA) to the list of known hosts.
Last login: Mon Nov 28 03:23:34 2016 from 71.69.148.152
[root@h1 ~]# ping -c 100000 -s 1000 -f -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 1000(1028) bytes of data.
...............................................................................
```

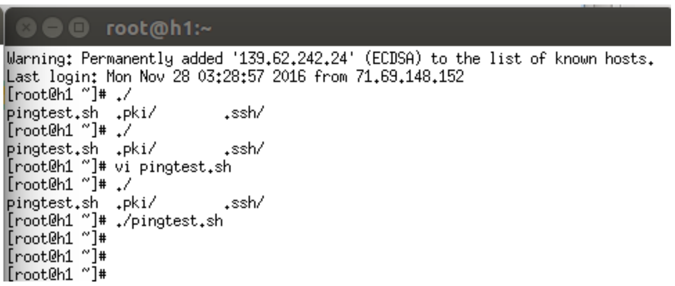
We observe an increase in the RTT from 0.511ms of the best test case to 0.82ms. We believe the reason for this is the lack of optimization of our code on the controller and also the additional header overhead in packets as they now contain an SimTP(MPLS) label and we estimate to see an improvement on an extremely large topology of nodes which are completely utilized on every link much like the production network environment.
Also the SimTP provides better performance when compared with actual MPLS code as there is no label swapping involved.

- Flood Ping all end-hosts(all flows gets matched) and execute normal Ping to one of the flows with count 5

We execute the pingtest.sh script and do a normal ping with count as 5 with default echo_request size of 56 Bytes-



```
Jarning: Permanently added '139.62.242.24' (ECDSA) to the list of known hosts.
.ast login: Mon Nov 28 03:30:59 2016 from 71.69.148.152
'root@h1 ~]# ping -c 5 -I 20.0.0.1 20.0.0.2
'ING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 56(84) bytes of data.
54 bytes from 20.0.0.2: icmp_seq=1 ttl=64 time=1.19 ms
54 bytes from 20.0.0.2: icmp_seq=2 ttl=64 time=1.01 ms
54 bytes from 20.0.0.2: icmp_seq=3 ttl=64 time=0.728 ms
54 bytes from 20.0.0.2: icmp_seq=4 ttl=64 time=0.959 ms
54 bytes from 20.0.0.2: icmp_seq=5 ttl=64 time=0.692 ms

-- 20.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
'tt min/avg/max/mdev = 0.692/0.917/1.199/0.191 ms
```

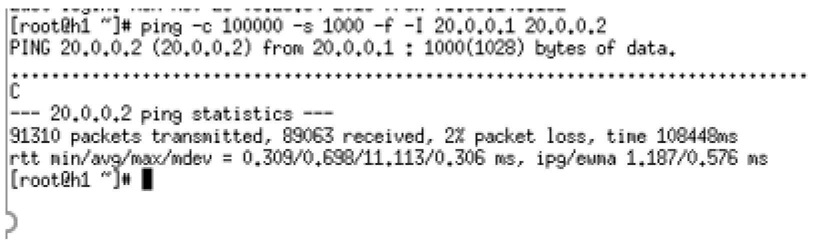


```
Warning: Permanently added '139.62.242.24' (ECDSA) to the list of known hosts.
Last login: Mon Nov 28 03:28:57 2016 from 71.69.148.152
[root@h1 ~]# ./
pingtest.sh  .pki/        .ssh/
[root@h1 ~]# ./
pingtest.sh  .pki/        .ssh/
[root@h1 ~]# vi pingtest.sh
[root@h1 ~]# ./
pingtest.sh  .pki/        .ssh/
[root@h1 ~]# ./pingtest.sh
[root@h1 ~]#
[root@h1 ~]#
[root@h1 ~]#
```

We observe an increase in the RTT from 0.529ms of the best test case to 0.917ms.

- Packet Loss Measurement-

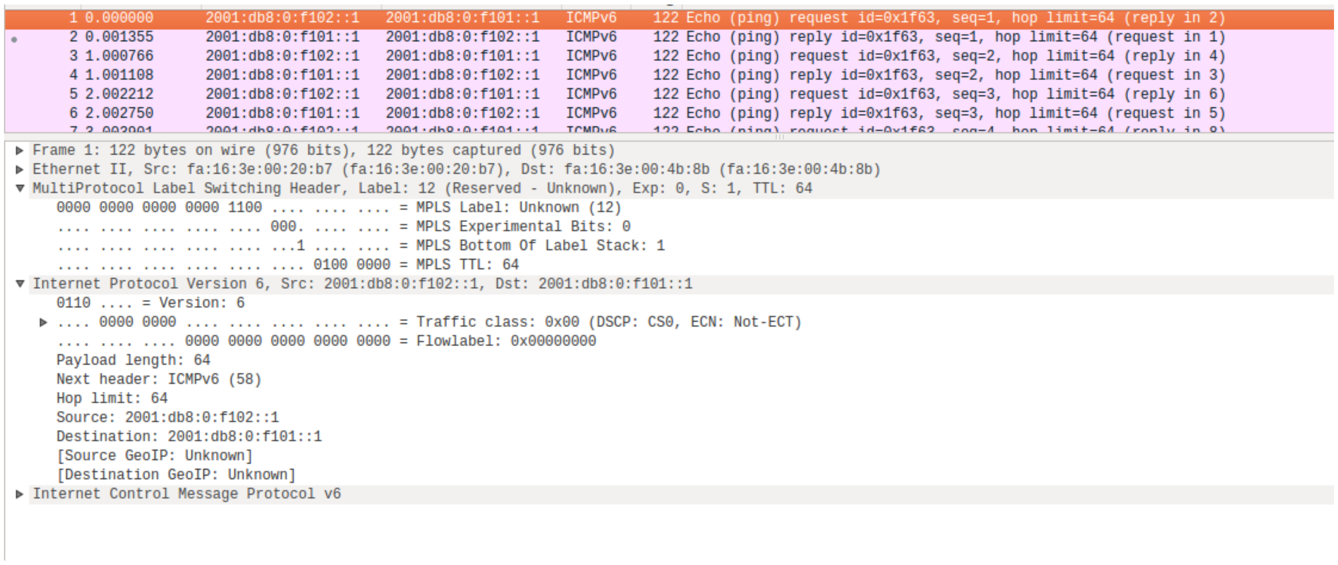
Flood ping test with packets of size 1KB with count as 91310-



```
[root@h1 ~]# ping -c 100000 -s 1000 -f -I 20.0.0.1 20.0.0.2
PING 20.0.0.2 (20.0.0.2) from 20.0.0.1 : 1000(1028) bytes of data.
.............................................................................
C
--- 20.0.0.2 ping statistics ---
91310 packets transmitted, 89063 received, 2% packet loss, time 108448ms
rtt min/avg/max/mdev = 0.309/0.698/11.113/0.306 ms, ipg/ewma 1.187/0.576 ms
[root@h1 ~]# 
```

We observe that out of 91310 packets **2% is lost** i.e. 89063 received which is an improvement from our base test case which gave 4% loss as only 87407 was received.

Payload Agnostic			
Test ID	**Action**	**Expected Observation**	**Conclusion**
6.1	Configure H1 and H3 with IPv6 address and source ICMPv6 packets from H1 to H3	Successful ping even though we have an Ipv4 core network	This is because the Ipv6 packets being encapsulated in SimTp label and forwarding based on the label rather than the ipv6 destination prefix



```
    1 0.000000     2001:db8:0:f102::1   2001:db8:0:f101::1   ICMPv6    122 Echo (ping) request id=0x1f63, seq=1, hop limit=64 (reply in 2)
    2 0.001355     2001:db8:0:f101::1   2001:db8:0:f102::1   ICMPv6    122 Echo (ping) reply id=0x1f63, seq=1, hop limit=64 (request in 1)
    3 1.000766     2001:db8:0:f102::1   2001:db8:0:f101::1   ICMPv6    122 Echo (ping) request id=0x1f63, seq=2, hop limit=64 (reply in 4)
    4 1.001108     2001:db8:0:f101::1   2001:db8:0:f102::1   ICMPv6    122 Echo (ping) reply id=0x1f63, seq=2, hop limit=64 (request in 3)
    5 2.002212     2001:db8:0:f102::1   2001:db8:0:f101::1   ICMPv6    122 Echo (ping) request id=0x1f63, seq=3, hop limit=64 (reply in 6)
    6 2.002750     2001:db8:0:f101::1   2001:db8:0:f102::1   ICMPv6    122 Echo (ping) reply id=0x1f63, seq=3, hop limit=64 (request in 5)

▶ Frame 1: 122 bytes on wire (976 bits), 122 bytes captured (976 bits)
▶ Ethernet II, Src: fa:16:3e:00:20:b7 (fa:16:3e:00:20:b7), Dst: fa:16:3e:00:4b:8b (fa:16:3e:00:4b:8b)
▼ MultiProtocol Label Switching Header, Label: 12 (Reserved - Unknown), Exp: 0, S: 1, TTL: 64
    0000 0000 0000 0000 1100 .... .... .... = MPLS Label: Unknown (12)
    .... .... .... .... .... 000. .... .... = MPLS Experimental Bits: 0
    .... .... .... .... .... ...1 .... .... = MPLS Bottom Of Label Stack: 1
    .... .... .... .... .... .... 0100 0000 = MPLS TTL: 64
▼ Internet Protocol Version 6, Src: 2001:db8:0:f102::1, Dst: 2001:db8:0:f101::1
    0110 .... = Version: 6
  ▶ .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 64
    Next header: ICMPv6 (58)
    Hop limit: 64
    Source: 2001:db8:0:f102::1
    Destination: 2001:db8:0:f101::1
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▶ Internet Control Message Protocol v6
```

Note- The above test was performed by manual addition of flows and not using the simtp controller application. The flows added would be the same as what the controller would add. We are yet to incorporate ipv6 packet handler in the simtp application due to time constraint.

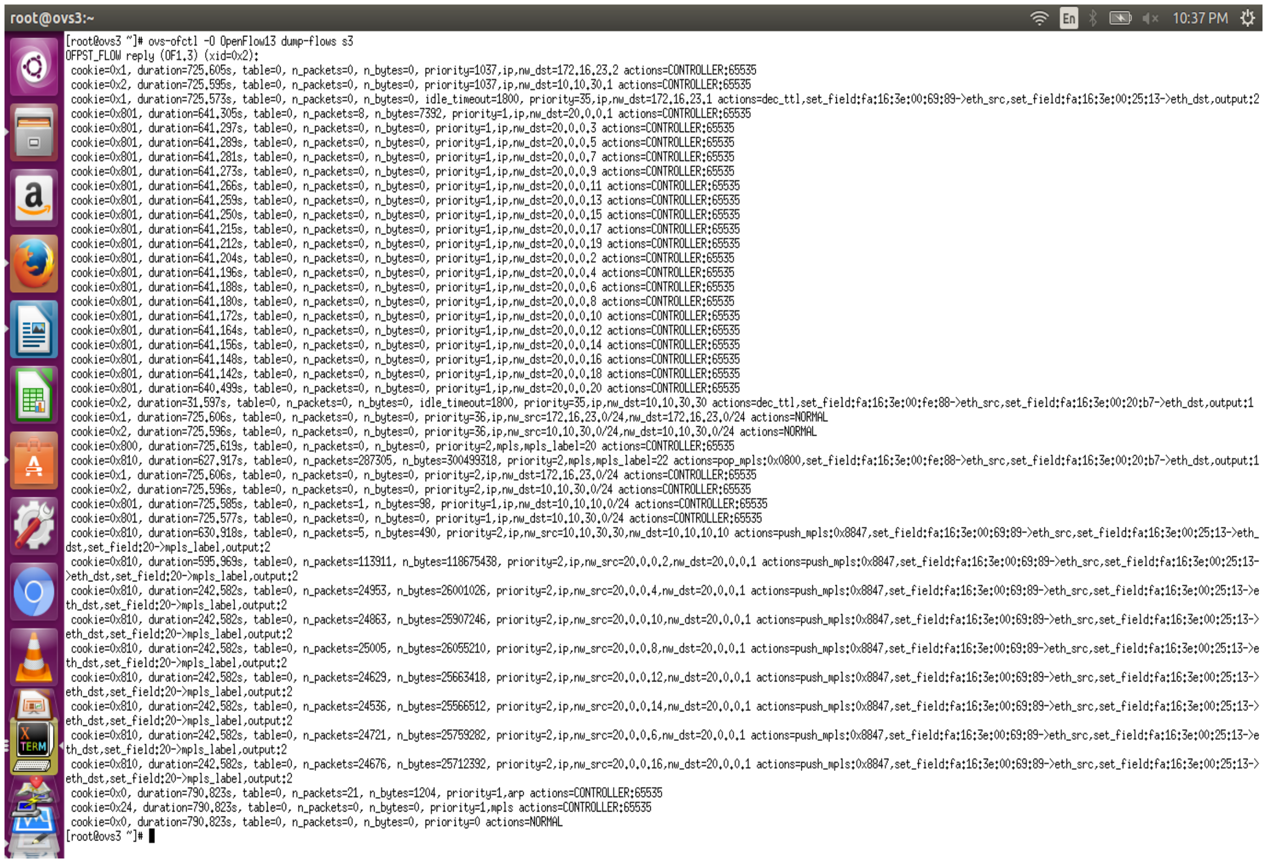
We take the above capture in the intermediate switch- OVS2. We notice above the label 12 gets attached and forwarding happens with this respect to this label. Though our Network does not have IPv6 configured- except on the end-hosts, we are still able to transport the traffic to remote end-Thanks to overlay tunneling.

Reduction in flow on intermediate OVS			
Test ID	**Action**	**Expected Observation**	**Conclusion**
7.1	Configure H2 to have 10 even IP addresses and H3 to have 10 odd IP addresses by using loopback on the end host Ethernet port, hence incorporating 20 destination prefixes	Successfully incorporated multiple end hosts having gateway as 1 Ethernet interface	We can proceed with the end goal for the test case
7.2	Now the intermediate ovs2 will install only two flows. One flow with matching based on SimTP even label sending it to H2 as action and another matching based on SimTP odd label sending it to H3 as action.	Ping is successful in the case for reaching the odd destination IPs and in the case for reaching the even destination IPs	Instead of introducing 200 distinct flows in the flow-table, we save the flow table memory as well as aid performance by enhancing the lookup time and hence would expect lesser latency for each ping.

Results-

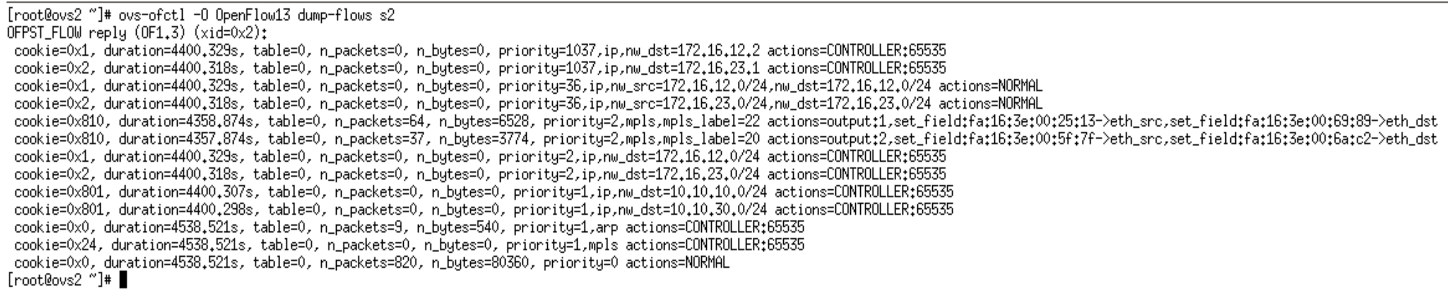
The LER would have the same number of flows as the normal -rest_routing.py application. However, on the LSR **we would observe much lesser number of flows and hence resulting in saving the flow-table size.** Below is a snip of the LER and LSR flow-table-

LER-



```
[root@ovs3 ~]# ovs-ofctl -O OpenFlow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=725.605s, table=0, n_packets=0, n_bytes=0, priority=1037,ip,nw_dst=172.16.23.2 actions=CONTROLLER:65535
 cookie=0x2, duration=725.595s, table=0, n_packets=0, n_bytes=0, priority=1037,ip,nw_dst=10.10.30.1 actions=CONTROLLER:65535
 cookie=0x1, duration=725.573s, table=0, n_packets=0, n_bytes=0, idle_timeout=1800, priority=35,ip,nw_dst=172.16.23.1 actions=dec_ttl,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->eth_dst,output:2
 cookie=0x801, duration=641.305s, table=0, n_packets=8, n_bytes=7392, priority=1,ip,nw_dst=20.0.0.1 actions=CONTROLLER:65535
 cookie=0x801, duration=641.297s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.3 actions=CONTROLLER:65535
 cookie=0x801, duration=641.289s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.5 actions=CONTROLLER:65535
 cookie=0x801, duration=641.281s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.7 actions=CONTROLLER:65535
 cookie=0x801, duration=641.273s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.9 actions=CONTROLLER:65535
 cookie=0x801, duration=641.266s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.11 actions=CONTROLLER:65535
 cookie=0x801, duration=641.259s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.13 actions=CONTROLLER:65535
 cookie=0x801, duration=641.250s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.15 actions=CONTROLLER:65535
 cookie=0x801, duration=641.215s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.17 actions=CONTROLLER:65535
 cookie=0x801, duration=641.212s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.19 actions=CONTROLLER:65535
 cookie=0x801, duration=641.204s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.2 actions=CONTROLLER:65535
 cookie=0x801, duration=641.196s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.4 actions=CONTROLLER:65535
 cookie=0x801, duration=641.188s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.6 actions=CONTROLLER:65535
 cookie=0x801, duration=641.180s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.8 actions=CONTROLLER:65535
 cookie=0x801, duration=641.172s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.10 actions=CONTROLLER:65535
 cookie=0x801, duration=641.164s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.12 actions=CONTROLLER:65535
 cookie=0x801, duration=641.156s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.14 actions=CONTROLLER:65535
 cookie=0x801, duration=641.148s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.16 actions=CONTROLLER:65535
 cookie=0x801, duration=641.142s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.18 actions=CONTROLLER:65535
 cookie=0x801, duration=640.499s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=20.0.0.20 actions=CONTROLLER:65535
 cookie=0x2, duration=31.597s, table=0, n_packets=0, n_bytes=0, idle_timeout=1800, priority=35,ip,nw_dst=10.10.30.30 actions=dec_ttl,set_field:fa:16:3e:00:fe:88->eth_src,set_field:fa:16:3e:00:20:b7->eth_dst,output:1
 cookie=0x1, duration=725.606s, table=0, n_packets=0, n_bytes=0, priority=36,ip,nw_src=172.16.23.0/24,nw_dst=172.16.23.0/24 actions=NORMAL
 cookie=0x2, duration=725.596s, table=0, n_packets=0, n_bytes=0, priority=36,ip,nw_src=10.10.30.0/24,nw_dst=10.10.30.0/24 actions=NORMAL
 cookie=0x800, duration=725.619s, table=0, n_packets=0, n_bytes=0, priority=2,mpls,mpls_label=20 actions=CONTROLLER:65535
 cookie=0x810, duration=627.917s, table=0, n_packets=287305, n_bytes=300499318, priority=2,mpls,mpls_label=22 actions=pop_mpls:0x0800,set_field:fa:16:3e:00:fe:88->eth_src,set_field:fa:16:3e:00:20:b7->eth_dst,output:1
 cookie=0x1, duration=725.606s, table=0, n_packets=0, n_bytes=0, priority=2,ip,nw_dst=172.16.23.0/24 actions=CONTROLLER:65535
 cookie=0x2, duration=725.596s, table=0, n_packets=0, n_bytes=0, priority=2,ip,nw_dst=10.10.30.0/24 actions=CONTROLLER:65535
 cookie=0x801, duration=725.585s, table=0, n_packets=1, n_bytes=98, priority=1,ip,nw_dst=10.10.10.0/24 actions=CONTROLLER:65535
 cookie=0x801, duration=725.577s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=10.10.30.0/24 actions=CONTROLLER:65535
 cookie=0x810, duration=630.918s, table=0, n_packets=5, n_bytes=490, priority=2,ip,nw_src=10.10.30.30,nw_dst=10.10.10.10 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->eth_
dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=595.969s, table=0, n_packets=113911, n_bytes=118675438, priority=2,ip,nw_src=20.0.0.2,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13-
>eth_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=24953, n_bytes=26001026, priority=2,ip,nw_src=20.0.0.4,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->e
th_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=24863, n_bytes=25907246, priority=2,ip,nw_src=20.0.0.10,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->
eth_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=25005, n_bytes=26055210, priority=2,ip,nw_src=20.0.0.8,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->e
th_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=24629, n_bytes=25663418, priority=2,ip,nw_src=20.0.0.12,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->
eth_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=24536, n_bytes=25566512, priority=2,ip,nw_src=20.0.0.14,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->
eth_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=24721, n_bytes=25759282, priority=2,ip,nw_src=20.0.0.6,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->e
th_dst,set_field:20->mpls_label,output:2
 cookie=0x810, duration=242.582s, table=0, n_packets=24676, n_bytes=25712392, priority=2,ip,nw_src=20.0.0.16,nw_dst=20.0.0.1 actions=push_mpls:0x8847,set_field:fa:16:3e:00:69:89->eth_src,set_field:fa:16:3e:00:25:13->
eth_dst,set_field:20->mpls_label,output:2
 cookie=0x0, duration=790.823s, table=0, n_packets=21, n_bytes=1204, priority=1,arp actions=CONTROLLER:65535
 cookie=0x24, duration=790.823s, table=0, n_packets=0, n_bytes=0, priority=1,mpls actions=CONTROLLER:65535
 cookie=0x0, duration=790.823s, table=0, n_packets=0, n_bytes=0, priority=0 actions=NORMAL
[root@ovs3 ~]# █
```

However, on the LSR-



```
[root@ovs2 ~]# ovs-ofctl -O OpenFlow13 dump-flows s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=4400.329s, table=0, n_packets=0, n_bytes=0, priority=1037,ip,nw_dst=172.16.12.2 actions=CONTROLLER:65535
 cookie=0x2, duration=4400.318s, table=0, n_packets=0, n_bytes=0, priority=1037,ip,nw_dst=172.16.23.1 actions=CONTROLLER:65535
 cookie=0x1, duration=4400.329s, table=0, n_packets=0, n_bytes=0, priority=36,ip,nw_src=172.16.12.0/24,nw_dst=172.16.12.0/24 actions=NORMAL
 cookie=0x2, duration=4400.318s, table=0, n_packets=0, n_bytes=0, priority=36,ip,nw_src=172.16.23.0/24,nw_dst=172.16.23.0/24 actions=NORMAL
 cookie=0x810, duration=4358.874s, table=0, n_packets=64, n_bytes=6528, priority=2,mpls,mpls_label=22 actions=output:1,set_field:fa:16:3e:00:25:13->eth_src,set_field:fa:16:3e:00:69:89->eth_dst
 cookie=0x810, duration=4357.874s, table=0, n_packets=37, n_bytes=3774, priority=2,mpls,mpls_label=20 actions=output:2,set_field:fa:16:3e:00:5f:7f->eth_src,set_field:fa:16:3e:00:6a:c2->eth_dst
 cookie=0x1, duration=4400.329s, table=0, n_packets=0, n_bytes=0, priority=2,ip,nw_dst=172.16.12.0/24 actions=CONTROLLER:65535
 cookie=0x2, duration=4400.318s, table=0, n_packets=0, n_bytes=0, priority=2,ip,nw_dst=172.16.23.0/24 actions=CONTROLLER:65535
 cookie=0x801, duration=4400.307s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=10.10.10.0/24 actions=CONTROLLER:65535
 cookie=0x801, duration=4400.298s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_dst=10.10.30.0/24 actions=CONTROLLER:65535
 cookie=0x0, duration=4538.521s, table=0, n_packets=9, n_bytes=540, priority=1,arp actions=CONTROLLER:65535
 cookie=0x24, duration=4538.521s, table=0, n_packets=0, n_bytes=0, priority=1,mpls actions=CONTROLLER:65535
 cookie=0x0, duration=4538.521s, table=0, n_packets=820, n_bytes=80360, priority=0 actions=NORMAL
[root@ovs2 ~]# █
```

Clearly, **we see that the number of flows is reduced significantly** and hence saving memory and aiding in a quicker lookup.

7.3 Demo Plan

Demo ID	Action	Expected Observation	Pass/Fail
1	Configure a tunnel from H1 to H2, followed by executing ping from H1 to H2.	Successful ping and all packets must be tunneled through to the destination with the specific Tunnel ID.	
2	Configure tunnels from H1 to H2 and H1 to H3. Source packets from H1 to H2 and H1 to H3	All packets must be tunneled to the destination. Packets from H1 to H3 will have a different Tunnel ID as compared to packets from H1 to H2.	
3	Configure tunnels from H1 to H2 and do not configure a tunnel from H1 to H3. Source packets from H1 to H2 and from H1 to H3.	All packets from H1 to H2 must be tunneled through to the destination using the specific Tunnel ID. Packets from H1 to H3 must not be tunneled through and must traverse using normal flow routing.	
4	Configure a tunnel from H1 to H2. No flow entry should be there, execute a ping from H1 to H2.	Ingress Ovs1 communicates with the controller and fetches the Tunnel ID for that flow.	
5	Configure tunnel from H1 to H2. Send packet for existing flow between H1 and H2	The ingress ovs1 forwards the packet without generating a packet-in to controller	
6	Configure tunnel from H1 to H2. Send packet from H1 to H2	Egress ovs3 strips the Tunnel ID and then sends the packet to H2. ovs3 does not send any packet to controller because it already having the flow present.	
7	Configure H2 to have 10 even IP addresses and H3 to have 10 odd IP addresses. Then source a ping from H1 to all the odd and even ip addresses simultaneously.	Successful ping and we would see that all packets transmitted to the odd ip address would have label ID 1 and the packets destined to the even IP address would have label ID 2. Also, there is an improved performance i.e. lesser delay due to lesser number of flows on the intermediate Ovs2-only 2 as well quicker lookups based on tunnel ID	

Note:

1. Wireshark output at each interface will show the respective PUSH/POP Tunnel Id operations for LER/LSR.
2. REST API GET config request command will show that which tunnel id mapped to which destination and outport.

Reference:

https://osrg.github.io/ryu/
https://www.opennetworking.org/...specifications/openflow/openflow-spec-v1.3.0.pdf
https://github.com/osrg/ryu/blob/master/ryu/lib/packet/mpls.py
http://sdnhub.org/tutorials/ryu/
https://www.scribd.com/document/321625798/sdn-book
https://events.linuxfoundation.org/images/stories/pdf/lcna_co2012_ohmura.pdf