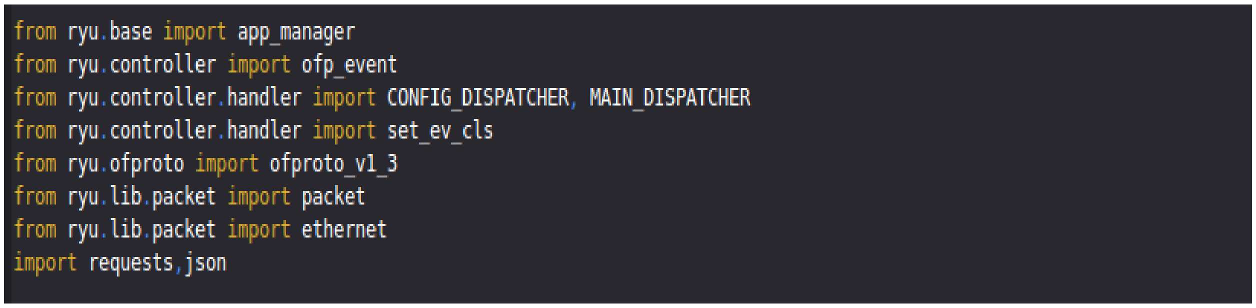
Code Dissection and Analysis:

<u>Queue Configuration and BW control using Ryu controller and OVS switches.</u>

1. <u>Required Libraries</u>

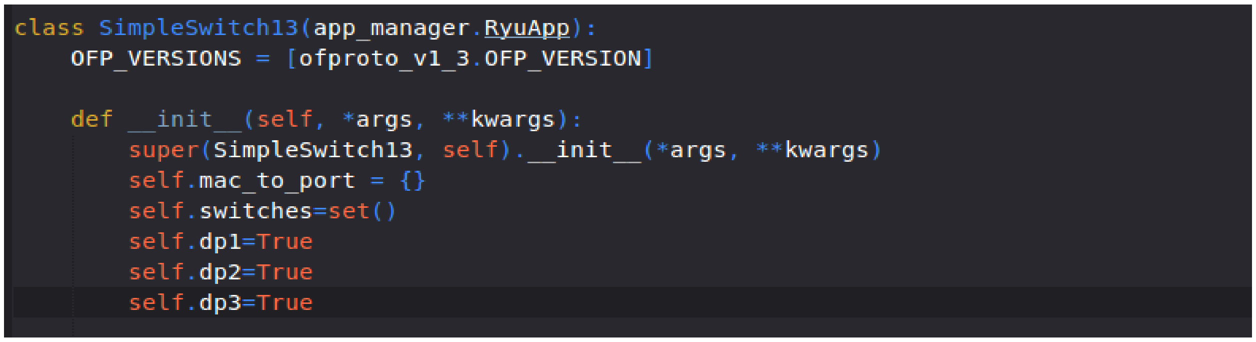
Before Starting to write the application code, it is necessary to import all the required libraries.



```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
import requests,json
```

The libraries in use consist of the following:
ryu – The Ryu package.
base – Provides base classes for Ryu Controller Applications.
app_manager – Provides the RyuApp base class used by the application.
controller – Contains modules for the base Ryu Controller.
ofp_event – OpenFlow event definitions.
handler – Event handler.
CONFIG_DISPATCHER – OpenFlow data path state.
MAIN_DISPATCHER – OpenFlow data path state.
set_ev_cls – Registers a method as an event handler for the specified class.
lib – Utility libraries.
packet – Packet parsing and serializing libraries.
ethernet – Ethernet protocol utilities.
ether_types– Defines several Ethertypes.
requests – http library written in python.
JSON – JSON data

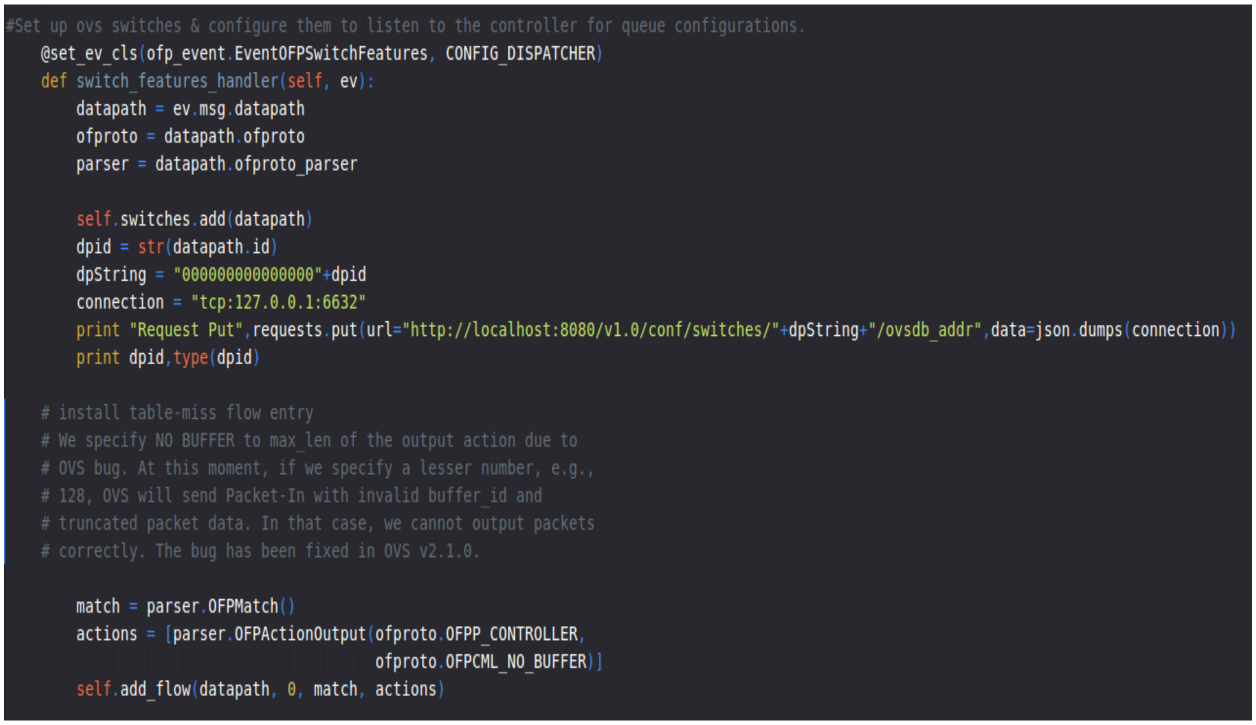
2. <u>Registration and Initialization of the Ryu controller application</u>



```python
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.switches=set()
        self.dp1=True
        self.dp2=True
        self.dp3=True
```

Here a new class is created, sub classing ryu.base.app_manager.RyuApp, which registers the application and gets instantiated by ryu-manager. Also, OpenFlow version 1.3 is specified which is used for this application.
The method __init__ will accept any number of arguments and then will pass them to the superclass. After that mac_to_port is initialized as an instance variable and is defined as a dictionary. Also, set of switches is initialized.

3. <u>Setting up of OVS switches and then configuring them to listen to the controller for the queue configuration</u>



```
#Set up ovs switches & configure them to listen to the controller for queue configurations.
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        self.switches.add(datapath)
        dpid = str(datapath.id)
        dpString = "000000000000000"+dpid
        connection = "tcp:127.0.0.1:6632"
        print "Request Put",requests.put(url="http://localhost:8080/v1.0/conf/switches/"+dpString+"/ovsdb_addr",data=json.dumps(connection))
        print dpid,type(dpid)

    # install table-miss flow entry
    # We specify NO BUFFER to max_len of the output action due to
    # OVS bug. At this moment, if we specify a lesser number, e.g.,
    # 128, OVS will send Packet-In with invalid buffer_id and
    # truncated packet data. In that case, we cannot output packets
    # correctly. The bug has been fixed in OVS v2.1.0.

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
```

Here @set_ev_cls is a decorator, which registers the attached function (as an event handler for the specified event type within the specified dispatcher. In this case the method switch_features_handler will be called whenever ryu processes an event of type EventOFPSwitchFeatures during the CONFIG_DISPATCHER that is Negotiation phase (between switch and the controller)

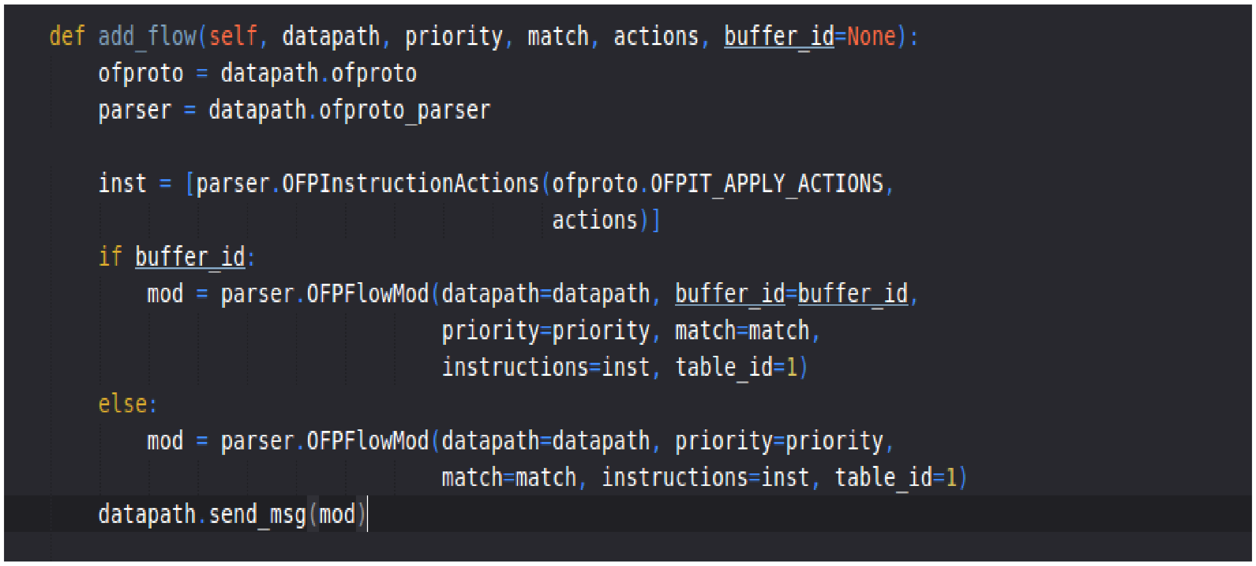
The event is passed in as ev which is an instance of ryu.controller.ofp_event.EventOFPswitchFeatures, msg is an instance of ryu.ofproto.ofproto_v1_3.OFPSwitchFeatures and datapath is an instance of ryu.controller.Datapath which represents the switch that sent the message.

switch.add(datapath) adds the switch to the set of switches

requests.put(url) is used for storing the enclosed entity under the supplied Request-URI for setting the listen port of OVSDB in the controller.

For building the flow entry, a table miss flow entry is installed first, it has the lowest priority (0) and this entry matches all the packets. In the instruction of this flow entry the output action is specified to output to the controller port if there is no match of the normal flow entry, whenever packet-in is issued. The Match is expressed in the OFPMatch class. An instance of output action class (OPPActionOutput) is generated to transfer to the controller port and OFPCML_NO_BUFFER is specified with the maximum length to send all the packets to the controller. And finally, an add_flow function with lowest priority of 0 is used to send the flow mod message.

4. <u>Add flow Helper function</u>

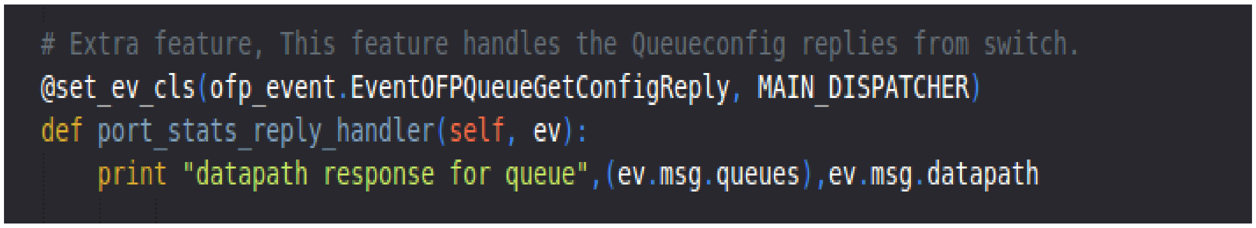


```
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    if buffer_id:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst, table_id=1)
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst, table_id=1)
    datapath.send_msg(mod)
```

Switch_feature_handler called a method add_flow which adds flow entries with predefined options and helps to reduce the controller application code elsewhere.

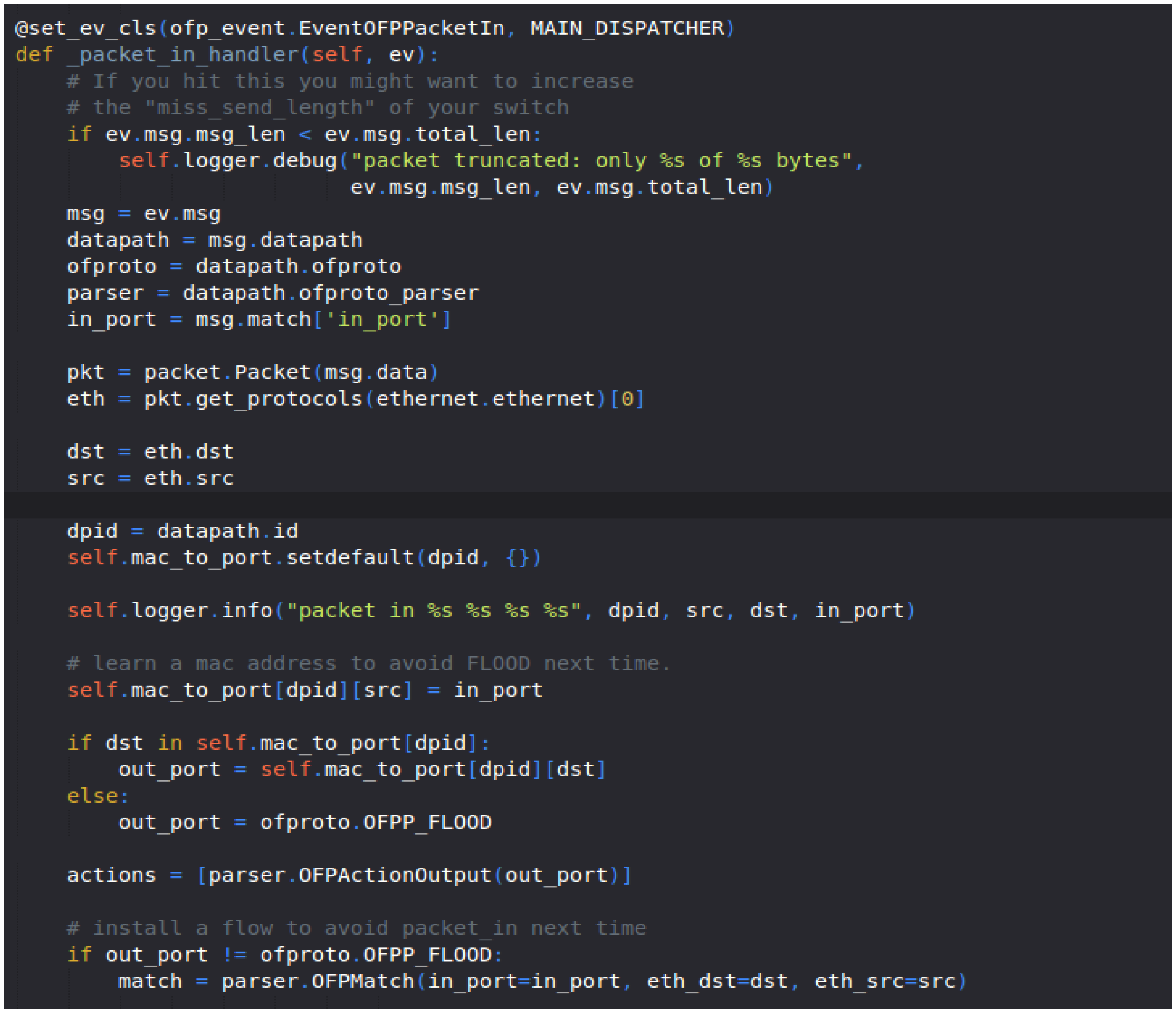
As we are working with OpenFlow version 1.3, just actions are not sufficient and thus a set of instructions must be included in flow mod messages, which is done by inst in the above method and then the actual flow mod is constructed and finally it is sent to the switch by the rye.

5. <u>Feature for handling the Queue Configuration replies from the switches</u>



```
# Extra feature, This feature handles the Queueconfig replies from switch.
@set_ev_cls(ofp_event.EventOFPQueueGetConfigReply, MAIN_DISPATCHER)
def port_stats_reply_handler(self, ev):
    print "datapath response for queue",(ev.msg.queues),ev.msg.datapath
```

6. <u>Packet In Handler</u>



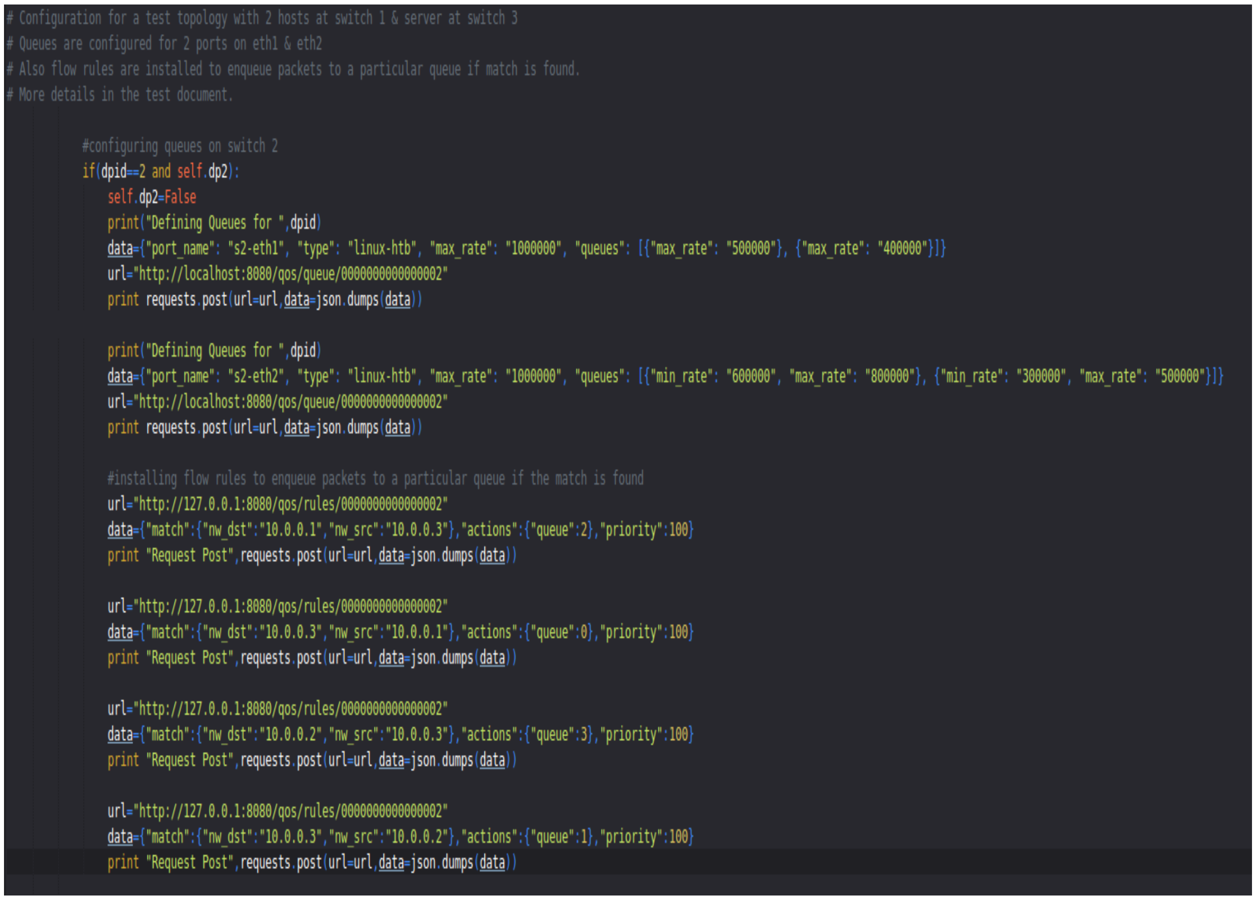
```python
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

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

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

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

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

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

Here @set_ev_cls is a decorator, which registers the attached function (as an event handler for the specified event type within the specified dispatcher. In this case the method packet_in_handler will be called whenever ryu processes an event of type EventOFPacketIn in the MAIN_DISPATCHER.

The first two lines in this method perform sanity checking for the message length. If the entire message is received or not. If the entire message is not received, then it provides the log for the truncated message.

After which important data is pulled, out of which the in_port retrieved from msg.match [in_Port'] is most important as it gives the physical port number where the packet was received on the switch that sent the packet-in message. The submitted packets are parsed and source MAC address is learnt and finally the mac_to_port[dpid][src] is updated and set to in_port, then the destination look-up is performed using the same table. Now if the destination MAC address is known then a flow entry is added to avoid packet_in next time.

7. Queue configuration and flow rules installation for the test topology



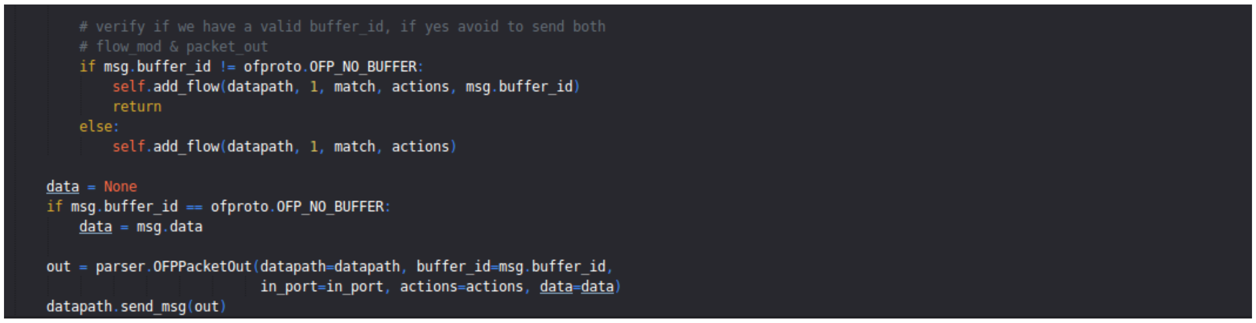
```
# Configuration for a test topology with 2 hosts at switch 1 & server at switch 3
# Queues are configured for 2 ports on eth1 & eth2
# Also flow rules are installed to enqueue packets to a particular queue if match is found.
# More details in the test document.

        #configuring queues on switch 2
        if(dpid==2 and self.dp2):
            self.dp2=False
            print("Defining Queues for ",dpid)
            data={"port_name": "s2-eth1", "type": "linux-htb", "max_rate": "1000000", "queues": [{"max_rate": "500000"}, {"max_rate": "400000"}]}
            url="http://localhost:8080/qos/queue/0000000000000002"
            print requests.post(url=url,data=json.dumps(data))

            print("Defining Queues for ",dpid)
            data={"port_name": "s2-eth2", "type": "linux-htb", "max_rate": "1000000", "queues": [{"min_rate": "600000", "max_rate": "800000"}, {"min_rate": "300000", "max_rate": "500000"}]}
            url="http://localhost:8080/qos/queue/0000000000000002"
            print requests.post(url=url,data=json.dumps(data))

            #installing flow rules to enqueue packets to a particular queue if the match is found
            url="http://127.0.0.1:8080/qos/rules/0000000000000002"
            data={"match":{"nw_dst":"10.0.0.1","nw_src":"10.0.0.3"},"actions":{"queue":2},"priority":100}
            print "Request Post",requests.post(url=url,data=json.dumps(data))

            url="http://127.0.0.1:8080/qos/rules/0000000000000002"
            data={"match":{"nw_dst":"10.0.0.3","nw_src":"10.0.0.1"},"actions":{"queue":0},"priority":100}
            print "Request Post",requests.post(url=url,data=json.dumps(data))

            url="http://127.0.0.1:8080/qos/rules/0000000000000002"
            data={"match":{"nw_dst":"10.0.0.2","nw_src":"10.0.0.3"},"actions":{"queue":3},"priority":100}
            print "Request Post",requests.post(url=url,data=json.dumps(data))

            url="http://127.0.0.1:8080/qos/rules/0000000000000002"
            data={"match":{"nw_dst":"10.0.0.3","nw_src":"10.0.0.2"},"actions":{"queue":1},"priority":100}
            print "Request Post",requests.post(url=url,data=json.dumps(data))
```

Here the Queues with min and max rate are configured on the ports s2-eth1 and s2-eth2 on switch 2 for the test topology which has three switches and three hosts, where two hosts are connected to switch 1 and the server is connected to switch 3.

Flow rules are installed to enqueue packets to a particular queue, if the match is found.

For more details about the Queue configuration, refer to rest_qos and rest_conf_switch applications provided by Ryu.

8. Forwarding the Packet received by the controller



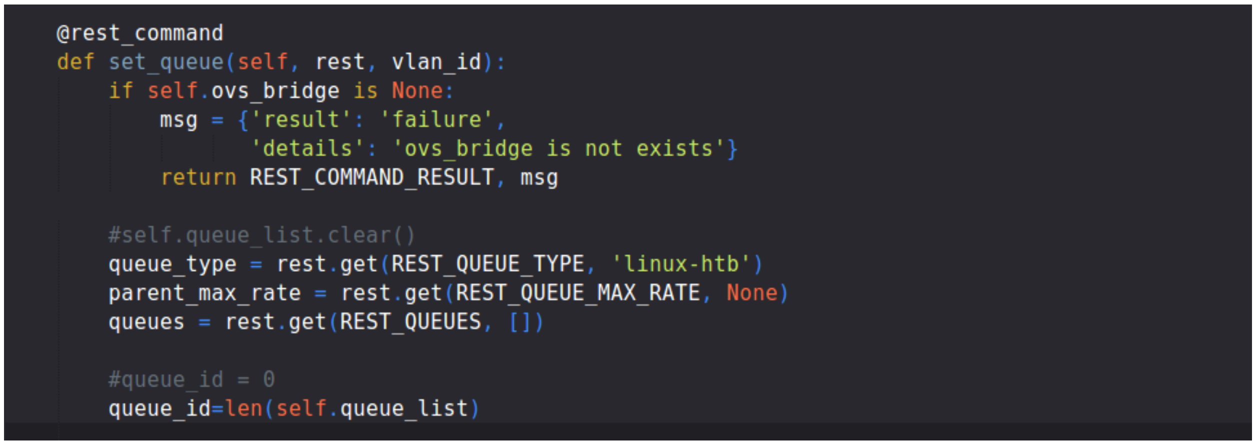
```
        # verify if we have a valid buffer_id, if yes avoid to send both
        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow(datapath, 1, match, actions)

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

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

If the switch had buffered the Packet, then packet_in message contains the buffer Id and is waiting for the controller to tell the switch about where to forward the packet, this means that the flow entry must be added while referencing to that buffer. A priority higher than the flow-miss entry is set for the new learned-path flow entry. As the packet was buffered the method returns immediately.

If the switch had not buffered the packet, the flow entry is still added but with no reference to buffer and the original message still be sent.

The data argument is prepared by setting it to None for the later parser.OFPacketOut call. If buffer is not referenced by the packet, then the switch did not buffer the packet, and so the data in a packet-in message sent to the controller must be supplied with the packet-out message. The parser.OFPacketOut is prepared with the above variables. The in_port argument is provided so that the packet won't be forwarded to that port if out_port is set to OFPP_FLOOD.

9. Changes in rest_qos



```
@rest_command
def set_queue(self, rest, vlan_id):
    if self.ovs_bridge is None:
        msg = {'result': 'failure',
               'details': 'ovs_bridge is not exists'}
        return REST_COMMAND_RESULT, msg

    #self.queue_list.clear()
    queue_type = rest.get(REST_QUEUE_TYPE, 'linux-htb')
    parent_max_rate = rest.get(REST_QUEUE_MAX_RATE, None)
    queues = rest.get(REST_QUEUES, [])

    #queue_id = 0
    queue_id=len(self.queue_list)
```

Here changes are made to avoid clearing of the queue list after new queues are set and queue_id is updated to the length of existing queues.

The code above (perf) is ran along with rest_qos and rest_conf_switch provided by ryu for achieving Queue configuration and BW control.

References:

https://osrg.github.io/ryu-book/en/Ryubook.pdf

http://ryu.readthedocs.io/en/latest/writing_ryu_app.html

https://inside-openflow.com/2016/07/21/ryu-api-dissecting-simple-switch/