Research Report on Routing Policy Static and Adaptive Routing (June 2017)

Jin Tea Kim-1485971, Mingjun Wu – 1490523, Computing Department of Unitec

1 Introduction

The OpenFlow is a standard in the software-defined networking (SDN). It initially demarcated the protocols on communication in the software-defined networking setting that allows the software-defined networking Controller to openly network with the network devices forwarding plane both virtual as well as the physical ones, so it can correctly adjust to varying corporate necessities. The SDN is an essential system component in the effort transformation. Software-defined networking splits the data and control accelerating planes. As an alternative for devising various interfacing mechanisms to different dealer networking tools, the controller is implemented to make the network interfacing possible to the plane in the data-forwarding that can deliver comprehensive network assessment and flow control at the planes in the forwarding [1].

The OpenFlow allows the deployment of the agile system. The software-defined networking controller using OpenFlow can accomplish and manage the flows in an integrated view, hence streamlining the administration of the network and allowing the updating and deployment of the agile system. The flows at the data forwarding plane are part of a flow table that can are managed by the SDN controller using OpenFlow.

The software-defined networking Controller in software-defined networking can be described as the center of the software-defined network; it is useful in the transmitting data to routers or switches through the southbound API and the submissions and corporate rationality through the northbound API. Of late, as establishments set up more software-defined networks, the software-defined networking controllers are being used to combine the software-defined networking controller domains, with the use of open virtual switch database or the OpenFlow among others [1].

The flow forwarding is used to describe the use of flow tables to configure the network paths in the forwarding information base. The flow tables are used to forward the packets and frames. In networks that use the flow forwarding, some packet fields are hashed collectively and utilized for the flow in the lookup hash table. The state data established in the hash table lookup is then used to forward packets.

In a standard switch/router, the data path (fast packet forwarding) and the control path (high-level routing decisions) take place on the same device. OpenFlow splits up these tasks. The data path functionality happens in the switch, while the control path goes on in another controller, generally a customary data server. The controller and Switch interconnect through the OpenFlow that outlines the messages, like sending out packets, receiving packets, and getting status

The data path provides a perfect abstraction in the flow table. All the entries in the flow table have matching packet fields as well as an action. After the OpenFlow gets a packet that it has at no time received, and that does not match with the existing flow entries, it directs it to the controller where the handling is decided on whether to drop the packet or log it in the network for future references [2].

The OpenFlow tables are either group tables or flow tables; the group tables are made up of the group entries, the capacity for the flow entries to refer to a group entry allows the OpenFlow to characterize further forwarding methods. All the group entries are connected to their corresponding group identifier and are made up of the group identifier, group type, the counters and action buckets. The flow tables are formed of the match fields, priority, counters, instructions, timeouts and the cookie [3].

In the OpenFlow design, the flow table contents are set by the OpenFlow Controller (OFC) on an OpenFlow Switch where the packets are processed. A combination of twelve fields is used for packet identification that is utilized by the Ethernet frames or the IP packets. OpenFlow has several limitations including, OpenFlow should be worldwide enabled before a person can enable the interface. Each interface should be explicitly disabled or enabled; a person cannot do this by using a ports range.

OpenFlow has undergone improvements in its different versions released, in the control plane separation, version 1.0 used the cookies, and version 1.4 improved to the utilization of the vacancy events, and in version 1.5 there is the use of flow learning delegation to the switch. In the northbound APIs, the 1.0 version used the per port priority queues, and in version 1.3 there was the use of per flow meters and Tunnel-id field. Other improvements include the use of Egress tables and Encapsulation action in the flow entries [2].

The RYU Controller is a software-defined networking that is open and structured to grow the network agility by the simplification of the management and adjustments of the handling of the traffic. The RYU controller delivers software constituents, with APIs that are well defined to make it simpler for designers to construct new control applications and network management. This component method helps establishments modify deployments to cater for their changing requirements. The designers and developers can swiftly and effortlessly adjust current components or apply their own to guarantee that the core network is up to date.

2 Part A: Static Routing

2.1 Naïve routing policy

By comparison the performance of the network based on different policy, we need to review the poor routing policy as below (Figure 1).

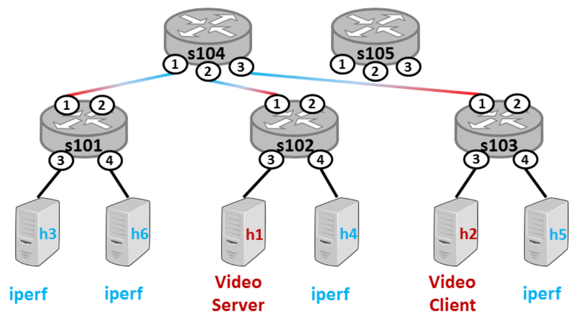


Figure 1. Naïve routing policy

In this policy, all traffic from edge switches is sent through one core switch, S104, while another core switch, S105 do nothing. Since the traffic of one tenant affects the performance of other tenant and the video application is adaptive, streaming video application can not secure enough bandwidth and will display in a bad quality.

The routing rules of the naive policy are defined as DefaultPolicy() in policy.py. In this class, only one core switch is used as below (Code 1.). In this code, topo.coreSwitches.keys()[0] indicates S104 and S105(topo.coreSwitches.keys()[1]) is not used for making routing table of both downward and upward rules.

```
# use only one core switch
core = topo.coreSwitches.keys()[0]
coreDpid = topo.coreSwitches[core].dpid
```

Code 1. Core switch used in Default policy

In the first block of source code creating the rule for traffic from core to edge, only S104(core, defined in above, Code 1.) is used to determine the outport as follows. The code (Code 2.) is executed for all of the hosts in topology (from h1 to h6), and outport of core switch is assigned by the variable core which indicates S104 and h.switch which indicate each edge switch of the host. At the end of the loop, the outport is appended to the routing table.

```
# create rules for packets from core -> edge (downward)
print("topo.host.values: ",topo.hosts.values() )
for h in topo.hosts.values():
    outport = topo.ports[core][h.switch]
    print("h.switch", h.switch, "outport", outport)
    routingTable[coreDpid].append({
        'eth_dst' : h.eth,
        'output' : [outport],
        'priority' : 2,
        'type' : 'dst'
    })
```

Code 2. Rules for downward in Default policy

The second block of code installs rules in the edge switches. An edge switch has two types of rules. Regarding the rules for upward traffics which are sent from edge switch to core switch. The default policy is making an empty routing table first for each edge switch. For each edge switch, all of the host connected to the edge switch are evaluated for creating the routing table. In case the destination host is matched with the neighbor's list, the policy does not send traffics up to core switch but sends it to host directly. However, if the host is not in the neighbor's list, the traffics are routed core switch. As mentioned before, in this policy, only one core switch(S104) is used. Below is the code for upward rules. Variable edge has a name of edge switches, S101, S102 and S103 and variable h is destination host name from each edge switches. In the source code, routing rule for all of host from h1 to h6 are evaluated to make routing table for each edge switches: S101, S102, and S103.

```
# create rules for packets from edge -> core (upward)
for edge in topo.edgeSwitches.values():
    routingTable[edge.dpid] = []
    for h in topo.hosts.values():
        # don't send edge switch's neighbors up to core
        if h.name in edge.neighbors:
            outport = topo.ports[edge.name][h.name]
        else:
            outport = topo.ports[edge.name][core]
```

Code 3. Rule for upward in Default policy

2.2 Static routing policy

As we discussed above, there are certain level of limitation in naive policy to secure enough bandwidth because only one core switch is used for routing rule. In static policy, both of the core switches are involved in routing traffics. One core switch is for handling video streaming while another core switch routing iperf traffic. In this project, the Python codes added to the policy.py file. In the StaticRouting class of this file, the extended function build() includes the code which installs on role in each edge switch. The package will be output the port which is directly connected to the host and switch. Otherwise, the packets are going to be sending “up” to the core switch. This core switch is related to the target host’s VLAN (Figure 2.).

First of all, the source code creates rules for core switches to send packets from a core switch down to an

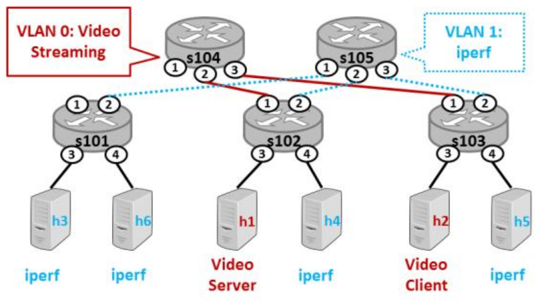


Figure 2. Static routing policy

edge switch (Code 4.). These rules define, for a given host target, core switches send the package to the host's edge switch. If the packet target matches host MAC address,

```
# core switches: for a given host destination, send packet
# to the host's edge switch ("downward")
print("CoreSwitches : ",topo.coreSwitches.values() )# S104, S105
for core in topo.coreSwitches.values():
    routingTable[core.dpid] = []
    for h in topo.hosts.values():
        outport = topo.ports[core.name][h.switch]
        print("Host: ", h, "core.name :", core.name, "h.switch : ",\
               h.switch, "Outport:", outport)
        routingTable[core.dpid].append({
            'eth_dst' : h.eth,
            'output' : [outport],
            'priority' : 2,
            'type' : 'dst'
        })
```

Code 4. Rules for downward in Static policy

then forward part of it. The port it forward is the destination host which connected to the edge switch. In the code, both core switches are used for routing rules.

The second block of code (Code 5.) installs rules in the

```
for edge in topo.edgeSwitches.values():
    print("======= Edge ======", edge)
    #print("Edeg : ", edge)
    print("Edge dpid :", edge.dpid)
    routingTable[edge.dpid] = []
    for h in topo.hosts.values():
        print
        print("****** Host ******", h)
        #print("Host : ", h)
        # don't send edge switch's neighbors up to core
        # if the destination of host is in the neighbor list,
        # ouput to appropriate port.
        # Do not send it up to a Core switch
        print("h.name :", h.name,"Edge; name : ",edge.name ,\
              "  edge.neighbors : ", edge.neighbors)
        if h.name in edge.neighbors:
            print("Found in neighbor list ")
            print("Output port : ", topo.ports[edge.name][h.name])
            outport = topo.ports[edge.name][h.name]
        else:
            # if not in the neighor list, send to the core switch
            # for the destinations's vlan
            # outport : edge switch's port number to core switch
            outport = topo.ports[edge.name][topo.getVlanCore(h.vlans[0])]
            print("Not in neighbor list, \
                   then send packet up to core switch of destination host
            print("VLAN ID for the host :", h.vlans[0])
            print (edge.name, h.name, topo.getVlanCore(h.vlans[0]),outport

        routingTable[edge.dpid].append({
            'eth_dst' : h.eth,   # MAC address of the host
            'output' : [outport],
            'priority' : 2,
            'type' : 'dst'
        })
```

Code 5. Rules for Upward in Static policy

edge switches. An edge switch has two types of rules. One is that outputting the appropriate port to the host if the host destination which connecting to the brink switch. It also means when neighboring host provided, leaving out sending to one core switch. Another need to use all switches. In the topo.port, the target port will return the core switch by using the function debt VLAN core. If the host has multiple VLANs, use the first one in the VLAN list. In the coding area, the main API is topo.ports[edge switch name][host name] and topo.getVlanCore(vlanId). The first returns the port number of edge switch and the second returns the pre-assigned vlanId of the core switch. The h.vlans[0] means the top value of stack in port list. In this topology, h.vlans[0] is S104.

In the code, if the destination host is in the neighbor list, then the packet is not send to core switch but route to appropriate port. The destination host which is not in the neighbor list will routed to core switch for the destination's vlan. Once outport is determined, it will add to routing table with MAC address of host.

Compared with the original method, the static routing function shows two improvements. In the downward part, the changing is from using single core switch to using all switches. In the upward part, the varying is separate VLAN video streaming and VLAN iperf.

2.3 Improvement of streaming video quality

In static policy, since all of traffics are distributed to both core switches, hosts of video streaming application will be allocated more network resource compare to default policy. As a result, the video quality is improved significantly. In default policy, hosts for video application, h1 and h2, only occupied very small portion of bandwidth, while in static policy, all of hosts share total bandwidth at the same level. Thus, hosts of steaming application will have more bandwidth and result in improved video qual ity (Figure 3, Figure 4).

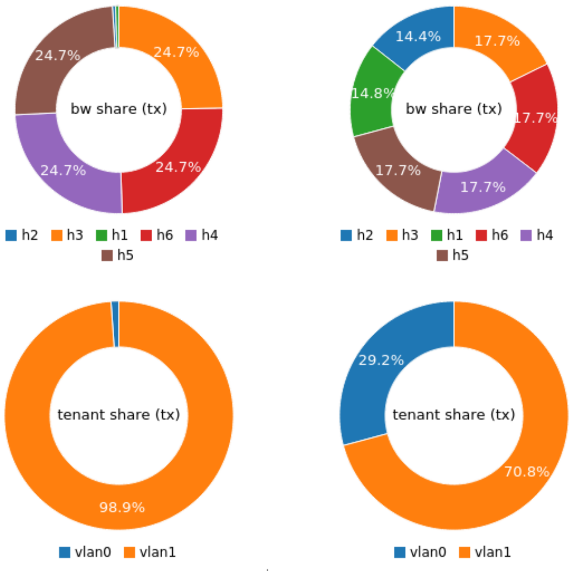


Figure 3. Bandwidth of hosts for Default and Static policy

Figure 4. Comparison of streaming video quality based on routing policies(Static(left) and Default policy)

3 Part B: Dynamic Routing

3.1 Aim of dynamic routing policy

Delivering traffic for each VLAN in core switches by using static routing policy exist some issues. The problem is that the bandwidth requirements of each tenant are not the same. It means that the distribution of load is not equal for each core switch. Therefore, one condition is that a significant amount of bandwidth use is dealt with by the single core switches rather than full utilization of each core switches.

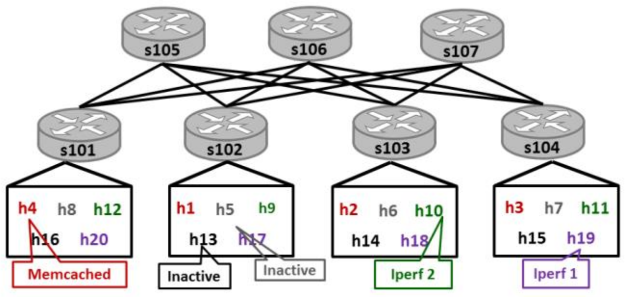
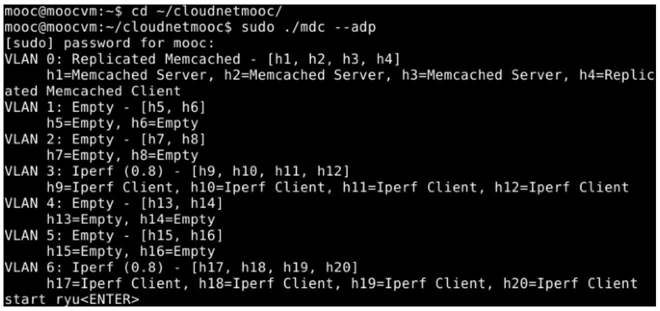


Figure 5. Topology for Adaptive policy

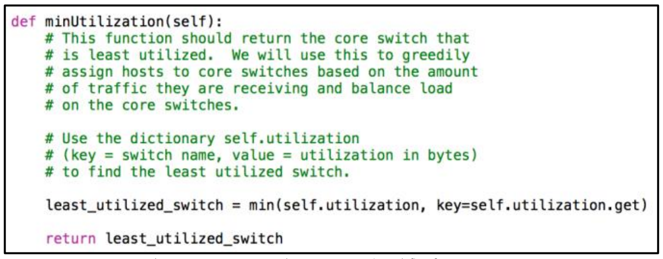
For a better assignment, a Memcached application would replace video streaming. The definition of Memcached is a distributed cash which can stores object in memory. The larger topology is used as shown in the following topology graph. The structure of this topology has three core switches and four edge switches. Each edge switch contains five hosts. The tenants are divided into three different types: Memcached, Iperf and inactive. Each type has four hosts. In the Memcached tenants, the Memcached objects are parallel retrieved from three different Memcached servers to the client. The retrieving process will fake to load multiple objects on one web page. There are two hosts in the two Iperf tenants, which will tenants simulate bulk transportation. As to the two inactive tenants, there is no data transfer between the hosts.

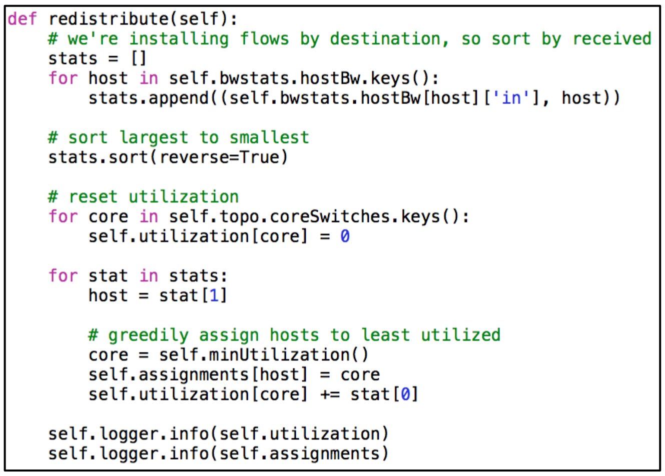
Compared to the previous static routing policy which assigns VLANs to the target core switch, the new policy would create the scheduling strategies. This adaptive routing policy would balance the traffic through the core switches rather than VLAN association. The added python code in the method fuctionminUtilization() of class AdaptivePolicy would imply the new routing policy by using self.utilization API. This code will find the target core switch which is the least utilized.


```
mooc@moocvm:~$ cd ~/cloudnetmooc/
mooc@moocvm:~/cloudnetmooc$ sudo ./mdc --adp
[sudo] password for mooc:
VLAN 0: Replicated Memcached - [h1, h2, h3, h4]
     h1=Memcached Server, h2=Memcached Server, h3=Memcached Server, h4=Replic
ated Memcached Client
VLAN 1: Empty - [h5, h6]
     h5=Empty, h6=Empty
VLAN 2: Empty - [h7, h8]
     h7=Empty, h8=Empty
VLAN 3: Iperf (0.8) - [h9, h10, h11, h12]
     h9=Iperf Client, h10=Iperf Client, h11=Iperf Client, h12=Iperf Client
VLAN 4: Empty - [h13, h14]
     h13=Empty, h14=Empty
VLAN 5: Empty - [h15, h16]
     h15=Empty, h16=Empty
VLAN 6: Iperf (0.8) - [h17, h18, h19, h20]
     h17=Iperf Client, h18=Iperf Client, h19=Iperf Client, h20=Iperf Client
start ryu<ENTER>
```
Figure 6. hosts connection in the topology

3.2 Code of dynamic routing policy

In the coding area, after initiation and rebuilding the statics of bandwidth, the central part is the minUtilization function. This function would return the switch with the lowest utilization by enumerating the full switch in the self.utlization dictionary. In the self. utilization dictionary, the switch name and its total traffic load is the key and value of it. This code sorted the entire switches by using the min(switches, key = func) method. The method would return the minimum argument when two or more argument exist. The minimum argument will put value into the defined parameter least_utilized_switch. As a result, minutilization(self) function will return the least utilized switch.


```
def minUtilization(self):
    # This function should return the core switch that
    # is least utilized.  We will use this to greedily
    # assign hosts to core switches based on the amount
    # of traffic they are receiving and balance load
    # on the core switches.

    # Use the dictionary self.utilization
    # (key = switch name, value = utilization in bytes)
    # to find the least utilized switch.

    least_utilized_switch = min(self.utilization, key=self.utilization.get)

    return least_utilized_switch
```
Code 6. minUtilizaiton(self) function

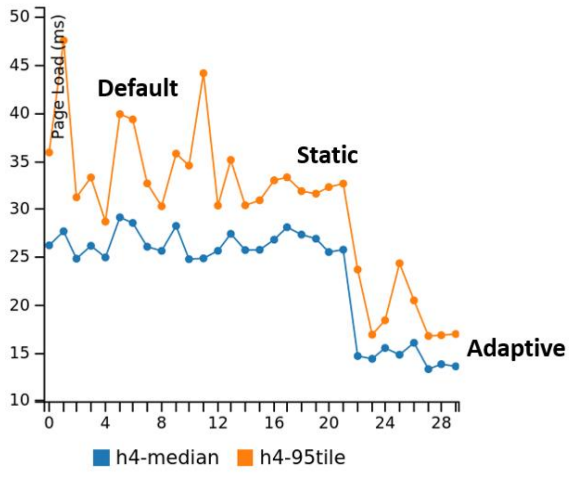

```
def redistribute(self):
    # we're installing flows by destination, so sort by received
    stats = []
    for host in self.bwstats.hostBw.keys():
        stats.append((self.bwstats.hostBw[host]['in'], host))

    # sort largest to smallest
    stats.sort(reverse=True)

    # reset utilization
    for core in self.topo.coreSwitches.keys():
        self.utilization[core] = 0

    for stat in stats:
        host = stat[1]

        # greedily assign hosts to least utilized
        core = self.minUtilization()
        self.assignments[host] = core
        self.utilization[core] += stat[0]

    self.logger.info(self.utilization)
    self.logger.info(self.assignments)
```
Code 7. redistribute(self) function

In the next section, the redistribute function starts to creating stats list. Then, it adds the tuple to the stats list after enumerating whole host and monitors of bandwidth. This tuple includes the host name and its transmitted bytes. Moveover, then the list is sorted from largest to smallest. It uses the tuple and usage of the host as the first two elements. Next, all core utilizations are resetting. Last but not least, the function saves the current argument in the variable stat after enumerating each tuple.

When it comes to the following code area, the variable hostname is stored and the second element of host tuple are remembered. Then, it will find the least utilized switch by invoking the minUtilizaion function. After that, to establish the forwarding rules, the host assignment are saved. Finally, it updates the core utilization which with the host and its usage.

3.1 Improvement of page loading time

The picture below indicates the improvement of page loading time among default, static and adaptive policy.



At first, the median response time of the system has about 22 milliseconds. The time of reply of 95th percentile is average 30. When choosing the static policy, it indicates the little change both median and 95th percentile.

After a several points load, by selecting the Adaptive button and pressing Update, it shows that a significant improvement not only in the median but also 95th percentile. The response times of both two has dropped to around 13 milliseconds.

7 Conclusion

In conclusion, the development of virtualization has transformed the Information Technology environment, making features that are on-demand and more scalability available everywhere. Nevertheless, the network world has to match this malleable domain of Cloud computing; there is a huge need to do network management that is considered inert, contains additional constraints and necessitate manual mediations. The primary subject is that common methods in networking may lead to a rather slow company infrastructure delaying the deployments. The business must come up with new approaches that are accessible and deliver automated tactic to the management of the networks [3].

The use of the OpenFlow protocol in Network Virtualization provides the organizations with high-level flexibility, leading to ease in the management of the current networks and can be utilized by both users and network administrators. It also makes available the user based network configurations so that it can meet the company's special requirements. Once the OpenFlow technology is set up, it is tough to uphold the Network Virtualization integrity based on those requirements. Therefore, it is crucial for the OpenFlow controller to have the information of every in the network.

This OpenFlow design is made up of three initial perceptions that are: The data plane is composed of OpenFlow-compliant switches that build up the network, the control plane is made up of the OpenFlow controllers, and a secure control channel connects the switches with the control plane. The OpenFlow allows the company to set up first effortlessly and advanced switching and routing protocols in the network [3].

Switches based on OpenFlow concept are the rudimentary devices used for the purpose of packet forwarding by the flow table. The flow table has flow table entries that are made up of matching fields, instructions, and counters. The flow tables' entries have "header fields" that label the packets that match with their particular entry. They are made up of a wildcard that has the capability to match the packets with their corresponding header fields. The ternary content addressable memory is used in the OpenFlow-based switches to help in the fast forwarding of the packets; it assists the switch in fast search for matches in the wildcard.

References

[1] McKeown, N. (2009). Software-defined networking. INFOCOM keynote talk,17(2), 30-32.

[2] Tootoonchian, A., & Ganjali, Y. (2010, April). Hyperflow: A distributed control plane for openflow. In Proceedings of the 2010 internet network management conference on Research on enterprise networking (pp. 3-3).

[3] Kim, H., & Feamster, N. (2013). Improving network management with software defined networking. IEEE Communications Magazine, 51(2), 114-119.

Appendix A: Code for Static policy

```
class StaticPolicy(object):
    def __init__(self, topo):
        self.routingTable = self.build(topo)


    def build(self, topo):
```

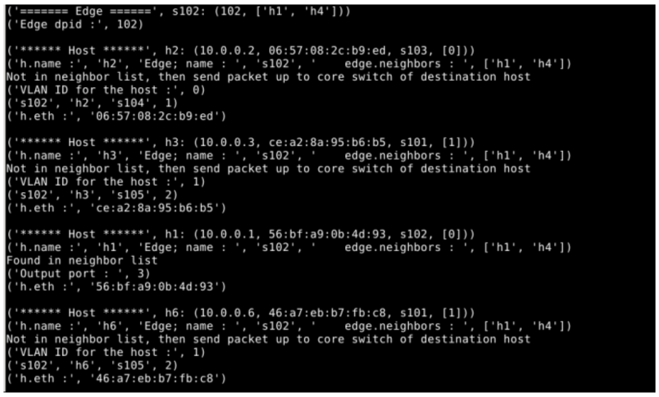
```
    routingTable = {}

    # core switches: for a given host destination, send packet
    # to the host's edge switch ("downward")
    print("CoreSwitches : ",topo.coreSwitches.values() )# S104, S105
    for core in topo.coreSwitches.values():
      routingTable[core.dpid] = []
      for h in topo.hosts.values():
        outport = topo.ports[core.name][h.switch]
        print("Host: ", h, "core.name :", core.name, "h.switch : ",\
              h.switch, "Outport:", outport)
        routingTable[core.dpid].append({
          'eth_dst' : h.eth,
          'output' : [outport],
          'priority' : 2,
          'type' : 'dst'
        })
        # Rules to Install:
    #   On the Edge Switches: output the appropriate port if the destination
    #   is a neighboring host (that is, don't send it up to a core switch).
    #   Otherwise, send to the core switch for that destination's vlan
    #   ("upward"). If a host has multiple VLANs, you can use the first
    #   in the list of VLANs.
    #   (Hint: you can look up the port of a neighboring host using
    #          topo.ports[edge switch name][host name]
    #   (Hint: to find a the VLAN, use topo.getVlanCore(vlanId))

    # create rules for packets from edge -> core (upward)
    print("topo.hosts.values():  ", topo.hosts.values())

        print("===================================
        ==========================")

    for edge in topo.edgeSwitches.values():
      print("======= Edge ======", edge)
      #print("Edeg : ", edge)
      print("Edge dpid :", edge.dpid)
      routingTable[edge.dpid] = []

      for h in topo.hosts.values():
        print
        print("****** Host ******", h)
        #print("Host : ", h)
        # if the destination of host is in the neighbor list, ouput to appro-
priate port.
        # Do not send it up to a Core switch
        print("h.name :", h.name,"Edge; name : ",edge.name , "    \
                    edge.neighbors : ", edge.neighbors)

        if h.name in edge.neighbors:
          print("Found in neighbor list ")
          print("Output port : ", topo.ports[edge.name][h.name])
          outport = topo.ports[edge.name][h.name]
        else:
          # if not in the neighor list, send to the core switch for the desti-
nations's vlan
          outport                                                          =
topo.ports[edge.name][topo.getVlanCore(h.vlans[0])]
          print("Not in neighbor list, then send packet up to core switch
of destination host ")
          print("VLAN ID for the host :", h.vlans[0])
          print            (edge.name,             h.name,
topo.getVlanCore(h.vlans[0]),outport)

          routingTable[edge.dpid].append({
            'eth_dst' : h.eth,   # MAC address of the host
            'output' : [outport],
            'priority' : 2,
            'type' : 'dst'
          })
          print("h.eth :",h.eth)

      RETURN FLOOD.ADD_ARPFLOOD(ROUTINGTABLE, TOPO)
```

Appendix B: Code for Adaptive policy

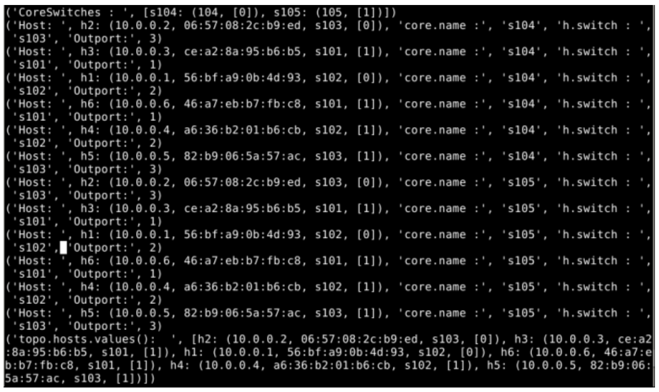
```
class AdaptivePolicy(object):
  def __init__(self, topo, bwstats, logger):
    self.topo = topo
    self.bwstats = bwstats
    self.logger = logger
    self.assignments = {}
    self.utilization = {}
    default = topo.coreSwitches.keys()[0]
    for host in topo.hosts.keys():
      self.assignments[host] = default
    self._routingTable = self.build(self.topo)

  @property
        def routingTable(self):
                self.redistribute()
    return self.build(self.topo)

  def minUtilization(self):
least_utilized_switch = min(self.utilization, key=self.utilization.get)
                          return least_utilized_switch

  def redistribute(self):
                    stats = []
    for host in self.bwstats.hostBw.keys():
      stats.append((self.bwstats.hostBw[host]['in'], host))

                    stats.sort(reverse=True)

                    for core in self.topo.coreSwitches.keys():
      self.utilization[core] = 0

    for stat in stats:
      host = stat[1]

                          core = self.minUtilization()
      self.assignments[host] = core
      self.utilization[core] += stat[0]

    self.logger.info(self.utilization)
    self.logger.info(self.assignments)

  def build(self, topo):
    routingTable = {}
```

```
                                    for core in topo.coreSwitches.values():
        routingTable[core.dpid] = []
        for h in topo.hosts.values():
            outport = topo.ports[core.name][h.switch]
            routingTable[core.dpid].append({
                'eth_dst' : h.eth,
                'output' : [outport],
                'priority' : 2,
                'type' : 'dst'
            })
    for edge in topo.edgeSwitches.values():
        routingTable[edge.dpid] = []
        for h in topo.hosts.values():
                                              if h.name in edge.neighbors:
                outport = topo.ports[edge.name][h.name]
            else:
               core = self.assignments[h.name]
                outport = topo.ports[edge.name][core]

            routingTable[edge.dpid].append({
                'eth_dst' : h.eth,
                'output' : [outport],
                'priority' : 2,
                'type' : 'dst'
            })

    return flood.add_arpflood(routingTable, topo)
```

APPENDIX C: LOG OF STATIC PLOICY (UPWARED AND DOWNWORD)



```
('======= Edge ======', s102: (102, ['h1', 'h4']))
('Edge dpid :', 102)

('****** Host ******', h2: (10.0.0.2, 06:57:08:2c:b9:ed, s103, [0]))
('h.name :', 'h2', 'Edge; name : ', 's102', '    edge.neighbors : ', ['h1', 'h4'])
Not in neighbor list, then send packet up to core switch of destination host
('VLAN ID for the host :', 0)
('s102', 'h2', 's104', 1)
('h.eth :', '06:57:08:2c:b9:ed')

('****** Host ******', h3: (10.0.0.3, ce:a2:8a:95:b6:b5, s101, [1]))
('h.name :', 'h3', 'Edge; name : ', 's102', '    edge.neighbors : ', ['h1', 'h4'])
Not in neighbor list, then send packet up to core switch of destination host
('VLAN ID for the host :', 1)
('s102', 'h3', 's105', 2)
('h.eth :', 'ce:a2:8a:95:b6:b5')

('****** Host ******', h1: (10.0.0.1, 56:bf:a9:0b:4d:93, s102, [0]))
('h.name :', 'h1', 'Edge; name : ', 's102', '    edge.neighbors : ', ['h1', 'h4'])
Found in neighbor list
('Output port : ', 3)
('h.eth :', '56:bf:a9:0b:4d:93')

('****** Host ******', h6: (10.0.0.6, 46:a7:eb:b7:fb:c8, s101, [1]))
('h.name :', 'h6', 'Edge; name : ', 's102', '    edge.neighbors : ', ['h1', 'h4'])
Not in neighbor list, then send packet up to core switch of destination host
('VLAN ID for the host :', 1)
('s102', 'h6', 's105', 2)
('h.eth :', '46:a7:eb:b7:fb:c8')
```



```
('CoreSwitches : ', [s104: (104, [0]), s105: (105, [1])])
('Host: ', h2: (10.0.0.2, 06:57:08:2c:b9:ed, s103, [0]), 'core.name :', 's104', 'h.switch : ',
 's103', 'Outport:', 3)
('Host: ', h3: (10.0.0.3, ce:a2:8a:95:b6:b5, s101, [1]), 'core.name :', 's104', 'h.switch : ',
 's101', 'Outport:', 1)
('Host: ', h1: (10.0.0.1, 56:bf:a9:0b:4d:93, s102, [0]), 'core.name :', 's104', 'h.switch : ',
 's102', 'Outport:', 2)
('Host: ', h6: (10.0.0.6, 46:a7:eb:b7:fb:c8, s101, [1]), 'core.name :', 's104', 'h.switch : ',
 's101', 'Outport:', 1)
('Host: ', h4: (10.0.0.4, a6:36:b2:01:b6:cb, s102, [1]), 'core.name :', 's104', 'h.switch : ',
 's102', 'Outport:', 2)
('Host: ', h5: (10.0.0.5, 82:b9:06:5a:57:ac, s103, [1]), 'core.name :', 's104', 'h.switch : ',
 's103', 'Outport:', 3)
('Host: ', h2: (10.0.0.2, 06:57:08:2c:b9:ed, s103, [0]), 'core.name :', 's105', 'h.switch : ',
 's103', 'Outport:', 3)
('Host: ', h3: (10.0.0.3, ce:a2:8a:95:b6:b5, s101, [1]), 'core.name :', 's105', 'h.switch : ',
 's101', 'Outport:', 1)
('Host: ', h1: (10.0.0.1, 56:bf:a9:0b:4d:93, s102, [0]), 'core.name :', 's105', 'h.switch : ',
 's102', 'Outport:', 2)
('Host: ', h6: (10.0.0.6, 46:a7:eb:b7:fb:c8, s101, [1]), 'core.name :', 's105', 'h.switch : ',
 's101', 'Outport:', 1)
('Host: ', h4: (10.0.0.4, a6:36:b2:01:b6:cb, s102, [1]), 'core.name :', 's105', 'h.switch : ',
 's102', 'Outport:', 2)
('Host: ', h5: (10.0.0.5, 82:b9:06:5a:57:ac, s103, [1]), 'core.name :', 's105', 'h.switch : ',
 's103', 'Outport:', 3)
('topo.hosts.values():  ', [h2: (10.0.0.2, 06:57:08:2c:b9:ed, s103, [0]), h3: (10.0.0.3, ce:a2
:8a:95:b6:b5, s101, [1]), h1: (10.0.0.1, 56:bf:a9:0b:4d:93, s102, [0]), h6: (10.0.0.6, 46:a7:e
b:b7:fb:c8, s101, [1]), h4: (10.0.0.4, a6:36:b2:01:b6:cb, s102, [1]), h5: (10.0.0.5, 82:b9:06:
5a:57:ac, s103, [1])])
```