# Author : Mrinal Aich

# Mail-id : cs16mtech11009@iith.ac.in, mr.mrinal.aich@gmail.com

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from operator import attrgetter

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, ether\_types

#, custom\_payload

from ryu.lib.packet import ether\_types

from ryu.topology.api import get\_switch, get\_link, get\_host, get\_all\_switch

from ryu.controller import mac\_to\_port

from ryu.topology import event, switches

from collections import defaultdict

from collections import deque

from sets import Set

import socket, select

from threading import Thread, Lock

from ryu.lib import hub

import Queue

from inspect import currentframe

import pdb, time, random, json, copy

from matplotlib import pyplot as plt

from matplotlib import animation

from ryu.ofproto import ether

from ryu.lib.packet import arp, packet

from ryu.lib.packet import ipv4, icmp, tcp,arp

from ryu.ofproto import ether

from ryu.ofproto import inet

from collections import namedtuple

#### GLOBAL MACROS

# Socket-level Constants

RECV\_BUFFER = 4096

OPERATOR\_API\_SOCK\_PORT = 12345

IP\_ADDR = "0.0.0.0"

CPU\_STATS\_SERVER\_PORT = 4444

UNIT\_OF\_BANDWIDTH = 1000 \* 1000 # Mega

# Normalization Constants

alpha = 0.85

beta = 0.0

gamma = 0.15

# Output Filenames

THROUGHPUT\_ANIM\_FILENAME = "throughput.json"

LATENCY\_ANIM\_FILENAME = "latency.json"

CPU\_ANIM\_FILENAME = "cpu.json"

LINK\_LATENCY\_ETH\_TYPE = 0x07c3

SWITCH\_LATENCY\_ETH\_TYPE = 0x07c4

# TODO Hypervisor CPU Memory # In MB

ALL\_HYPERVISOR\_CPU\_MEMORY\_INIT = 100

LINK\_BW\_LIMIT = 10000 # 100 Kbps

LINK\_BW\_THRESHOLD = 0.98

# Timeout till the SLA agrrement starts coming

SLA\_INPUT\_TIMEOUT = 12

# Total VNFs installed

gUsedVNFIds = {}

gUsedVNFCount = 0

# Docker SubNet

DOCKER\_SUB\_NET = "192.168.111."

DOCKER\_SUBNET\_NAME = 'overnet'

# Configuration File

SLA\_CONFIG\_FILE = "sla\_input.txt"

HYPERVISOR\_CONFIGURATION\_FILE = "hyp\_config.txt"

# Miscellaneous

SUCCESS = 1

FAILURE = 0

LOGGING\_DEBUG = False

LOGGING\_INFO = 0

COLOR\_WHITE = 0

COLOR\_BLACK = 1

#### GLOBAL VARIABLES

# Prioirty Level

START\_PRIORITY\_VAL = 65500

# Cloud DPID

CLOUD\_DPID = 5

# Hack for Bottleneck Detection

gNodeBottleneckHackCount = True

#### Structure of a SLA aggrement

class struct\_SLA(object):

def \_\_init\_\_(self, identifier, vnfInputList, vnfCPUMemReq, delayTolerated, reqBW, endUsersMac, endPointsDpid):

self.identifier = identifier

self.isInstalled = False # Whether SLA is installed

self.vnfInputList = vnfInputList # VNF Types

self.vnfCPUMemReq = vnfCPUMemReq # CPU Mem required by VNF

self.delayTolerated = delayTolerated # ms

self.reqBW = reqBW # Mbps

self.endUsersMac = endUsersMac # End-Users

self.endPointsDpid = endPointsDpid # End-Points DPID

self.delayBuffer = -1 # Delay-buffer

self.compPathOfSLA = {} # Complete Path of SLA

self.VNFsNetworkMac = []

self.VNFsNetworkDpid = []

self.VNFIds = []

self.centre = -1

self.pathToCentre = {}

self.pathOfServiceChain = [] # Used during Placement only

self.vlanCommonTag = ""

#### Structure of an installed VNF

class struct\_Installed\_VNF(object):

def \_\_init\_\_(self, iden, slaIdentifier, vnfType, ipAddr, macAddr, memReq, hypervisor\_dpid, servChainIndex):

self.identifier = iden

self.slaIdentifier = slaIdentifier # SLA to which it belongs

self.ipAddr = ipAddr # IP Address

self.macAddr = macAddr # MAC Address

self.vnfType = vnfType # Type of VNF

self.cpuMemReq = memReq # CPU Mem Requirement

self.dpid = hypervisor\_dpid # Hypervisor's dpid

self.servChainIndex = servChainIndex # Index in Service Chain

class Orchestrator(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(Orchestrator, self).\_\_init\_\_(\*args, \*\*kwargs)

global gUsedVNFIds

# Monitor

self.DM\_createMonitorThreads()

self.mac\_to\_port = {}

# Orchestrator

self.orchestratorMsgQueue = ""

self.DO\_createOrchestratorThread()

# API

self.DAPI\_adminThread()

# Graph data structure

self.m\_graph = {}

self.m\_graph.setdefault('edges', {})

self.m\_graph.setdefault('switches', set())

self.m\_graph.setdefault('hosts', {}) # Hosts connected to the Switch

# Monitor

self.m\_switchFlowStats = {}

self.m\_switchLatencyStats = {}

self.m\_linkLatencyStats = {}

self.m\_hypervisorMemStats = {}

self.m\_hypervisorCpuUtilStats = {}

self.m\_spTreeLinks = {}

self.m\_mac\_to\_dpid\_port = {}

self.m\_dpid\_to\_mac\_port = {}

self.m\_vnfOperationsOnStart = {}

self.m\_SLACloudOperationsOnStart = {}

# Hypervisor VNF status

self.m\_hypVNFStatus = {}

self.m\_topology\_api\_app = self

self.debugFlag = False

self.m\_SLAsCount = 0

self.m\_SLAs = {} # Maintains all the SLAs

self.m\_end\_users\_to\_SLA = {} # Reverse Map

self.m\_ovs\_mac = {} # Hardcoded MAC address for OVS based on dpid

# Communication with the Hypervisor System

self.m\_server\_socket = ""

self.m\_dpid\_to\_hyp\_ip = self.read\_HYP\_config\_file() # Map of dpid to IP Address of Hypervisor

self.m\_hypervisor\_socket\_list = []

self.m\_hyp\_ip\_sockfd\_pair = [] # Map Hypervisor IP to sockfds

self.m\_socket\_mutex = Lock() # TODO :Mutex lock for Socket DS's

# Init unused VNF Ids

for index in range(1, 255):

gUsedVNFIds[index] = False

# Monitor Thread

def \_monitorLink\_SwitchStatsThread(self):

while True:

# Flow Stats from Switch

for dp in self.m\_graph['switches']:

self.DM\_request\_stats(dp)

hub.sleep(1)

# Switch Latency

for dpid in self.m\_graph['switches']:

datapath = self.getDatapath(dpid.id)

# Temporary Mac addresses, Dst Mac Address contains identifier

switchFlowStatsSrcMac = datapath.ports[datapath.ofproto.OFPP\_LOCAL].hw\_addr

switchFlowStatsDstMac = "00:00:00:00:" + format(dpid.id,'x') + str(':') + format(dpid.id,'x')

self.sendPacket(SWITCH\_LATENCY\_ETH\_TYPE, dpid.id, datapath.ofproto.OFPP\_CONTROLLER, switchFlowStatsSrcMac, switchFlowStatsDstMac, dpid, 0)

#pdb.set\_trace()

# Link Latency

for dpid in self.m\_graph['edges']:

# Retrieve Nbrs of the current Dpid

for nbr\_dpid,out\_port in self.m\_graph['edges'][dpid].items():

# Initialize Link monitoring

if dpid > nbr\_dpid:

if dpid not in self.m\_linkLatencyStats:

self.m\_linkLatencyStats[dpid] = {}

if nbr\_dpid not in self.m\_linkLatencyStats[dpid]:

self.m\_linkLatencyStats[dpid][nbr\_dpid] = {}

self.m\_linkLatencyStats[dpid][nbr\_dpid]['lastSentTime'] = float(0.0)

self.m\_linkLatencyStats[dpid][nbr\_dpid]['lastUpdateTime'] = float(0.0)

self.m\_linkLatencyStats[dpid][nbr\_dpid]['data'] = float(0.0)

datapath = self.getDatapath(dpid)

# format(number , 'x') converts number to hex removing 'x' here switch number to hex

linkFlowStatsDstMac = "00:00:00:11:" + format(dpid,'x') + str(':') + format(nbr\_dpid,'x')

linkFlowStatsSrcMac = datapath.ports[datapath.ofproto.OFPP\_LOCAL].hw\_addr

self.sendPacket(LINK\_LATENCY\_ETH\_TYPE, dpid, out\_port, linkFlowStatsSrcMac, linkFlowStatsDstMac, dpid, nbr\_dpid)

hub.sleep(1)

# Function sends custom packet for Link monitoring

# Ref: Monitoring Latency with Openflow

def sendPacket(self, eth\_type, current\_dpid, out\_port, temp\_src\_mac, temp\_dst\_mac, src\_dpid, dst\_dpid):

# Create a custom packet

ethernet\_type = eth\_type

ethernet\_header = ethernet.ethernet(ethertype=ethernet\_type, src=temp\_src\_mac, dst=temp\_dst\_mac)

ip\_header = ipv4.ipv4(total\_length=len(ipv4.ipv4()), proto=inet.IPPROTO\_ICMP, ttl=1, src='192.111.111.111', dst='192.222.222.222')

custom\_packet = packet.Packet()

custom\_packet.add\_protocol(ethernet\_header)

custom\_packet.add\_protocol(ip\_header)

custom\_packet.serialize()

datapath = self.getDatapath(current\_dpid)

actions = [datapath.ofproto\_parser.OFPActionOutput(port=out\_port)]

msg = datapath.ofproto\_parser.OFPPacketOut(datapath=datapath, buffer\_id=0xffffffff, in\_port=datapath.ofproto.OFPP\_CONTROLLER, actions=actions, data=custom\_packet.data)

currentTime = time.time() \* 1000 # In Milliseconds

# Maintain Timestamp

if eth\_type == SWITCH\_LATENCY\_ETH\_TYPE:

# Initialize, used for Graphs

if 'startTime' not in self.m\_switchLatencyStats[current\_dpid]:

self.m\_switchLatencyStats[current\_dpid]['startTime'] = currentTime

self.m\_switchLatencyStats[current\_dpid]['lastSentTime'] = currentTime

elif eth\_type == LINK\_LATENCY\_ETH\_TYPE:

# Initialize, used for Graphs

if 'startTime' not in self.m\_linkLatencyStats[src\_dpid][dst\_dpid]:

self.m\_linkLatencyStats[src\_dpid][dst\_dpid]['startTime'] = currentTime

self.m\_linkLatencyStats[src\_dpid][dst\_dpid]['lastSentTime'] = currentTime

datapath.send\_msg(msg)

# Orchestrator Thread

def \_orchestratorThread(self):

self.orchestratorMsgQueue = Queue.Queue()

# Socket server create

self.DO\_openSocketThreadAPI()

while(1):

# Check Message Queue

self.DO\_checkMessageQueue()

hub.sleep(0.1)

# ------------------------------------------------------------------------------

# Utility Functions

# ------------------------------------------------------------------------------

# Creates the Orchestrator Thread

def DO\_createOrchestratorThread(self):

try:

# Create the Orchestrator Thread

self.socket\_thread = hub.spawn(self.\_orchestratorThread)

except:

LOG\_DEBUG("Error: unable to start Orchestrator thread")

# Check for messages in Message Queue

def DO\_checkMessageQueue(self):

if not self.orchestratorMsgQueue.empty():

message = self.orchestratorMsgQueue.get()

if message['type'] == "FlowManager":

self.\_DO\_handleFlowManagerMessage(message)

elif message['type'] == "Operator":

self.\_DO\_handleOperatorMessage(message['message'], message['socket'])

elif message['type'] == "Detector":

self.\_DO\_handleDetectorMessage(message)

# -------------------------------------------------------------------------------------------------------------------------

# ADMIN API MODULE : Temporary TODO To be removed.

# -------------------------------------------------------------------------------------------------------------------------

# Creates the Admin Thread for periodic SLA input

def DAPI\_adminThread(self):

self.DAPI\_slaInputThread = hub.spawn(self.\_slaInputThread)

# Reads SLAs to be installed over the network

def readSLAList(self):

pdb.set\_trace()

skipFirstLine = True

chars\_to\_remove = ['[', ']', '\'']

temp\_chars\_to\_remove = ['[', ']']

with open(SLA\_CONFIG\_FILE,'rb') as file:

for line in file:

# Skip First Line, contains Format

if skipFirstLine:

skipFirstLine = False

line.rstrip()

continue

# Empty Line, interpret as EOF

elif line == "\n" or line == "\r\n" or line == "\r":

return

data = line.rstrip()

data = data.split(',')

# VNFTypes

vnfTypes = data[0].strip(' ')

vnfTypes = vnfTypes.translate(None, ''.join(chars\_to\_remove)).split(' ')

# CPU Mem requiremtn of VNFs

vnfsMemReq = data[1].strip(' ')

vnfsMemReq = vnfsMemReq.translate(None, ''.join(temp\_chars\_to\_remove)).split(' ')

for index in range(0, len(vnfsMemReq)):

vnfsMemReq[index] = int(vnfsMemReq[index])

# Delay Tolerated

delayTolerated = int(data[2].strip(' '))

# Minimum Bandwidth Required

reqBW = int(data[3].strip(' '))

# Retreive Hosts MAC Addresses

macAddrs = data[4].strip(' ')

macAddrs = macAddrs.translate(None, ''.join(chars\_to\_remove)).split(' ')

# Retreieve Hosts Dpid

dpids = data[5].strip(' ')

dpids = dpids.translate(None, ''.join(chars\_to\_remove)).split(' ')

for index in range(0, len(dpids)):

dpids[index] = int(dpids[index])

if len(macAddrs) != len(dpids):

print "Error: Incorrect SLA, Count of MAC Addr and their Dpids do not match. Ignoring the SLA..."

continue

self.m\_SLAsCount = self.m\_SLAsCount + 1

slaObject = struct\_SLA(self.m\_SLAsCount, vnfTypes, vnfsMemReq, delayTolerated, reqBW, macAddrs, dpids)

self.m\_SLAs[self.m\_SLAsCount] = slaObject

# SLA Input Thread

def \_slaInputThread(self):

hub.sleep(SLA\_INPUT\_TIMEOUT)

# Reads SLAs to be installed

if False:

self.readSLAList()

for key,slaObject in self.m\_SLAs.items():

# Creates reverse Map of EndPoints to SLAs

self.mapEndUsersToSLAs(slaObject)

# Algorithm - 1 : Placement of the SLA

self.placementOfSLA(slaObject)

else:

# Hardcoded SLA agreement

# Format = struct\_SLA(# ID, [<List of VNFs>], [CPU], DelayTolerated, reqBandwidth, endUsersMacAddr, endUsersConnDpid)

#slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:21'], [1,2])

#slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:21', '00:00:00:00:00:31'], [2,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:31'], [1,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:41'], [1,4])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox', 'Middlebox'], [10, 10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:32'], [1,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:31'], [1,3])

# SLACase

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:32'], [1,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox', 'Middlebox'], [10, 10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:31'], [1,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:31'], [1,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox', 'Middlebox'], [10, 10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:42'], [1,4])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox', 'Middlebox', 'Middlebox'], [10, 10, 10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:21'], [1,2])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox'], [10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:31'], [1,3])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox', 'Middlebox'], [10, 10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:21'], [1,2])

slaObject = struct\_SLA(self.m\_SLAsCount, ['Middlebox', 'Middlebox'], [10, 10], 10, 100, ['00:00:00:00:00:11', '00:00:00:00:00:41'], [1,4])

self.m\_SLAs[slaObject.identifier] = slaObject

# -------------------------------------------------------------------------------------------------------------------------

# MONITORING MODULE :

# Reference simple\_monitor13.py

# -------------------------------------------------------------------------------------------------------------------------

# Creates the Monitor Thread

def DM\_createMonitorThreads(self):

self.DM\_monitor\_thread = hub.spawn(self.\_monitorLink\_SwitchStatsThread)

self.DM\_cpu\_stats\_server\_thread = hub.spawn(self.\_cpuStatsServerThread)

# Creates Server Thread to receive all CPU Statistics from the Hypervisors

def \_cpuStatsServerThread(self):

# Socket-level functions

self.m\_server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.m\_server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.m\_server\_socket.bind((IP\_ADDR, CPU\_STATS\_SERVER\_PORT))

self.m\_server\_socket.listen(10)

# Add server socket object to the list of readable connections

self.m\_hypervisor\_socket\_list.append(self.m\_server\_socket)

while 1:

# get the list sockets which are ready to be read through select

# 4th arg, time\_out = 0 : poll and never block

ready\_to\_read,ready\_to\_write,in\_error = select.select(self.m\_hypervisor\_socket\_list,[],[],0)

for sock in ready\_to\_read:

# new connection request recieved

if sock == self.m\_server\_socket:

sockfd,addr= self.m\_server\_socket.accept()

self.m\_hypervisor\_socket\_list.append(sockfd)

self.m\_hyp\_ip\_sockfd\_pair.append(make\_tuple(addr, sockfd))

# message from a Hypervisor system, not a new connection

else:

try:

data = sock.recv(RECV\_BUFFER)

if data:

# Function interprets the Docker stats information from a Hypervisor

self.processHypervisorCPUInfo(data)

else:

LOG\_DEBUG("Received empty data from Hypervisor. Closing Socket connection" % pair.val2)

# Remove the socket that's broken

if sock in self.m\_hypervisor\_socket\_list:

self.m\_hypervisor\_socket\_list.remove(sock)

# Clean up related data-structures

for pair in self.m\_hyp\_ip\_sockfd\_pair:

if pair.val2 == sock:

self.m\_hyp\_ip\_sockfd\_pair.remove(pair)

break

# exception

except:

continue

self.m\_server\_socket.close()

# Updates Hypervisor's CPU Utilization by Docker Containers

def processHypervisorCPUInfo(self, data):

global gNodeBottleneckHackCount

data = data.split(":")

hypervisorIP = str(data[0])

hypervisorRcvdTime = float(data[1])

percentCpuUsage = float(data[2])

dpid = ""

#LOG\_DEBUG("Received data from Hypervisor (%s) : %s" % (hypervisorIP, data))

for hDpid,ipAddr in self.m\_dpid\_to\_hyp\_ip.items():

if ipAddr == hypervisorIP:

dpid = hDpid

break

# Sanity Check

if dpid == "":

LOG\_DEBUG("Hypervisor IP (%s) to Dpid map not found. This scenario should not occur. Programming Error!!!" % (hypervisorIP))

return

# TODO: Bottleneck Detection Module

# Tweak

if False and dpid == 2:

gNodeBottleneckHackCount += 1

if gNodeBottleneckHackCount == 10:

LOG\_DEBUG("CPU Utilization Bottleneck detected at Hypervisor(%s). Neccessary actions to be taken." % hypervisorIP)

# Handle Node Bottleneck

self.handleNodeBottleneck(dpid)

if self.debugFlag:

print "Hypervisor [%s] at %s | %s" % (hypervisorIP, hypervisorRcvdTime, percentCpuUsage)

if dpid not in self.m\_hypervisorCpuUtilStats:

return

# 1st time information from the Hypervisor

if self.m\_hypervisorCpuUtilStats[dpid]['startRcvdTime'] == float(0.0):

self.m\_hypervisorCpuUtilStats[dpid]['startRcvdTime'] = hypervisorRcvdTime

self.m\_hypervisorCpuUtilStats[dpid]['data'] = percentCpuUsage

self.m\_hypervisorCpuUtilStats[dpid]['lastRcvdTime'] = hypervisorRcvdTime

if self.debugFlag:

LOG\_DEBUG("CPU Stat Update of %s @ %s -> %s" % (hypervisorIP,

str(self.m\_hypervisorCpuUtilStats[dpid]['lastRcvdTime']),

str(self.m\_hypervisorCpuUtilStats[dpid]['data'])))

self.write\_to\_file(CPU\_ANIM\_FILENAME, self.m\_hypervisorCpuUtilStats)

# Function requests Port Stats from Switches

def DM\_request\_stats(self, datapath):

currentTime = float('%.2f' % time.time())

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

dpid = datapath.id

if dpid not in self.m\_switchFlowStats:

self.m\_switchFlowStats[dpid] = {}

# Maintaing Stats Request time

self.m\_switchFlowStats[dpid]['LastSentTime'] = currentTime

req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP\_ANY)

datapath.send\_msg(req)

# Event - OFP Flow Stats Reply

@set\_ev\_cls(ofp\_event.EventOFPFlowStatsReply, MAIN\_DISPATCHER)

def \_flow\_stats\_reply\_handler(self, ev):

body = ev.msg.body

self.logger.info('datapath '

'in-port eth-dst '

'out-port packets bytes')

self.logger.info('---------------- '

'-------- ----------------- '

'-------- -------- --------')

for stat in sorted([flow for flow in body if flow.priority == 1],

key=lambda flow: (flow.match['in\_port'], flow.match['eth\_dst'])):

self.logger.info('%016x %8x %17s %8x %8d %8d',

ev.msg.datapath.id,

stat.match['in\_port'], stat.match['eth\_dst'],

stat.instructions[0].actions[0].port,

stat.packet\_count, stat.byte\_count)

# Event - OFP Port Stats Reply

@set\_ev\_cls(ofp\_event.EventOFPPortStatsReply, MAIN\_DISPATCHER)

def \_port\_stats\_reply\_handler(self, ev):

body = ev.msg.body

for stat in sorted(body, key=attrgetter('port\_no')):

if stat.port\_no != 0xfffffffe: # Ignoring Local Port

self.DM\_update\_bandwidth(ev.msg.datapath.id, stat.port\_no, stat.rx\_bytes, stat.tx\_bytes)

if LOGGING\_DEBUG:

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

self.logger.info('---------------- -------- -------- -------- -------- -------- -------- --------')

for stat in sorted(body, key=attrgetter('port\_no')):

self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',ev.msg.datapath.id, stat.port\_no,

stat.rx\_packets, stat.rx\_bytes, stat.rx\_errors, stat.tx\_packets, stat.tx\_bytes, stat.tx\_errors)

# Functions updates all-Ports Bandwidth acheived

def DM\_update\_bandwidth(self, dpid, port, rxBytes, txBytes):

# Sanity Checks

if dpid not in self.m\_switchFlowStats:

self.m\_switchFlowStats[dpid] = {}

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

# Initialize Port-Stats Monitoring

if port not in self.m\_switchFlowStats[dpid]:

currentTime = float('%.2f' % time.time())

self.m\_switchFlowStats[dpid][port] = {}

self.m\_switchFlowStats[dpid][port]['startRecordedTime'] = currentTime

self.m\_switchFlowStats[dpid][port]['LastTotalBytes'] = rxBytes + txBytes

self.m\_switchFlowStats[dpid][port]['LastRecordedTime'] = currentTime

self.m\_switchFlowStats[dpid][port]['data'] = []

else:

# Update Port-Stats Monitoring

currentTime = float('%.2f' % time.time())

currentBytesProcessed = rxBytes + txBytes

deltaBytesProcessed = abs(currentBytesProcessed - self.m\_switchFlowStats[dpid][port]['LastTotalBytes'])

deltaTime = currentTime - self.m\_switchFlowStats[dpid][port]['LastRecordedTime']

durationFromStart = currentTime - self.m\_switchFlowStats[dpid][port]['startRecordedTime']

# Sanity Check

if deltaTime == float(0.0):

LOG\_DEBUG("This scenario should not occur. Programming or System Error!!!")

return

bandwidthAcheived = float(deltaBytesProcessed \* 8)/float(float(UNIT\_OF\_BANDWIDTH) \* float(deltaTime)) # Kbps

self.m\_switchFlowStats[dpid][port]['data'] = [durationFromStart, bandwidthAcheived]

self.m\_switchFlowStats[dpid][port]['LastTotalBytes'] = currentBytesProcessed

self.m\_switchFlowStats[dpid][port]['LastRecordedTime'] = currentTime

self.write\_to\_file(THROUGHPUT\_ANIM\_FILENAME, self.m\_switchFlowStats)

# Functions updates Link Latency

def DM\_update\_latency(self, dpid, ethernet\_type, eth):

curr\_time = time.time() \* 1000 # In Milliseconds

src\_mac = eth.src

dst\_mac = eth.dst

# Sanity Check

if "00:00:00" not in dst\_mac:

return

### Switch Latency

if ethernet\_type == SWITCH\_LATENCY\_ETH\_TYPE:

# Format of Dst\_mac : "00:00:00:00:(dpid):(dpid)"

dpid = int(dst\_mac.split(":")[4].lstrip())

# Sanity Check

if dpid not in self.m\_switchLatencyStats or 'lastSentTime' not in self.m\_switchLatencyStats[dpid]:

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

return

# Update Switch-Latency Monitoring

sent\_time = self.m\_switchLatencyStats[dpid]['lastSentTime']

deltaSwitchLatencyTime = curr\_time - sent\_time

self.m\_switchLatencyStats[dpid]['data'] = float(deltaSwitchLatencyTime/2)

self.m\_switchLatencyStats[dpid]['lastUpdateTime'] = float(curr\_time)

### Link Latency

elif ethernet\_type == LINK\_LATENCY\_ETH\_TYPE:

# Format of Dst\_mac : "00:00:00:00:(dpid):(nbr\_dpid)"

dpid1 = int(dst\_mac.split(":")[4].lstrip())

dpid2 = int(dst\_mac.split(":")[5].lstrip())

# Sanity Check

if dpid1 not in self.m\_linkLatencyStats or dpid2 not in self.m\_linkLatencyStats[dpid1] or 'startTime' not in self.m\_linkLatencyStats[dpid1][dpid2]:

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

if self.debugFlag:

LOG\_DEBUG("Link Latency Packet: Src Dpid (%s) -> Dst Dpid (%s)" % (dpid1, dpid2))

return

# Check, required for Triangular Rule Calculation

if dpid1 not in self.m\_switchLatencyStats or dpid2 not in self.m\_switchLatencyStats:

LOG\_DEBUG("Switch Latency Module not initialized. Programming Error!!!")

return

# Calculate Latency since the last sent

deltaLinkLatency = curr\_time - self.m\_linkLatencyStats[dpid1][dpid2]['lastSentTime']

dpid1\_latency = self.m\_switchLatencyStats[dpid1]['data']

dpid2\_latency = self.m\_switchLatencyStats[dpid2]['data']

link\_latency = deltaLinkLatency - dpid1\_latency - dpid2\_latency

# Update Link Latency

self.m\_linkLatencyStats[dpid1][dpid2]['data'] = abs(link\_latency)

self.m\_linkLatencyStats[dpid1][dpid2]['lastUpdateTime'] = curr\_time

if False:

LOG\_DEBUG("Switch Latency : %s - %s seconds" % (dpid1, dpid1\_latency))

LOG\_DEBUG("Switch Latency : %s - %s seconds" % (dpid2, dpid2\_latency))

LOG\_DEBUG("Round Latency : %s <-> %s : %s seconds" % (dpid1, dpid2, deltaLinkLatency))

LOG\_DEBUG("Latency over the link %s <---> %s : %s seconds " % (dpid1, dpid2, abs(link\_latency)))

self.write\_to\_file(LATENCY\_ANIM\_FILENAME, self.m\_linkLatencyStats)

# Function writes data to a file

def write\_to\_file(self, fileName, data):

with open(fileName, mode='w') as outfile:

json.dump(data, outfile)

return

# -------------------------------------------------------------------------------------------------------------------------

# FLOW MANAGER MODULE : AS RYU CONTROLLER APPLICATION - OPENFLOW v1.3

# -------------------------------------------------------------------------------------------------------------------------

@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

# 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)]

self.add\_flow(datapath, 1, match, actions)

match = parser.OFPMatch(eth\_type=ether\_types.ETH\_TYPE\_ARP)

actions = [parser.OFPActionOutput(ofproto.OFPP\_CONTROLLER)]

self.add\_flow(datapath, 65535, match, actions)

# Adds Flow entry on the OVS connected with the datapath

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)

else:

mod = parser.OFPFlowMod(datapath=datapath, priority=priority,

match=match, instructions=inst)

datapath.send\_msg(mod)

# Deletes Flow entry on the OVS connected with the datapath

def del\_flow(self, datapath, priority, match):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

# OFPFC\_DELETE\_STRICT : Delete entry strictly matching wildcards and priority.

mod = parser.OFPFlowMod(datapath=datapath, priority=priority,

match=match, command=ofproto.OFPFC\_DELETE\_STRICT,

out\_port=ofproto.OFPP\_ANY,

out\_group=ofproto.OFPG\_ANY)

datapath.send\_msg(mod)

@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]

if self.debugFlag:

print "Packet passing started"

print "-------------------------------------------------------------------------------------------------------------------------------"

self.debugFlag = False

# Ignore LLDP and IPv6 packets

if eth.ethertype == ether\_types.ETH\_TYPE\_LLDP or \

eth.ethertype == ether\_types.ETH\_TYPE\_IPV6:

return

# Check for Link Monitoring Packets

if eth.ethertype in [LINK\_LATENCY\_ETH\_TYPE, SWITCH\_LATENCY\_ETH\_TYPE]:

# Handle Link Monitoring Packets

self.DM\_update\_latency(datapath.id, eth.ethertype, eth)

return

dst\_mac = eth.dst

src\_mac = eth.src

dpid = datapath.id

if "00:00:00:00:00" not in src\_mac and "00:00:00:00:00" not in dst\_mac:

return

match = []

actions = []

src\_dpid = dpid

retVal,dst\_dpid = self.get\_dpid\_from\_mac(dst\_mac)

# Check whether EndPoints belong to an SLA

isUsersBelongToSLA = self.checkEndUsersToSLA(src\_mac, dst\_mac)

# Unknown Destination Host

# Flood along the edge switches and the Hosts connected to the nodes of the spanning Tree

if eth.ethertype == ether\_types.ETH\_TYPE\_ARP or retVal != SUCCESS:

sendPktOutFlag = False

# Forwarding to the switches

if dpid in self.m\_graph['edges']:

for nbr\_dpid,out\_port in self.m\_graph['edges'][dpid].items():

if in\_port != out\_port and nbr\_dpid in self.m\_spTreeLinks and dpid in self.m\_spTreeLinks[nbr\_dpid]:

actions.append(datapath.ofproto\_parser.OFPActionOutput(out\_port))

sendPktOutFlag = True

if LOGGING\_DEBUG:

LOG\_DEBUG("%s | Switch: %s <-> Port : %s" % (dpid, nbr\_dpid, out\_port))

# Forwarding to the hosts

if dpid in self.m\_graph['hosts']:

for mac,out\_port in self.m\_graph['hosts'][dpid].items():

if out\_port != in\_port:

actions.append(datapath.ofproto\_parser.OFPActionOutput(out\_port))

sendPktOutFlag = True

if LOGGING\_DEBUG:

LOG\_DEBUG("%s | Host: %s <-> Port : %s" % (dpid, mac, out\_port))

# Send the packet

if sendPktOutFlag:

data = None

if msg.buffer\_id == ofproto.OFP\_NO\_BUFFER:

data = msg.data

out = datapath.ofproto\_parser.OFPPacketOut( datapath=datapath, buffer\_id=msg.buffer\_id, in\_port=in\_port, actions=actions, data=data)

datapath.send\_msg(out)

# First Packet of an SLA, installing respective flow rules

elif len(isUsersBelongToSLA):

# Retreive the IP packet

ip\_pkt = pkt.get\_protocols(ipv4.ipv4)[0]

sla\_object = 0

for sla\_object in isUsersBelongToSLA:

LOG\_DEBUG("%s & %s belong to SLA : %s" % (src\_mac, dst\_mac, sla\_object.identifier))

break

# Install SLA based Flow Rules (also, if updated)

if sla\_object.compPathOfSLA[src\_mac][dst\_mac]['isInstalled'] == False:

# Note Old Flow

oldPrioirty = sla\_object.compPathOfSLA[src\_mac][dst\_mac]['priority']

# TODO: Use only One function to handle First and Rest flow rule installs

self.installSLAFlowRules(sla\_object, src\_mac, dst\_mac, msg, ofproto, parser, in\_port, ip\_pkt.src, ip\_pkt.dst, sla\_object.compPathOfSLA[src\_mac][dst\_mac]['priority'])

sla\_object.compPathOfSLA[src\_mac][dst\_mac]['isInstalled'] = True

# Add New flow rules for this change with Higher Priority

self.updateNewSLAFlowRules(sla\_object, False, src\_mac, dst\_mac)

# Remove Previous Flow Rules

if sla\_object.compPathOfSLA[src\_mac][dst\_mac]['prevPath'] != []:

self.removeOldSLAFlowRules(sla\_object, src\_mac, dst\_mac, sla\_object.compPathOfSLA[src\_mac][dst\_mac]['prevPath'])

sla\_object.compPathOfSLA[src\_mac][dst\_mac]['prevPath'] = []

LOG\_DEBUG("Flow rules installed and Current Packet forwarded.")

# Tweak : aich Node Bottleneck

global gNodeBottleneckHackCount

if gNodeBottleneckHackCount == True:

gNodeBottleneckHackCount = False

LOG\_DEBUG("Tweaked Bottleneck detected at Hypervisor(3).")

self.handleNodeBottleneck(3)

# Known Host - Install flow rules along the path

else:

LOG\_DEBUG("Known Host - Install flow rules along the path")

LOG\_DEBUG("This Scenario should not occur!!!")

# Assign VNF Resources

def assignVNFResources(self, sla, index, dpid, isMigrated):

global gUsedVNFIds, gUsedVNFCount

gUsedVNFCount += 1

# Find the VNF Id not in use

for item, val in gUsedVNFIds.items():

if gUsedVNFIds[item] == False and item >= 50: # TODO

vnfId = item

break

ipAddr = DOCKER\_SUB\_NET + str(vnfId)

macAddr = "00:00:00:00:00:" + str(vnfId).zfill(2)

# Update VNF MAC for the Migrated VNF

if isMigrated:

sla.VNFsNetworkMac[index] = macAddr

else:

sla.VNFsNetworkMac.append(macAddr)

vnfInfo = struct\_Installed\_VNF(vnfId, sla.identifier, sla.vnfInputList[index], ipAddr, macAddr, sla.vnfCPUMemReq[index], dpid, index)

self.m\_hypervisorMemStats[dpid]['used'] += sla.vnfCPUMemReq[index]

gUsedVNFIds[item] = vnfInfo

return gUsedVNFIds[item]

# Removes assigned VNF Resouces

def recoverVNFResources(self, sla, vnfInfo):

global gUsedVNFIds, gUsedVNFCount

vnfId = vnfInfo.identifier

dpid = vnfInfo.dpid

# Recover Hypervisor CPU Mem Utilization

self.m\_hypervisorMemStats[dpid]['used'] -= vnfInfo.cpuMemReq

# VNF Id to be re-used

gUsedVNFIds[vnfInfo.identifier] = False # Empty Slot

gUsedVNFCount -= 1

return

# Creates message to Start VNF at Hypervisor

def sendStartVNFCommand(self, vnfInfo, sockfd):

container\_type = vnfInfo.vnfType.lower()

container\_id = vnfInfo.identifier

cont\_name = "c%s" % (container\_id)

cont\_mac = vnfInfo.macAddr

cont\_ip = vnfInfo.ipAddr

msg = ""

# Firewall

if container\_type == "firewall":

cont\_ip = "192.168.111.%s" % (container\_id)

msg = "docker run -d --mac-address=\"%s\" --name %s --network=%s --ip=%s --privileged kalilinux/kali-linux-docker sleep 10000" % (cont\_mac, cont\_name, DOCKER\_SUBNET\_NAME, cont\_ip)

# BusyBox

elif container\_type == "busy\_box":

cont\_ip = "192.168.111.%s" % (container\_id)

mag = "docker run -d --mac-address=\"%s\" --name %s --network=%s --ip=%s centos:latest sleep 10000" % (cont\_mac, cont\_name, DOCKER\_SUBNET\_NAME, cont\_ip)

# Web Server

elif container\_type == "webserver":

cont\_ip = "192.168.111.%s" % (container\_id)

msg = "docker run -d --mac-address=\"%s\" --name %s --network=%s --ip=%s nginx\_server:1.0" % (cont\_mac, cont\_name, DOCKER\_SUBNET\_NAME, cont\_ip)

# FTP Server

elif container\_type == "ftp":

cont\_ip = "192.168.111.%s" % (container\_id)

msg = "docker run -d --mac-address=\"%s\" --name %s --network=%s --ip=%s -p 21:21 -e \"PUBLICHOST=localhost\" ftpd\_server:1.0" % (cont\_mac, cont\_name, DOCKER\_SUBNET\_NAME, cont\_ip)

message = "FTP Server: Please enter login credentials in FTP server."

else:

LOG\_DEBUG("Container %s not supported." % container\_type)

# Send Message to Hypervisor

sockfd.send(msg)

hub.sleep(0.5)

# Creates message to Stop VNF at Hypervisor

def sendStopVNFCommand(self, vnfInfo, sockfd):

cont\_name = "c%s" % (vnfInfo.identifier)

# Force the removal of a running container (uses SIGKILL)

msg = "docker rm --force %s" % cont\_name

# Send Message to Hypervisor

sockfd.send(msg)

hub.sleep(0.5)

# Creates message for Forwarding in VNFs

def sendForwadingVNFCommand(self, vnfInfo, forwardToIP, sockfd, isLastVNFToDest):

container\_type = vnfInfo.vnfType.lower()

container\_id = vnfInfo.identifier

cont\_name = "c%s" % (container\_id)

cont\_mac = vnfInfo.macAddr

cont\_ip = vnfInfo.ipAddr

msgs = []

msgs.append("docker exec %s echo 1 \> /proc/sys/net/ipv4/ip\_forward" % cont\_name)

msgs.append("docker exec %s iptables -F" % cont\_name)

msgs.append("docker exec %s iptables -A OUTPUT -j DROP" % cont\_name)

if isLastVNFToDest == False:

msgs.append("docker exec %s iptables -A FORWARD -i eth0 -o eth0 --to-destination %s -j ACCEPT" % (cont\_name, forwardToIP))

#msgs.append("docker exec %s iptables -t nat -A PREROUTING -j DNAT --to-destination %s" % (cont\_name, forwardToIP))

else:

msgs.append("docker exec %s iptables -A FORWARD -i eth0 -o eth0 -j ACCEPT" % cont\_name)

# Send Messages to Hypervisor

for msg in msgs:

sockfd.send(msg)

hub.sleep(0.5)

# Installs VNFs at their respective Hypervisors

# Used only during Placement Algorithm

def installVNFsAtHypervisors(self, sla, toBeInstalledVNFInfo):

## Step - 1: Start Msg Command

for vnfInfo in toBeInstalledVNFInfo:

# Retrieve Hypervisor IP

hypIPAddr = self.m\_dpid\_to\_hyp\_ip[vnfInfo.dpid]

sockfd = ""

for pair in self.m\_hyp\_ip\_sockfd\_pair:

if hypIPAddr == pair.val1[0]:

sockfd = pair.val2

break

# Sanity Check

if sockfd == "":

LOG\_DEBUG("Incorrect placement at Hypervisor (%s) or System communication with Hypervisor(%s) is broken." % (vnfInfo.dpid, vnfInfo.dpid))

return FAILURE

# Send Commands to Hypervisor System

self.sendStartVNFCommand(vnfInfo, sockfd)

# Map VNFInfo to Hypervisor

# Sanity Check

if vnfInfo.dpid not in self.m\_hypVNFStatus:

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

return

# Map VNF Information w.r.t. Hypervisor

self.m\_hypVNFStatus[vnfInfo.dpid].append(vnfInfo.identifier)

## Step - 2 : Forwarding Msg Command

destHostIP = DOCKER\_SUB\_NET + str(sla.endUsersMac[1].split(":")[5])

for index in range(0, len(toBeInstalledVNFInfo)):

vnfInfo = toBeInstalledVNFInfo[index]

# Retrieve Hypervisor IP

hypIPAddr = self.m\_dpid\_to\_hyp\_ip[vnfInfo.dpid]

sockfd = ""

for pair in self.m\_hyp\_ip\_sockfd\_pair:

if hypIPAddr == pair.val1[0]:

sockfd = pair.val2

break

# Sanity Check

if sockfd == "":

LOG\_DEBUG("Incorrect placement at Hypervisor (%s) or System communication with Hypervisor(%s) is broken." % (vnfInfo.dpid, vnfInfo.dpid))

return FAILURE

# Check for last VNF

if index + 1== len(toBeInstalledVNFInfo):

break

nextVNFInfo = toBeInstalledVNFInfo[index + 1]

# Forward to Next VNF in the Service Chain

self.sendForwadingVNFCommand(vnfInfo, nextVNFInfo.ipAddr, sockfd, False)

# Forward from Last VNF to Destination Host

self.sendForwadingVNFCommand(vnfInfo, destHostIP, sockfd, True)

return SUCCESS

# Actual Placement of SLA-defined Flow Rules

def placementOfSLA(self, sla):

LOG\_DEBUG("Placement for SLA (%s) started." % sla.identifier)

## Step 1: Find Centre for this SLA to place

## the 1st VNF of the SLA Service Chain

# Initialize the 'Seen' DS,

# i.e. The list of EndPoints it has seen so far

seenEndPoints = {}

delayFromEndPoint = {}

# For Every 'vertex' to maintain

# 1. list of EndPoints it has seen

# 2. Delay from each End Point

for datapath in self.m\_graph['switches']:

uVertex = datapath.id

seenEndPoints[uVertex] = []

delayFromEndPoint[uVertex] = {}

for endPointDpid in sla.endPointsDpid:

delayFromEndPoint[uVertex][endPointDpid] = 0.0

dctEndPoints = {}

# For Every 'endPoint' start Dijkstra, iteratively

for endPoint in sla.endPointsDpid:

dctEndPoints[endPoint] = {}

dctEndPoints[endPoint]['queue'] = [endPoint]

dctEndPoints[endPoint]['visited'] = set()

dctEndPoints[endPoint]['visited'].add(endPoint)

dctEndPoints[endPoint]['parent'] = {}

for datapath in self.m\_graph['switches']:

uVertex = datapath.id

dctEndPoints[endPoint]['parent'][uVertex] = -1

dctEndPoints[endPoint]['parent'][endPoint] = endPoint

seenEndPoints[endPoint].append(endPoint)

queueEndPoints = deque()

for endDpid in sla.endPointsDpid:

queueEndPoints.append(endDpid)

# CPU Mem Requirement of 1st VNF of Chain

cpuMemReq = sla.vnfCPUMemReq[0]

# Temporary Information of the VNFs

toBeInstalledVNFInfo = []

while len(queueEndPoints):

endPoint = queueEndPoints.popleft()

if len(dctEndPoints[endPoint]['queue']):

uVertex = dctEndPoints[endPoint]['queue'].pop(0)

for vVertex in self.m\_graph['edges'][uVertex]:

# Do not consider the Parent vertex

if vVertex == dctEndPoints[endPoint]['parent'][uVertex]:

continue

# Check for constraints

if vVertex not in dctEndPoints[endPoint]['visited'] and endPoint not in seenEndPoints[vVertex]:

# 'C1' (Dynamic Constraint) - Constraint for Edge's Latency/Delay

# Retreive Link latency

linkLatency = 0.0

if uVertex in self.m\_linkLatencyStats and vVertex in self.m\_linkLatencyStats[uVertex]:

linkLatency = self.m\_linkLatencyStats[uVertex][vVertex]['data']

elif vVertex in self.m\_linkLatencyStats and uVertex in self.m\_linkLatencyStats[vVertex]:

linkLatency = self.m\_linkLatencyStats[vVertex][uVertex]['data']

if delayFromEndPoint[uVertex][endPoint] + linkLatency <= sla.delayTolerated :

delayFromEndPoint[vVertex][endPoint] = delayFromEndPoint[uVertex][endPoint] + linkLatency

else:

continue

# 'C2' (Static Constraint) : Constraint for CPU utilization

if cpuMemReq > (self.m\_hypervisorMemStats[vVertex]['capacity'] - self.m\_hypervisorMemStats[vVertex]['used']):

continue

# 'C3' (Dynamic Constraint) : Constraint for Link's Available BW

uVertex\_to\_vVertexPort = self.m\_graph['edges'][uVertex][vVertex]

linkBandwidthAchieved = self.m\_switchFlowStats[uVertex][uVertex\_to\_vVertexPort]['data'][1]

if sla.reqBW > LINK\_BW\_THRESHOLD \* (LINK\_BW\_LIMIT - linkBandwidthAchieved):

continue

dctEndPoints[endPoint]['visited'].add(vVertex)

# Add EndPoint to Seen DS

seenEndPoints[vVertex].append(endPoint)

# Update Parent Pointer

dctEndPoints[endPoint]['parent'][vVertex] = uVertex

# Push into the Queue

dctEndPoints[endPoint]['queue'].append(vVertex)

# Check whether all End points are observed

if len(seenEndPoints[vVertex]) == len(sla.endPointsDpid):

# Center Found - Eureka!!!

sla.centre = vVertex

dctEndPoints[endPoint]['queue'] = []

queueEndPoints = []

# Update CPU Memory Used at Hypervisor

self.m\_hypervisorMemStats[sla.centre]['used'] += cpuMemReq

break

if len(dctEndPoints[endPoint]['queue']):

queueEndPoints.append(endPoint)

if sla.centre == -1:

# TODO : Moving SLA to the Cloud

LOG\_DEBUG("SLA (%s) cannot be placed in the Current Network. Moving the entire SLA to the Cloud." % sla.identifier)

return

sla.pathToCentre = self.getPathFromCentreToAllEndPoints(sla, dctEndPoints)

# 1st VNF of the SLA Chain

toBeInstalledVNFInfo.append(self.assignVNFResources(sla, 0, sla.centre, False))

sla.pathOfServiceChain.append(sla.centre)

sla.VNFsNetworkDpid.append(sla.centre)

# Hardcoded VLAN Start Tag to distinguish between SLAs

sla.vlanCommonTag = 0x1000 | (sla.identifier << 8)

# Update Delay Buffer used for future migration purposes

for item,val in delayFromEndPoint[vVertex].items():

if val <= sla.delayTolerated:

sla.delayBuffer = max(sla.delayBuffer, val)

sla.delayBuffer = sla.delayTolerated - sla.delayBuffer

if sla.delayBuffer == -1.0:

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

# Recover assigned Resources

for vnfInfo in toBeInstalledVNFInfo:

self.recoverVNFResources(sla, vnfInfo)

return

# Step 2: Map rest of the VNFs

lastMappedVNFDpid = sla.centre

if len(sla.vnfInputList) > 1: # Service Chain

for index in range(1, len(sla.vnfInputList)):

dpid = lastMappedVNFDpid

slaType = sla.vnfInputList[index]

cpuMemReq = sla.vnfCPUMemReq[index]

# Case 1 : Map to the same Hypervisor, if possible

# Assuming, Link Bandwidth and Link Latency within the Hypervisor is infinite and zero, respectively

# 'C1' (Static Constraint) : Constraint for CPU utilization

if False and cpuMemReq <= (self.m\_hypervisorMemStats[dpid]['capacity'] - self.m\_hypervisorMemStats[dpid]['used']):

toBeInstalledVNFInfo.append(self.assignVNFResources(sla, index, dpid, False))

# Case 2 : Map to neighbors within Link Latency constraints

else:

# 'C1' (Dynamic Constraint) - Constraint for Edge's Latency/Delay

# Retreive Link latency

nbr\_latency = {}

for nbr\_dpid in self.m\_graph['edges'][dpid]:

linkLatency = 0.0

if dpid in self.m\_linkLatencyStats and nbr\_dpid in self.m\_linkLatencyStats[dpid]:

linkLatency = self.m\_linkLatencyStats[dpid][nbr\_dpid]['data']

elif nbr\_dpid in self.m\_linkLatencyStats and dpid in self.m\_linkLatencyStats[nbr\_dpid]:

linkLatency = self.m\_linkLatencyStats[nbr\_dpid][dpid]['data']

# Check within Delay Buffer of SLA

if sla.delayBuffer < linkLatency:

continue

# 'C2' (Static Constraint) : Constraint for CPU utilization

if cpuMemReq > (self.m\_hypervisorMemStats[nbr\_dpid]['capacity'] - self.m\_hypervisorMemStats[nbr\_dpid]['used']):

continue

# 'C3' (Dynamic Constraint) : Constraint for Link's Available BW

dpid\_to\_nbrDpid\_port = self.m\_graph['edges'][dpid][nbr\_dpid]

linkBandwidthAchieved = self.m\_switchFlowStats[uVertex][dpid\_to\_nbrDpid\_port]['data'][1]

if sla.reqBW > LINK\_BW\_THRESHOLD \* (LINK\_BW\_LIMIT - linkBandwidthAchieved):

continue

# All Constraints satisfied...

nbr\_latency[nbr\_dpid] = linkLatency

# Sanity Check

if len(nbr\_latency) == 0:

LOG\_DEBUG("VNF(%s) of SLA (%s) cannot be placed at any Nbrs of %s." % (index, sla.identifier, dpid))

LOG\_DEBUG("SLA (%s) cannot be placed in the Current Network. Moving the entire SLA to the Cloud." % (sla.identifier))

# Recover assigned Resources

for vnfInfo in toBeInstalledVNFInfo:

self.recoverVNFResources(sla, vnfInfo)

return

# Greedy Approach : Choose the Neighbor with minimum Latency

minLatency = float("inf")

for nbr\_dpid, linkLatency in nbr\_latency.items():

if linkLatency < minLatency:

minLatency = linkLatency

selected\_dpid = nbr\_dpid

# Sanity Check

if minLatency == float("inf"):

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

LOG\_DEBUG("SLA (%s) cannot be placed in the Current Network. Moving the entire SLA to the Cloud." % (sla.identifier))

# Recover assigned Resources

for vnfInfo in toBeInstalledVNFInfo:

self.recoverVNFResources(sla, vnfInfo)

return

# Update Information about the VNF

sla.delayBuffer = sla.delayBuffer - minLatency

lastMappedVNFDpid = selected\_dpid

toBeInstalledVNFInfo.append(self.assignVNFResources(sla, index, selected\_dpid, False))

sla.VNFsNetworkDpid.append(lastMappedVNFDpid)

sla.pathOfServiceChain.append(lastMappedVNFDpid)

LOG\_DEBUG("SLA (%s) Placed at Hypervisor(s) (%s) with overall Delay Buffer (%s)." % (sla.identifier, sla.pathOfServiceChain, sla.delayBuffer))

# Step 3: Install VNF's of the SLA at their assigned Hypervisor's

if FAILURE == self.installVNFsAtHypervisors(sla, toBeInstalledVNFInfo):

LOG\_DEBUG("SLA (%s) cannot be placed in the Current Network. System issue!!!" % (sla.identifier))

# Recover assigned Resources

for vnfInfo in toBeInstalledVNFInfo:

self.recoverVNFResources(sla, vnfInfo)

return

# Step 4: Installation of the respective flow Rules will be done when

# End-points intiates communication

# Reference : installSLAFlowRules(sla)

# Yes Yes Yes... SLA is properly installed over the Network

sla.isInstalled = True

vnfsMACAddr = []

for vnfInfo in toBeInstalledVNFInfo:

vnfsMACAddr.append(vnfInfo.macAddr)

# Note the complete path of the SLA in the network for all Hosts in the SLA

sla.compPathOfSLA = self.retreivePathOfSLA(sla)

LOG\_DEBUG("SLA (%s) is installed over the Network with the VNFs placed at"

" Hypervisors (%s) having MAC addresses (%s)." %

(sla.identifier, sla.pathOfServiceChain, vnfsMACAddr))

# Removes consecutive duplicates from Path of Service Chain

# e.g. Input - 2 3 3 4 5 5 5 6

# Output - [[2 1] [3 2] [4 1] [5 3] [6 1]]

def convertPathOfServiceChain(self, inList):

outList = []

outListLen = 1

outList.append([inList[0], outListLen])

for index in range(1, len(inList)):

if outList[outListLen-1][0] == inList[index]:

outList[outListLen-1][1] += 1

else:

outList.append([inList[index], 1])

outListLen += 1

return outList

# Retreives the complete Path of the SLA in the Network

def retreivePathOfSLA(self, sla):

allPaths = {}

for src\_mac in sla.endUsersMac:

for dst\_mac in sla.endUsersMac:

# Sanity Check

if src\_mac == dst\_mac:

continue

path = []

src\_mac = src\_mac

dst\_mac = dst\_mac

src\_dpid = self.m\_mac\_to\_dpid\_port[src\_mac].val1

dst\_dpid = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

flowpath\_src\_to\_centre = sla.pathToCentre[src\_dpid]

flowpath\_src\_to\_centre = list(reversed(flowpath\_src\_to\_centre))

# -1 indicates an End-Point

path = [[flowpath\_src\_to\_centre[0], -1]] # Source Hypervisor

# Intermediate Switches between {Src Host to Centre}

if len(flowpath\_src\_to\_centre) > 2:

for index in range(1, len(flowpath\_src\_to\_centre) - 1):

path.append([flowpath\_src\_to\_centre[index], 0])

# Path of the Service Chain

midPathOfServiceChain = self.convertPathOfServiceChain(sla.pathOfServiceChain)

for item in midPathOfServiceChain:

path.append(item)

flowpath\_last\_to\_dst = self.getSrcToDestPath(sla.pathOfServiceChain[-1], dst\_dpid)

# Intermediate Switches between {Last VNF and Dst}

if len(flowpath\_last\_to\_dst) > 2:

for index in range(1, len(flowpath\_last\_to\_dst) - 1):

path.append([flowpath\_last\_to\_dst[index], 0])

# -1 indicates an End-Point

path.append([flowpath\_last\_to\_dst[-1], -1]) # Destination Hypervisor

# Complete Path of SLA from Srouce to Destination

if src\_mac not in allPaths:

allPaths[src\_mac] = {}

if dst\_mac not in allPaths[src\_mac]:

allPaths[src\_mac][dst\_mac] = {}

allPaths[src\_mac][dst\_mac]['prevPath'] = []

allPaths[src\_mac][dst\_mac]['currPath'] = path

allPaths[src\_mac][dst\_mac]['isInstalled'] = False

allPaths[src\_mac][dst\_mac]['priority'] = START\_PRIORITY\_VAL

# Path of SLA Chain

if True or self.debugFlag:

LOG\_DEBUG("SLA Complete Path (Src %s to Dst %s) : %s" % (src\_dpid, dst\_dpid, path))

return allPaths

# Installs the Flow Rules w.r.t. SLA using Hard Timeout

def installSLAFlowRules(self, sla, src\_mac, dst\_mac, msg, ofproto, parser, pkt\_in\_port, src\_ip, dst\_ip, priority):

processed\_vnf\_cnt = 0

## 1. Actor - Ingress Switch via In-between Switches

src\_dpid = self.m\_mac\_to\_dpid\_port[src\_mac].val1

flowpath\_src\_to\_centre = sla.pathToCentre[src\_dpid]

flowpath\_src\_to\_centre = list(reversed(flowpath\_src\_to\_centre))

centre\_dpid = sla.VNFsNetworkDpid[0]

prev\_switch = src\_dpid

# TODO: Src End point and 1st VNF at the same place

# Packet reaches from End-point (Src. Point) to the 1st VNF (centre)

if len(flowpath\_src\_to\_centre) != 1:

for i in range(1, len(flowpath\_src\_to\_centre)-1):

this\_dpid = flowpath\_src\_to\_centre[i]

datapath = self.getDatapath(this\_dpid)

next\_switch = flowpath\_src\_to\_centre[i+1]

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

in\_port = self.m\_graph['edges'][this\_dpid][prev\_switch]

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

## 2. Actor - All the VNFs of the Service Chain

## Packet starts processing from 1st VNF to the rest of the VNF(s), if any

flowpath\_start\_to\_last = sla.pathOfServiceChain

prev\_in\_port = pkt\_in\_port

if len(flowpath\_start\_to\_last):

# Check whether multiple VNF(s) are connected to the same Hypervisor

isMultipleVNFsAtSameDpid = False

for i in range(0, len(flowpath\_start\_to\_last)):

this\_dpid = flowpath\_start\_to\_last[i]

datapath = self.getDatapath(this\_dpid)

in\_port = self.m\_graph['edges'][this\_dpid][prev\_switch] if this\_dpid != prev\_switch else prev\_in\_port

if i != len(flowpath\_start\_to\_last)-1:

next\_switch = flowpath\_start\_to\_last[i+1]

out\_port = ofproto.OFPP\_IN\_PORT if this\_dpid == next\_switch else self.m\_graph['edges'][this\_dpid][next\_switch]

# If installed VNF's dpid

if this\_dpid == sla.VNFsNetworkDpid[processed\_vnf\_cnt]:

# Change dst\_Mac to mac\_address of the VNF

vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt]

vnf\_out\_port = self.m\_mac\_to\_dpid\_port[vnf\_mac].val2

vnf\_out\_port = ofproto.OFPP\_IN\_PORT if vnf\_out\_port == in\_port else vnf\_out\_port

# If multiple VNFs at same Hypervisor

if isMultipleVNFsAtSameDpid:

# Do nothing, Packet sent in previous iteration

isMultipleVNFsAtSameDpid = False

# Receiving Packet for first time

elif this\_dpid == prev\_switch:

match = parser.OFPMatch(in\_port=in\_port,eth\_src=src\_mac,eth\_dst=dst\_mac)

actions = [parser.OFPActionSetField(eth\_src=self.m\_ovs\_mac[this\_dpid]),

parser.OFPActionSetField(eth\_dst=vnf\_mac),

parser.OFPActionOutput(vnf\_out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, in\_port, src\_mac, dst\_mac, vnf\_mac, vnf\_out\_port))

else:

# Check VLAN Tag

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt),in\_port=in\_port,eth\_src=src\_mac,eth\_dst=dst\_mac)

actions = [parser.OFPActionPopVlan(0x8100),

parser.OFPActionSetField(eth\_src=self.m\_ovs\_mac[this\_dpid]),

parser.OFPActionSetField(eth\_dst=vnf\_mac),

parser.OFPActionOutput(vnf\_out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_vid-%s in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, vnf\_mac, vnf\_out\_port))

processed\_vnf\_cnt += 1

# Packet receiving from Last VNF

# Scenario Handled in next Section.

if i == len(flowpath\_start\_to\_last) - 1:

break

# Re-direct the packet received from the VNF through the Service Chain/Destination

match = parser.OFPMatch(in\_port=vnf\_out\_port, eth\_src=vnf\_mac, eth\_dst=dst\_mac)

# Next VNF at the same Hypervisor

if this\_dpid == next\_switch:

out\_port = ofproto.OFPP\_IN\_PORT

next\_vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt]

actions = [parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionSetField(eth\_dst=next\_vnf\_mac),

parser.OFPActionOutput(out\_port)]

isMultipleVNFsAtSameDpid = True

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, eth\_dst-%s Output : out\_port-%s" % (this\_dpid, vnf\_out\_port, vnf\_mac, dst\_mac, src\_mac, next\_vnf\_mac, out\_port))

else:

# Add VLAN Tag for processing in next Switch(s)

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

actions = [parser.OFPActionPushVlan(0x8100),

parser.OFPActionSetField(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt)),

parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionOutput(out\_port)]

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : vlan\_tag-%s, eth\_src - %s, Output : out\_port-%s" % (this\_dpid, vnf\_out\_port, vnf\_mac, dst\_mac, sla.vlanCommonTag | processed\_vnf\_cnt, src\_mac, out\_port))

self.add\_flow(datapath, priority, match, actions)

# Intermediate Switches

else:

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

## 3. Actor - Last VNF and Intermediate Switches

# Re-direct the packet from Last VNF to the Egress Point (Actual Dest. Point)

last\_vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt - 1]

last\_vnf\_dpid = sla.VNFsNetworkDpid[processed\_vnf\_cnt - 1]

flowpath\_last\_to\_dst = self.getSrcToDestPath(last\_vnf\_dpid, self.m\_mac\_to\_dpid\_port[dst\_mac].val1)

dst\_dpid = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

## Case 3a. Destination VNF Hypervisor is also connected to the Egress Switch

if len(flowpath\_last\_to\_dst) == 0:

# Re-direct the packet from the VNF towards the destination

datapath = self.getDatapath(dst\_dpid)

same\_port = self.m\_mac\_to\_dpid\_port[dst\_mac].val2

match = parser.OFPMatch(in\_port=same\_port, eth\_src=last\_vnf\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionOutput(ofproto.OFPP\_IN\_PORT)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, Output : out\_port-%s" % (dst\_dpid, same\_port, last\_vnf\_mac, dst\_mac, src\_mac, ofproto.OFPP\_IN\_PORT))

## Case 3b. Last VNF Hypervisor and Egress Switch are not same

else:

last\_vnf\_in\_port = self.m\_dpid\_to\_mac\_port[last\_vnf\_dpid][last\_vnf\_mac]

for i in range(0, len(flowpath\_last\_to\_dst)-1):

this\_dpid = flowpath\_last\_to\_dst[i]

datapath = self.getDatapath(this\_dpid)

next\_switch = flowpath\_last\_to\_dst[i+1]

in\_port = last\_vnf\_in\_port if i == 0 else self.m\_graph['edges'][this\_dpid][prev\_switch]

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

out\_port = ofproto.OFPP\_IN\_PORT if out\_port == in\_port else out\_port

# Re-format the packet from the Last VNF towards the actual destination

if this\_dpid == last\_vnf\_dpid:

match = parser.OFPMatch(in\_port=in\_port, eth\_src=last\_vnf\_mac, eth\_dst=dst\_mac)

# Add VLAN Tag for processing in the last Switch

actions = [parser.OFPActionPushVlan(0x8100),

parser.OFPActionSetField(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt)),

parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : vlan\_tag-%s eth\_src - %s, Output : out\_port-%s" % (this\_dpid, in\_port, last\_vnf\_mac, dst\_mac, sla.vlanCommonTag | processed\_vnf\_cnt, src\_mac, out\_port))

# Intermediate Switches

else:

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

## Case 4. Actor - Egress Switch

if len(flowpath\_last\_to\_dst) != 1:

datapath = self.getDatapath(dst\_dpid)

in\_port = self.m\_graph['edges'][dst\_dpid][prev\_switch]

out\_port = self.m\_mac\_to\_dpid\_port[dst\_mac].val2

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_dst=dst\_mac)

actions = [parser.OFPActionPopVlan(0x8100),

parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_dst-%s | Output : out\_port-%s" % (dst\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, dst\_mac, out\_port))

## 5. Actor - Ingress Switch

# Case 5a. Ingrees Switch and Centre Switch are not same

if len(flowpath\_src\_to\_centre) != 1:

datapath = self.getDatapath(src\_dpid)

next\_switch = flowpath\_src\_to\_centre[1]

out\_port = self.m\_graph['edges'][src\_dpid][next\_switch]

match = parser.OFPMatch(in\_port=pkt\_in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

# Add VLAN Tag for processing in next Switch(s)

actions = [parser.OFPActionPushVlan(0x8100),

parser.OFPActionSetField(vlan\_vid=sla.vlanCommonTag),

parser.OFPActionOutput(out\_port)]

if pkt\_in\_port == out\_port:

return

self.add\_flow(datapath, priority, match, actions) # Flow-Mod

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | Output : vlan\_vid-%s out\_port-%s" % (src\_dpid, pkt\_in\_port, src\_mac, dst\_mac, sla.vlanCommonTag, out\_port))

data = msg.data if msg.buffer\_id == ofproto.OFP\_NO\_BUFFER else None

out\_msg = datapath.ofproto\_parser.OFPPacketOut(datapath=datapath, buffer\_id=msg.buffer\_id, in\_port=pkt\_in\_port, actions=actions, data=data)

LOG\_DEBUG("PacketOUT - Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | Output : vlan\_vid-%s out\_port-%s" % (src\_dpid, pkt\_in\_port, src\_mac, dst\_mac, sla.vlanCommonTag, out\_port))

datapath.send\_msg(out\_msg) # Packet-Out

# Case 5b:(Special) Handling Packet-Out when Src-switch and Centre-Switch are the same

# Flow-mod is handled earlier for this scenario

else:

vnf\_mac = sla.VNFsNetworkMac[0]

same\_port = self.m\_mac\_to\_dpid\_port[vnf\_mac].val2

actions = [parser.OFPActionSetField(eth\_src=self.m\_ovs\_mac[this\_dpid]),

parser.OFPActionSetField(eth\_dst=vnf\_mac),

parser.OFPActionOutput(ofproto.OFPP\_IN\_PORT)]

data = msg.data if msg.buffer\_id == ofproto.OFP\_NO\_BUFFER else None

out\_msg = datapath.ofproto\_parser.OFPPacketOut(datapath=datapath, buffer\_id=msg.buffer\_id, in\_port=pkt\_in\_port, actions=actions, data=data)

LOG\_DEBUG("PacketOUT - Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, pkt\_in\_port, src\_mac, dst\_mac, vnf\_mac, vnf\_out\_port))

datapath.send\_msg(out\_msg) # Packet-Out

# Removes Flow Rule entries for the input path

def removeOldSLAFlowRules(self, sla, src\_mac, dst\_mac, path):

pdb.set\_trace()

LOG\_DEBUG("-----------------------------------------------------------------------------------------")

LOG\_DEBUG("Handling Path : %s -> %s" % (src\_mac, dst\_mac))

src\_dpid = self.m\_mac\_to\_dpid\_port[src\_mac].val1

dst\_dpid = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

priority = sla.compPathOfSLA[src\_mac][dst\_mac]['priority'] - 1

# Delete Flow for the Old/Previous Path

processed\_vnf\_cnt = 0

src\_dpid = self.m\_mac\_to\_dpid\_port[src\_mac].val1

dst\_dpid = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

src\_in\_port = self.m\_mac\_to\_dpid\_port[src\_mac].val2

## 1. Actor - Ingress Switch via In-between Switches

prev\_switch = src\_dpid

slaIndex = 1

while slaIndex < len(path):

# Pair contains [dpid, Information]

# Information : {-1 : End Point, 0 : Intermediate Switch, <Num> : No. of VNFs isntalled}

pair = path[slaIndex]

# Reached the Switch with the First VNF installed

if pair[1] != 0:

break

this\_dpid = pair[0]

datapath = self.getDatapath(this\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

next\_switch = path[slaIndex + 1][0]

# Assuming Hosts connected to the Hypervisor's

in\_port = self.m\_graph['edges'][this\_dpid][prev\_switch] if this\_dpid != prev\_switch else src\_in\_port

out\_port = ofproto.OFPP\_IN\_PORT if this\_dpid == next\_switch else self.m\_graph['edges'][this\_dpid][next\_switch]

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

slaIndex += 1

## 2. Actor - All the VNFs of the Service Chain

## Packet starts processing from 1st VNF to the rest of the VNF(s), if any

prev\_in\_port = src\_in\_port

while True:

# Check whether multiple VNF(s) are connected to the same Hypervisor

isMultipleVNFsAtSameDpid = False

this\_dpid = path[slaIndex][0]

datapath = self.getDatapath(this\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

in\_port = self.m\_graph['edges'][this\_dpid][prev\_switch] if this\_dpid != prev\_switch else prev\_in\_port

prev\_in\_port = in\_port

next\_switch = path[slaIndex + 1][0]

out\_port = ofproto.OFPP\_IN\_PORT if this\_dpid == next\_switch else self.m\_graph['edges'][this\_dpid][next\_switch]

# VNF installed over this dpid

if path[slaIndex][1] != 0:

# Change dst\_Mac to mac\_address of the VNF

vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt]

vnf\_out\_port = self.m\_mac\_to\_dpid\_port[vnf\_mac].val2

vnf\_out\_port = ofproto.OFPP\_IN\_PORT if vnf\_out\_port == in\_port else vnf\_out\_port

# If multiple VNFs at same Hypervisor

if isMultipleVNFsAtSameDpid:

# Do nothing, Packet sent in previous iteration

isMultipleVNFsAtSameDpid = False

# Receiving Packet for first time

elif this\_dpid == prev\_switch:

match = parser.OFPMatch(in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, in\_port, src\_mac, dst\_mac, vnf\_mac, vnf\_out\_port))

else:

# Check VLAN Tag

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt),in\_port=in\_port,eth\_src=src\_mac,eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : vlan\_vid-%s in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, self.m\_ovs\_mac[this\_dpid], vnf\_mac, vnf\_out\_port))

processed\_vnf\_cnt += 1

# Re-direct the packet received from the VNF through the Service Chain/Destination

match = parser.OFPMatch(in\_port=vnf\_out\_port, eth\_src=vnf\_mac, eth\_dst=dst\_mac)

# Next VNF at the same Hypervisor

if this\_dpid == next\_switch:

out\_port = ofproto.OFPP\_IN\_PORT

next\_vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt - 1]

isMultipleVNFsAtSameDpid = True

LOG\_DEBUG("-- Removed Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, eth\_dst-%s Output : out\_port-%s" % (this\_dpid, vnf\_out\_port, vnf\_mac, dst\_mac, src\_mac, next\_vnf\_mac, out\_port))

else:

# Add VLAN Tag for processing in next Switch(s)

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

LOG\_DEBUG("-- Removed Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : vlan\_tag-%s, eth\_src - %s, Output : out\_port-%s" % (this\_dpid, vnf\_out\_port, vnf\_mac, dst\_mac, sla.vlanCommonTag | processed\_vnf\_cnt, src\_mac, out\_port))

self.del\_flow(datapath, priority, match)

# Intermediate Switches

else:

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt),in\_port=in\_port,eth\_src=src\_mac,eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

slaIndex += 1

# All VNF(s) have been processed

if processed\_vnf\_cnt == len(sla.VNFsNetworkDpid):

break

## 3. Actor - Last VNF and Intermediate Switches

# Re-direct the packet from Last VNF to the Egress Point (Actual Dest. Point)

last\_vnf\_mac = sla.VNFsNetworkMac[-1]

last\_vnf\_dpid = sla.VNFsNetworkDpid[-1]

## Case 3a. Destination VNF Hypervisor is also connected to the Egress Switch

if prev\_switch == dst\_dpid:

# Re-direct the packet from the VNF towards the destination

datapath = self.getDatapath(dst\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

same\_port = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

match = parser.OFPMatch(in\_port=same\_port,eth\_src=last\_vnf\_mac,eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, Output : out\_port-%s" % (dst\_dpid, same\_port, last\_vnf\_mac, dst\_mac, src\_mac, ofproto.OFPP\_IN\_PORT))

## Case 3b. Last VNF Hypervisor and Egress Switch are not same

else:

last\_vnf\_in\_port = self.m\_dpid\_to\_mac\_port[last\_vnf\_dpid][last\_vnf\_mac]

while path[slaIndex][1] != -1:

this\_dpid = path[slaIndex][0]

datapath = self.getDatapath(this\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

next\_switch = path[slaIndex + 1][0]

in\_port = last\_vnf\_in\_port if slaIndex == 0 else self.m\_graph['edges'][this\_dpid][prev\_switch]

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

out\_port = ofproto.OFPP\_IN\_PORT if out\_port == in\_port else out\_port

# Re-format the packet from the Last VNF towards the actual destination

if this\_dpid == last\_vnf\_dpid:

match = parser.OFPMatch(in\_port=in\_port, eth\_src=last\_vnf\_mac, eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : vlan\_tag-%s eth\_src - %s, Output : out\_port-%s" % (this\_dpid, in\_port, last\_vnf\_mac, dst\_mac, sla.vlanCommonTag | processed\_vnf\_cnt, src\_mac, out\_port))

# Intermediate Switches

else:

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

slaIndex += 1

## Case 4. Actor - Egress Switch

if prev\_switch != dst\_dpid:

datapath = self.getDatapath(dst\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

in\_port = self.m\_graph['edges'][dst\_dpid][prev\_switch]

out\_port = self.m\_mac\_to\_dpid\_port[dst\_mac].val2

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_dst=dst\_mac)

self.del\_flow(datapath, priority, match)

LOG\_DEBUG("-- Removed Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_dst-%s | Output : out\_port-%s" % (dst\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, dst\_mac, out\_port))

## 5. Actor - Ingress Switch

# Case 5a. Ingrees Switch and Centre Switch are not same

if len(path) > 2 and path[0][0] != path[1][0]:

datapath = self.getDatapath(src\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

next\_switch = path[1][0]

out\_port = self.m\_graph['edges'][src\_dpid][next\_switch]

match = parser.OFPMatch(in\_port=src\_in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

# Add VLAN Tag for processing in next Switch(s)

if src\_in\_port == out\_port:

return

self.del\_flow(datapath, priority, match) # Flow-Mod

LOG\_DEBUG("-- Removed Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | Output : vlan\_vid-%s out\_port-%s" % (src\_dpid, src\_in\_port, src\_mac, dst\_mac, sla.vlanCommonTag, out\_port))

# Returns the Path from Source Dpid to the Dest. Dpid

def getSrcToDestPath(self, src, dst):

#Sanity Check

if src == dst:

return [src]

path = []

# Simple BFS on computed Spanning Tree Links

queue = deque()

color = {}

parent = {}

for node in self.m\_graph['switches']:

color[node.id] = COLOR\_WHITE

parent[node.id] = ""

queue.append(src)

parent[src] = -1

while queue:

u = queue.popleft()

color[u] = COLOR\_BLACK

for v in self.m\_spTreeLinks[u]:

# TODO : Improve by using weights among the edges

# Boils down to simple BFS

if color[v] == COLOR\_WHITE:

parent[v] = u

if v == dst:

path.append(v)

while parent[v] != -1:

v = parent[v]

path.append(v)

break

else:

color[v] = COLOR\_BLACK

queue.append(v)

return list(reversed(path))

# Retreive Path from the Centre to all End Points

def getPathFromCentreToAllEndPoints(self, sla, dctEndPoints):

# This is the entire path from Centre to each EndPoint of the SLA

pathToCentre = {}

for endPoint in sla.endPointsDpid:

pathToCentre[endPoint] = []

current = sla.centre

# Traverse parent pointers to reach the EndPoint

while current != dctEndPoints[endPoint]['parent'][current]:

pathToCentre[endPoint].append(current)

if current == -1: # Invalid Parent

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

break

current = dctEndPoints[endPoint]['parent'][current]

pathToCentre[endPoint].append(endPoint)

return pathToCentre

def get\_sla\_flow\_path(self, src\_dpid, dst\_dpid, sla):

# Sanity Check

if sla.isInstalled == 0:

LOG\_DEBUG("SLA (%s) is not mapped." % sla.identifier + 1)

return

# Algorithm for VNF(s) Placement

# Returns dpid of the switch to which the host(mac) is connected to

def get\_dpid\_from\_mac(self, mac):

if mac in self.m\_mac\_to\_dpid\_port:

value = self.m\_mac\_to\_dpid\_port[mac]

return SUCCESS, value.val1

else:

return FAILURE,None

# Checks whether a Switch exists or not

def IsSwitchExist(self, dpid):

for dp in self.m\_graph['switches']:

if dp.id == dpid:

return SUCCESS

return FAILURE

def getDatapath(self, dpid):

datapath = None

# Retreive datapath of 'this\_switch'

for dp in self.m\_graph['switches']:

if dp.id == dpid:

datapath = dp

break

if datapath == None:

print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

LOG\_DEBUG("datapath not found , switches:")

for val in self.m\_graph['switches']:

print val.id

print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

return datapath

def DFS\_visit(self, src\_dpid, dst\_dpid, color, path\_list):

u = src\_dpid

for v, port in self.m\_graph['edges'][u].items():

if v in self.m\_spTreeLinks and u in self.m\_spTreeLinks[v]:

if color[v] == COLOR\_WHITE:

color[v] = COLOR\_BLACK

# Found dst\_dpid

if v == dst\_dpid:

path\_list = [int(dst\_dpid)]

return SUCCESS, path\_list

else:

ret\_flag, ret\_path\_list = self.DFS\_visit(v, dst\_dpid, color, path\_list)

if ret\_flag == SUCCESS:

ret\_path\_list.append(int(v))

return ret\_flag, ret\_path\_list

return FAILURE, None

# Switch Enter Event

@set\_ev\_cls(event.EventSwitchEnter)

def update\_topology\_switchEnter(self, ev):

if LOGGING\_DEBUG:

LOG\_DEBUG("Event: Switch Added")

# Retreive the list of all switches available in the topology

all\_switch\_info = get\_all\_switch(self.m\_topology\_api\_app)

# Add Switch

for switch in all\_switch\_info:

dpid = switch.dp.id

if switch.dp not in self.m\_graph['switches']:

# Check whether any VNF is to be installed

self.m\_graph['switches'].add(switch.dp)

# Assign a MAC address to the OVS for Service chaining

if dpid not in self.m\_ovs\_mac:

self.m\_ovs\_mac[dpid] = "00:00:00:%s%s:%s%s:%s%s" % (dpid,dpid,dpid,dpid,dpid,dpid)

if dpid not in self.m\_graph['edges']:

self.m\_graph['edges'].setdefault(dpid, {})

if dpid not in self.m\_graph['hosts']:

self.m\_graph['hosts'].setdefault(dpid,{})

# Initialize Switch-Latency Monitoring

if dpid not in self.m\_switchLatencyStats:

self.m\_switchLatencyStats[dpid] = {}

self.m\_switchLatencyStats[dpid]['lastSentTime'] = float(0.0)

self.m\_switchLatencyStats[dpid]['lastUpdateTime'] = float(0.0)

self.m\_switchLatencyStats[dpid]['data'] = float(0.0)

# Initialize CPU Memory Stats

# TODO : Assuming, a Hypervisor is connected to only one OVS-switch

if dpid not in self.m\_hypervisorMemStats:

self.m\_hypervisorMemStats[dpid] = {}

self.m\_hypervisorMemStats[dpid]['capacity'] = ALL\_HYPERVISOR\_CPU\_MEMORY\_INIT

self.m\_hypervisorMemStats[dpid]['used'] = 0

# Initialize CPU Percent Utilization Stats

if dpid not in self.m\_hypervisorCpuUtilStats:

self.m\_hypervisorCpuUtilStats[dpid] = {}

self.m\_hypervisorCpuUtilStats[dpid]['startRcvdTime'] = float(0.0)

self.m\_hypervisorCpuUtilStats[dpid]['lastRcvdTime'] = float(0.0)

self.m\_hypervisorCpuUtilStats[dpid]['data'] = float(0.0)

# Initialize Hypervisor VNF Status

if dpid not in self.m\_hypVNFStatus:

self.m\_hypVNFStatus[dpid] = []

# Update network's Spanning Tree

for datapath in self.m\_graph['switches']:

self.\_calculate\_SpanningTree(datapath.id)

break

# Find appropriate location for the VNF,

# Static: Hardcoded

# Dynamic : TODO To be decided by the Orchestrator's logic

def findApprLocForVNF(self, switchName):

if switchName == 's1':

return 's2'

else:

print "Programming Error!!!"

# Switch Leave event

@set\_ev\_cls(event.EventSwitchLeave)

def update\_topology\_switchLeave(self, ev):

if LOGGING\_DEBUG:

LOG\_DEBUG("Event: Switch Deleted - Re-calculating Spanning Tree")

# Retreive the list of all switches available in the topology

all\_switch\_info = get\_all\_switch(self.m\_topology\_api\_app)

findFlag = False

# Find the switch which has left

for addedSwitchInfo in self.m\_graph['switches']:

findFlag = False

for switch in all\_switch\_info:

# Match dpid's

if addedSwitchInfo.id == switch.dp.id:

findFlag = True

break

if findFlag == False:

leftSwitch = addedSwitchInfo

break

# Sanity Check

if findFlag == True:

return

# Remove Switch entries

self.m\_graph['switches'].remove(leftSwitch)

dpid = leftSwitch.id

if dpid in self.m\_ovs\_mac:

del(self.m\_ovs\_mac[dpid])

for other\_dpid,linked\_dpid in self.m\_graph['edges'].items():

for link,port in linked\_dpid.items():

if link == dpid:

self.m\_graph['edges'][other\_dpid].pop(link)

break

if dpid in self.m\_graph['edges']:

self.m\_graph['edges'].pop(dpid)

LOG\_DEBUG(self.m\_graph['edges'])

if dpid in self.m\_graph['hosts']:

self.m\_graph['hosts'].pop(dpid)

LOG\_DEBUG(self.m\_graph['hosts'])

if dpid in self.m\_dpid\_to\_mac\_port:

for key,value in self.m\_dpid\_to\_mac\_port[dpid].items():

mac = key

self.m\_mac\_to\_dpid\_port.pop(mac, None)

if dpid in self.m\_dpid\_to\_mac\_port:

self.m\_dpid\_to\_mac\_port.pop(dpid, None)

LOG\_DEBUG(self.m\_dpid\_to\_mac\_port)

# Remove Switch-Latency Monitoring Entry

if dpid in self.m\_switchLatencyStats:

del self.m\_switchLatencyStats[dpid]

# Remove Link-Latency Monitoring Entry

if dpid in self.m\_linkLatencyStats:

del self.m\_linkLatencyStats[dpid]

# Remove Hypervisor Memory Stats

if dpid in self.m\_hypervisorMemStats:

del self.m\_hypervisorMemStats[dpid]

# Remove Hypervisor CPU Utilization Stats

if dpid in self.m\_hypervisorCpuUtilStats:

del self.m\_hypervisorCpuUtilStats[dpid]

# Remove Hypervisor VNF Status

if dpid in self.m\_hypVNFStatus:

del self.m\_hypVNFStatus[dpid]

# Update network's Spanning Tree

for datapath in self.m\_graph['switches']:

self.\_calculate\_SpanningTree(datapath.id)

break

# Host Add Event

@set\_ev\_cls(event.EventHostAdd, MAIN\_DISPATCHER)

def update\_topology\_hostAdd(self, ev):

# Host information

mac = ev.host.mac

dpid = ev.host.port.dpid

port = ev.host.port.port\_no

# Code commented for Testing with Mininet

if "00:00:00:00:00" not in mac:

return

LOG\_DEBUG("Host added @ dpid - %s : %s via port - %s" % (dpid, mac, port))

# Sanity Check

if dpid in self.m\_graph['hosts'] and mac in self.m\_graph['hosts'][dpid]:

return

# Maintain the Data-structures

# m\_graph['hosts'] [dpid] [mac] => port

if mac not in self.m\_graph['hosts'][dpid]:

self.m\_graph['hosts'][dpid].setdefault(mac,{})

self.m\_graph['hosts'][dpid][mac]=port

# mac -> (dpid,port)

self.m\_mac\_to\_dpid\_port[mac] = make\_tuple(dpid, port)

# dpid -> (mac, port)

if dpid not in self.m\_dpid\_to\_mac\_port:

self.m\_dpid\_to\_mac\_port.setdefault(dpid, {})

self.m\_dpid\_to\_mac\_port[dpid][mac] = port

if False and LOGGING\_DEBUG:

LOG\_DEBUG("Saved Host information : " + str(mac) + " : " + str(dpid) + " : " + str(port))

# If VNF is a Migrated, context will be maintained

if mac in self.m\_vnfOperationsOnStart:

# Complete rest of the actions

self.migrateVNF\_RestOp(self.m\_vnfOperationsOnStart[mac], dpid)

del self.m\_vnfOperationsOnStart[mac]

# Update network's Spanning Tree

#LOG\_DEBUG("Event: Host Added @ %s : %s - Re-calcuting Spanning Tree" % (dpid,mac))

# Link add event

@set\_ev\_cls(event.EventLinkAdd)

def update\_topology\_linkAdd(self, ev):

link = ev.link

# Switches inter-connections

self.m\_graph['edges'][link.src.dpid][link.dst.dpid] = int(link.src.port\_no)

self.m\_graph['edges'][link.dst.dpid][link.src.dpid] = int(link.dst.port\_no)

if LOGGING\_DEBUG:

LOG\_DEBUG("Saved Link Information: " + str(link.src.dpid) + " <-> " + str(link.dst.dpid))

LOG\_DEBUG("Event: Link Added - Re-calcuting Spanning Tree")

# Update network's Spanning Tree used for Flooding purpose

for datapath in self.m\_graph['switches']:

self.\_calculate\_SpanningTree(datapath.id)

break

# Updates the controller's view of switches in network topology

def \_calculate\_SpanningTree(self, src):

self.m\_spTreeLinks = {}

queue = deque()

color = {}

for node in self.m\_graph['switches']:

self.m\_spTreeLinks[node.id] = set()

color[node.id] = COLOR\_WHITE

queue.append(src)

while queue:

u = queue.popleft()

color[u] = COLOR\_BLACK

for v,port in self.m\_graph['edges'][u].items():

# TODO : Improve by using weights among the edges

# Boils down to simple BFS

if color[v] == COLOR\_WHITE:

self.m\_spTreeLinks[v].add(u)

self.m\_spTreeLinks[u].add(v)

color[v] = COLOR\_BLACK

queue.append(v)

if LOGGING\_DEBUG:

LOG\_DEBUG("Spanning Tree: " )

for u in self.m\_graph['switches']:

LOG\_DEBUG( str(u.id) + " <-> " + str(self.m\_spTreeLinks[u.id]))

def \_DFS\_visit(self, src\_dpid, dst\_dpid, color, path\_list):

u = src\_dpid

for v, port in self.m\_graph['edges'][u].items():

if v in self.m\_spTreeLinks and u in self.m\_spTreeLinks[v]:

if color[v] == COLOR\_WHITE:

color[v] = COLOR\_BLACK

# Found dst\_dpid

if v == dst\_dpid:

path\_list = [dst\_dpid]

return SUCCESS, path\_list

else:

ret\_flag, ret\_path\_list = self.\_DFS\_visit(v, dst\_dpid, color, path\_list)

if ret\_flag == SUCCESS:

ret\_path\_list.append(v)

return ret\_flag, ret\_path\_list

return FAILURE, None

def get\_flow\_path(self, dpid1, dpid2):

ret\_path\_list = []

# Sanity Check, same dpid

if dpid1 == dpid2:

ret\_path\_list.append(int(dpid1))

ret\_path\_list.append(int(dpid1))

return ret\_path\_list

# Run DFS to get the flow-path

color = {}

path\_list = []

for u in self.m\_graph['switches']:

color[u.id] = COLOR\_WHITE

color[dpid1] = COLOR\_BLACK

ret\_flag, ret\_path\_list = self.DFS\_visit(dpid1, dpid2, color, path\_list)

ret\_path\_list.append(int(dpid1))

return ret\_path\_list

# ---------------------------------------------

# ORCHESTRATOR's MESSAGE HANDLERS

# ---------------------------------------------

# Handles messages from Flow Manager

def \_DO\_handleFlowManagerMessage(self, message):

return

LOG\_DEBUG("-------------------------------")

LOG\_DEBUG("Flow Manager says :")

LOG\_DEBUG(message)

LOG\_DEBUG("-------------------------------")

# Handles messages from bottleneck Detector

def \_DO\_handleDetectorMessage(self, message):

LOG\_DEBUG("-------------------------------")

LOG\_DEBUG("Detector says :")

LOG\_DEBUG(message)

LOG\_DEBUG("-------------------------------")

# Handle messages from Operator API

def \_DO\_handleOperatorMessage(self, message, sock):

retMsg = ""

# Commands

msgPart = message.split()

# List Information - Read-Only Operation

if msgPart[0] == 'list':

# Format - "list all"

if len(msgPart) == 1 or msgPart[1] == "all":

retMsg = "Every Switch, VNFs and their links\n"

retMsg += "Switches: \n"

for item in self.m\_graph['switches']:

LOG\_DEBUG("----")

LOG\_DEBUG(item.id)

LOG\_DEBUG("----")

retVal,anot\_msg = self.\_getSwitchInfo(item.id)

LOG\_DEBUG(str(retVal) + " : " + str(anot\_msg))

if retVal == SUCCESS:

retMsg += anot\_msg

retMsg += "VNFs: \n"

if self.m\_dpid\_to\_vnf.items:

for key,val in self.m\_dpid\_to\_vnf.items():

retMsg += str(key) + " : "

for item in val:

retMsg += str(item) + " "

retMsg += '\n'

else:

retMsg += "No VNFs are currently installed\n"

# Format - "list switches"

elif msgPart[1] == "switches":

retMsg = "All switch Information\n"

for item in self.m\_graph['switches']:

retVal,anot\_msg = self.\_getSwitchInfo(item.id)

if retVal == SUCCESS:

retMsg += anot\_msg

# Format - "list <switch\_name>"

# ex. list s1

# Shows all the links connected to this 'specific' switch and all VNFs connected to this.

elif msgPart[1] == "switch":

retMsg = "Switch specific information :\n"

dpid = int(msgPart[2].lstrip('s'))

retVal,retMsg = self.\_getSwitchInfo(dpid) # Retreive Switch Info

if retVal == SUCCESS:

retVal,anot\_msg = self.\_getLinkedVNF(dpid) # Retreive Linked VNFs

retMsg = retMsg + anot\_msg if retVal == SUCCESS else anot\_msg

# Format - "list nfvs"

# Shows all NFVs with whom they are connected

elif msgPart[1] == "vnfs":

retMsg = ""

if self.m\_dpid\_to\_vnf:

retMsg = "All NFVs installed\n"

for key,val in self.m\_dpid\_to\_vnf.items():

retMsg += str(key)

for item in val:

retMsg += str(item) + " "

retMsg += '\n'

else:

retMsg += 'No VNFs are currently installed.\n'

# Format - "list nfv <nfv\_name>"

# ex. list nfv f2

elif msgPart[1] == "vnf" and msgPart[2] is not None:

retMsg = ""

vNFid = msgPart[2].lstrip('f')

# Retreive VNF Info

retVal,anot\_msg = self.\_getVNFInfo(vNFid)

retMsg = "NFV specific information\n" + anot\_msg if retVal == SUCCESS else anot\_msg

else:

retMsg = "Invalid List Command\n"

retMsg += "List arguments:\n"

retMsg += "List all information - list all\n"

retMsg += "List all switches - list switches\n"

retMsg += "List specific switch - list switch <switch\_name>\n"

retMsg += "List all VNFs - list vnf\n"

retMsg += "List specific VNF - list vnf <vnf\_name>\n"

elif msgPart[0] == 'add':

if len(msgPart) == 3:

# Format - "add <switch\_name> <nfv\_name>"

# ex. add s1 f2

# Adds a VNF with a specific Switch

retMsg = "VNF added\n"

else:

retMsg = "Invalid Command\n"

retMsg += "add <switch\_name> <nfv\_name>\n"

# Reply to Operator

sock.send(retMsg)

# ------------------------

# Sub-Functions

# ------------------------

def \_getSwitchInfo(self, dpid):

retMsg = "Switch s%s -\n" % dpid

checkSwitch = False

# Validate Switch

for switch in self.m\_graph['switches']:

if switch.id == dpid:

checkSwitch = True

# Get Linked VNF

if checkSwitch:

retVal,anot\_msg = self.\_getLinkedVNF(dpid)

if retVal != SUCCESS:

return retVal,anot\_msg

else:

retMsg += "VNFs: " + anot\_msg

else:

return FAILURE,"Switch not found.\n"

# Get Linked Nbrs

retMsg += "Nbrs: "

retVal, anot\_msg = self.\_getSwitchNbrs(dpid)

return (SUCCESS,retMsg + anot\_msg) if retVal == SUCCESS else (retVal,anot\_msg)

# Returns all VNFs linked to the Switch

def \_getLinkedVNF(self, dpid):

retMsg = ""

if self.m\_dpid\_to\_vnf and dpid in self.m\_dpid\_to\_vnf:

for item in self.m\_dpid\_to\_vnf[dpid]:

retMsg += str(item) + " "

retMsg += '\n'

return SUCCESS,retMsg

else:

return SUCCESS,"No VNF attached to this Switch\n"

# Returns Switch Connected to the VNF

def \_getVNFInfo(self, vNFid):

if self.m\_dpid\_to\_vnf:

for key,value in self.m\_dpid\_to\_vnf.items():

if value == vNFid:

return SUCCESS, "Connected to: s%s\n" % key

return FAILURE, "VNF not found.\n"

# Returns all neighbours of the Switch

def \_getSwitchNbrs(self, dpid):

retMsg = ""

if dpid in self.m\_graph['edges']:

for item in self.m\_graph['edges'][dpid]:

retMsg += "s" + str(item).lstrip('0') + " "

retMsg += '\n'

return SUCCESS,retMsg

else:

return FAILURE, "Switch Not found\n"

# -----------------------------------------

# Orchestor's Operator Thread - using Sockets

# -----------------------------------------

def DO\_openSocketThreadAPI(self):

try:

# Create the Orchestrator Thread

self.socket\_thread = hub.spawn(self.\_DO\_operatorSocket)

except:

LOG\_DEBUG("Error: unable to start Orchestrator thread")

def \_DO\_operatorSocket(self):

connectionList = []

server\_socket = socket.socket() # Create a socket object

host = IP\_ADDR # Get local machine name

port = OPERATOR\_API\_SOCK\_PORT # Reserve a port for your service.

server\_socket.bind((host, port)) # Bind to the port

server\_socket.listen(5) # Now wait for client connection.

# Add server socket to the list of readable connections

connectionList.append(server\_socket)

while True:

read\_sockets,write\_sockets,error\_sockets = select.select(connectionList,[],[])

for sock in read\_sockets:

# New connection

if sock == server\_socket:

sockfd, addr = server\_socket.accept()

connectionList.append(sockfd)

LOG\_DEBUG("Operator (%s) connected" % addr)

print "Operator (%s) connected" % addr

# Incoming message from Operator API

else:

# Data recieved

try:

message = sock.recv(RECV\_BUFFER)

if message:

# Handle Operator Message

# Push Message into Orchestrator Queue

customMsg = {}

customMsg['type'] = "Operator"

customMsg['message'] = message

customMsg['socket'] = sock

self.orchestratorMsgQueue.put(customMsg)

LOG\_DEBUG("Recvd Message: %s" % customMsg)

except:

sock.close()

connectionList.remove(sock)

continue

server\_socket.close()

# -----------------------------------------

# SLA's related Functions

# -----------------------------------------

# Maps EndUsers to SLA aggrements

def mapEndUsersToSLAs(self, sla\_object):

for end\_user in sla\_object.endUsersMac:

# Initialize

if end\_user not in self.m\_end\_users\_to\_SLA:

self.m\_end\_users\_to\_SLA[end\_user] = []

# The Endpoint may belong to many SLA's

self.m\_end\_users\_to\_SLA[end\_user].append(sla\_object.identifier)

# Allocate VNF ID

vnfID = int(end\_user.split(":")[5])

if gUsedVNFIds[vnfID] == False: # Empty Slot

gUsedVNFIds[vnfID] = True # Used by Host

# Check wether end-users belong to an SLA

def checkEndUsersToSLA(self, endUser1, endUser2):

# End-users may belong to many SLAs

result = []

if endUser1 in self.m\_end\_users\_to\_SLA:

for sla\_id in self.m\_end\_users\_to\_SLA[endUser1]:

sla\_object = self.m\_SLAs[sla\_id]

# Both endpoints belong to the same SLA

if endUser2 in sla\_object.endUsersMac:

result.append(sla\_object)

return result

# -----------------------------------------

# Bottleneck Removal related Functions

# -----------------------------------------

# Creates Priority Score of moving VNF to Nbr Hypervisor

def generatePriorityScore(self, vnfInfo, dpid, nbr\_dpid):

# Get linked SLA

sla = self.m\_SLAs[vnfInfo.slaIdentifier]

delayBuffer = sla.delayBuffer

#pdb.set\_trace()

#### Parameter - 1: Link Latency between Current and Nbr

if dpid > nbr\_dpid:

linkLatency = self.m\_linkLatencyStats[dpid][nbr\_dpid]['data']

else:

linkLatency = self.m\_linkLatencyStats[nbr\_dpid][dpid]['data']

# TODO: Improve based on the existing Path of the Service Chain

# Case 1 : Along the path of the Service Chain

# Case 2 : Addition in the path

#### Parameter - 2: VNF's CPU Mem Requirement

cpuMemReq = vnfInfo.cpuMemReq

cpuMemUnused = self.m\_hypervisorMemStats[nbr\_dpid]['capacity'] - self.m\_hypervisorMemStats[nbr\_dpid]['used']

if cpuMemReq > cpuMemUnused:

return [], FAILURE

# CPU Mem Remaining after Moving the VNF to the Nbr

paramCpuMemRemain = cpuMemUnused - cpuMemReq

#### Parameter - 3: Link Bandwidth between Current and Nbr

if dpid > nbr\_dpid:

curr\_to\_nbrPort = self.m\_graph['edges'][dpid][nbr\_dpid]

linkBandwidthUsed = self.m\_switchFlowStats[dpid][curr\_to\_nbrPort]['data'][1]

else:

curr\_to\_nbrPort = self.m\_graph['edges'][nbr\_dpid][dpid]

linkBandwidthUsed = self.m\_switchFlowStats[nbr\_dpid][curr\_to\_nbrPort]['data'][1]

linkBandwidthReq = sla.reqBW

# TODO : Link capacity for each link is different

#paramLinkBandwidthUnused = self.m\_switchFlowStats[dpid][curr\_to\_nbrPort]['capacity'] - linkBandwidthUsed

paramLinkBandwidthUnused = LINK\_BW\_LIMIT - linkBandwidthUsed

if linkBandwidthReq > paramLinkBandwidthUnused:

return [], FAILURE

# ppscore

priorityScore = (alpha \* paramCpuMemRemain) + (beta \* paramLinkBandwidthUnused) - (gamma \* linkLatency)

return priorityScore,SUCCESS

# Updates new SLA Flow Rules

def updateNewSLAFlowRules(self, sla, isForAllPairs, in\_src\_mac, in\_dst\_mac):

#pdb.set\_trace()

if isForAllPairs:

srcUsersMac = sla.endUsersMac

dstUsersMac = sla.endUsersMac

else:

srcUsersMac = [in\_src\_mac]

dstUsersMac = [in\_dst\_mac]

for src\_mac in srcUsersMac:

for dst\_mac in dstUsersMac:

if src\_mac == dst\_mac:

continue

# Sanity Check

if src\_mac not in sla.compPathOfSLA and dst\_mac not in sla.compPathOfSLA[src\_mac]:

LOG\_DEBUG("This scenario should not occur. Programming Error!!!")

continue

# Installation of the respective flow Rules will be done when End-points intiates communication

# Already installed Flow Rules have to be overwritten

# Reference : installSLAFlowRules(sla)

if sla.compPathOfSLA[src\_mac][dst\_mac]['isInstalled'] == True:

# Use Old prioirty to delete previous set of Flow Rules

priority = sla.compPathOfSLA[src\_mac][dst\_mac]['priority']

path = sla.compPathOfSLA[src\_mac][dst\_mac]['currPath']

# Add New Flows for the Current Path

processed\_vnf\_cnt = 0

src\_dpid = self.m\_mac\_to\_dpid\_port[src\_mac].val1

dst\_dpid = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

src\_in\_port = self.m\_mac\_to\_dpid\_port[src\_mac].val2

## 1. Actor - Ingress Switch via In-between Switches

prev\_switch = src\_dpid

slaIndex = 1

while slaIndex < len(path):

# Pair contains [dpid, Information]

# Information : {-1 : End Point, 0 : Intermediate Switch, <Num> : No. of VNFs isntalled}

pair = path[slaIndex]

# Reached the Switch with the First VNF installed

if pair[1] != 0:

break

this\_dpid = pair[0]

datapath = self.getDatapath(this\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

next\_switch = path[slaIndex + 1][0]

# TODO : Assuming Hosts connected to the Hypervisor's

in\_port = self.m\_graph['edges'][this\_dpid][prev\_switch] if this\_dpid != prev\_switch else src\_in\_port

out\_port = ofproto.OFPP\_IN\_PORT if this\_dpid == next\_switch else self.m\_graph['edges'][this\_dpid][next\_switch]

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("New Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

slaIndex += 1

## 2. Actor - All the VNFs of the Service Chain

## Packet starts processing from 1st VNF to the rest of the VNF(s), if any

prev\_in\_port = src\_in\_port

while True:

# Check whether multiple VNF(s) are connected to the same Hypervisor

isMultipleVNFsAtSameDpid = False

this\_dpid = path[slaIndex][0]

datapath = self.getDatapath(this\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

in\_port = self.m\_graph['edges'][this\_dpid][prev\_switch] if this\_dpid != prev\_switch else prev\_in\_port

prev\_in\_port = in\_port

next\_switch = path[slaIndex + 1][0]

out\_port = ofproto.OFPP\_IN\_PORT if this\_dpid == next\_switch else self.m\_graph['edges'][this\_dpid][next\_switch]

# VNF installed over this dpid

if path[slaIndex][1] != 0:

# Change dst\_Mac to mac\_address of the VNF

vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt]

vnf\_out\_port = self.m\_mac\_to\_dpid\_port[vnf\_mac].val2

vnf\_out\_port = ofproto.OFPP\_IN\_PORT if vnf\_out\_port == in\_port else vnf\_out\_port

# If multiple VNFs at same Hypervisor

if isMultipleVNFsAtSameDpid:

# Do nothing, Packet sent in previous iteration

isMultipleVNFsAtSameDpid = False

# Receiving Packet for first time

elif this\_dpid == prev\_switch:

match = parser.OFPMatch(in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionSetField(eth\_src=self.m\_ovs\_mac[this\_dpid]),

parser.OFPActionSetField(eth\_dst=vnf\_mac),

parser.OFPActionOutput(vnf\_out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, in\_port, src\_mac, dst\_mac, vnf\_mac, vnf\_out\_port))

else:

# Check VLAN Tag

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt),in\_port=in\_port,eth\_src=src\_mac,eth\_dst=dst\_mac)

actions = [parser.OFPActionPopVlan(0x8100),

parser.OFPActionSetField(eth\_src=self.m\_ovs\_mac[this\_dpid]),

parser.OFPActionSetField(eth\_dst=vnf\_mac),

parser.OFPActionOutput(vnf\_out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_vid-%s in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, eth\_dst - %s, Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, self.m\_ovs\_mac[this\_dpid], vnf\_mac, vnf\_out\_port))

processed\_vnf\_cnt += 1

# Re-direct the packet received from the VNF through the Service Chain/Destination

match = parser.OFPMatch(in\_port=vnf\_out\_port, eth\_src=vnf\_mac, eth\_dst=dst\_mac)

# Next VNF at the same Hypervisor

if this\_dpid == next\_switch:

out\_port = ofproto.OFPP\_IN\_PORT

next\_vnf\_mac = sla.VNFsNetworkMac[processed\_vnf\_cnt - 1]

actions = [parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionSetField(eth\_dst=next\_vnf\_mac),

parser.OFPActionOutput(out\_port)]

isMultipleVNFsAtSameDpid = True

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, eth\_dst-%s Output : out\_port-%s" % (this\_dpid, vnf\_out\_port, vnf\_mac, dst\_mac, src\_mac, next\_vnf\_mac, out\_port))

else:

# Add VLAN Tag for processing in next Switch(s)

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

actions = [parser.OFPActionPushVlan(0x8100),

parser.OFPActionSetField(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt)),

parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionOutput(out\_port)]

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : vlan\_tag-%s, eth\_src - %s, Output : out\_port-%s" % (this\_dpid, vnf\_out\_port, vnf\_mac, dst\_mac, sla.vlanCommonTag | processed\_vnf\_cnt, src\_mac, out\_port))

self.add\_flow(datapath, priority, match, actions)

# Intermediate Switches

else:

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt),in\_port=in\_port,eth\_src=src\_mac,eth\_dst=dst\_mac)

actions = [parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

slaIndex += 1

# All VNF(s) have been processed

if processed\_vnf\_cnt == len(sla.VNFsNetworkDpid):

break

## 3. Actor - Last VNF and Intermediate Switches

# Re-direct the packet from Last VNF to the Egress Point (Actual Dest. Point)

last\_vnf\_mac = sla.VNFsNetworkMac[-1]

last\_vnf\_dpid = sla.VNFsNetworkDpid[-1]

## Case 3a. Destination VNF Hypervisor is also connected to the Egress Switch

if prev\_switch == dst\_dpid:

# Re-direct the packet from the VNF towards the destination

datapath = self.getDatapath(dst\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

same\_port = self.m\_mac\_to\_dpid\_port[dst\_mac].val1

match = parser.OFPMatch(in\_port=same\_port,eth\_src=last\_vnf\_mac,eth\_dst=dst\_mac)

actions = [parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionOutput(ofproto.OFPP\_IN\_PORT)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : eth\_src - %s, Output : out\_port-%s" % (dst\_dpid, same\_port, last\_vnf\_mac, dst\_mac, src\_mac, ofproto.OFPP\_IN\_PORT))

## Case 3b. Last VNF Hypervisor and Egress Switch are not same

else:

last\_vnf\_in\_port = self.m\_dpid\_to\_mac\_port[last\_vnf\_dpid][last\_vnf\_mac]

while path[slaIndex][1] != -1:

this\_dpid = path[slaIndex][0]

datapath = self.getDatapath(this\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

next\_switch = path[slaIndex + 1][0]

in\_port = last\_vnf\_in\_port if slaIndex == 0 else self.m\_graph['edges'][this\_dpid][prev\_switch]

out\_port = self.m\_graph['edges'][this\_dpid][next\_switch]

out\_port = ofproto.OFPP\_IN\_PORT if out\_port == in\_port else out\_port

# Re-format the packet from the Last VNF towards the actual destination

if this\_dpid == last\_vnf\_dpid:

match = parser.OFPMatch(in\_port=in\_port, eth\_src=last\_vnf\_mac, eth\_dst=dst\_mac)

# Add VLAN Tag for processing in the last Switch

actions = [parser.OFPActionPushVlan(0x8100),

parser.OFPActionSetField(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt)),

parser.OFPActionSetField(eth\_src=src\_mac),

parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | SetFeild : vlan\_tag-%s eth\_src - %s, Output : out\_port-%s" % (this\_dpid, in\_port, last\_vnf\_mac, dst\_mac, sla.vlanCommonTag | processed\_vnf\_cnt, src\_mac, out\_port))

# Intermediate Switches

else:

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

actions = [parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_src-%s eth\_dst-%s | Output : out\_port-%s" % (this\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, src\_mac, dst\_mac, out\_port))

prev\_switch = this\_dpid

slaIndex += 1

## Case 4. Actor - Egress Switch

if prev\_switch != dst\_dpid:

datapath = self.getDatapath(dst\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

in\_port = self.m\_graph['edges'][dst\_dpid][prev\_switch]

out\_port = self.m\_mac\_to\_dpid\_port[dst\_mac].val2

match = parser.OFPMatch(vlan\_vid=(sla.vlanCommonTag | processed\_vnf\_cnt), in\_port=in\_port, eth\_dst=dst\_mac)

actions = [parser.OFPActionPopVlan(0x8100),

parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, priority, match, actions)

LOG\_DEBUG("Datapath : %s | Match : vlan\_tag-%s in\_port-%s eth\_dst-%s | Output : out\_port-%s" % (dst\_dpid, sla.vlanCommonTag | processed\_vnf\_cnt, in\_port, dst\_mac, out\_port))

## 5. Actor - Ingress Switch

# Case 5a. Ingrees Switch and Centre Switch are not same

if len(path) > 2 and path[0][0] != path[1][0]:

datapath = self.getDatapath(src\_dpid)

parser = datapath.ofproto\_parser

ofproto = datapath.ofproto

next\_switch = path[1][0]

out\_port = self.m\_graph['edges'][src\_dpid][next\_switch]

match = parser.OFPMatch(in\_port=src\_in\_port, eth\_src=src\_mac, eth\_dst=dst\_mac)

# Add VLAN Tag for processing in next Switch(s)

actions = [parser.OFPActionPushVlan(0x8100),

parser.OFPActionSetField(vlan\_vid=sla.vlanCommonTag),

parser.OFPActionOutput(out\_port)]

if src\_in\_port == out\_port:

return

self.add\_flow(datapath, priority, match, actions) # Flow-Mod

LOG\_DEBUG("Datapath : %s | Match : in\_port-%s eth\_src-%s eth\_dst-%s | Output : vlan\_vid-%s out\_port-%s" % (src\_dpid, src\_in\_port, src\_mac, dst\_mac, sla.vlanCommonTag, out\_port))

# Merges Duplicate consecutive entries from the input List

# e.g. Input - [[2 1] [3 2] [3 1] [5 3]]

# Output - [[2 1] [3 3] [5 3]]

def mergeDuplicateEntries(self, inputList):

isUpdateInLastIteration = True

inList = copy.deepcopy(inputList)

outList = []

while isUpdateInLastIteration:

isUpdateInLastIteration = False

outList = []

outListLen = 0

for index in range(0, len(inList)):

if inList[index][1] != -1 and outList[outListLen-1][1] != -1 and outList[outListLen-1][0] == inList[index][0]:

outList[outListLen-1][1] += inList[index][1]

isUpdateInLastIteration = True

else:

outList.append(inList[index])

outListLen += 1

inList = copy.deepcopy(outList)

return outList

# Updates the Complete Path of SLA for all pairs of Hosts in the SLA

def updateCompPathOfSLA(self, sla, vnfId, nbr\_dpid):

vnfInfo = gUsedVNFIds[vnfId]

for src\_mac in sla.endUsersMac:

for dst\_mac in sla.endUsersMac:

# Sanity Check

if src\_mac == dst\_mac:

continue

if sla.compPathOfSLA[src\_mac][dst\_mac]['isInstalled'] == True:

sla.compPathOfSLA[src\_mac][dst\_mac]['prevPath'] = copy.deepcopy(sla.compPathOfSLA[src\_mac][dst\_mac]['currPath'])

else:

sla.compPathOfSLA[src\_mac][dst\_mac]['prevPath'] = []

path = copy.deepcopy(sla.compPathOfSLA[src\_mac][dst\_mac]['currPath'])

sla.compPathOfSLA[src\_mac][dst\_mac]['priority'] += 1

vnfCount = 0

for index in range(0, len(path)):

# Note the dpid(s) with the VNF

if path[index][1] != -1 and path[index][1] != 0:

vnfCount += 1

# Index starts from '0'

if vnfCount == vnfInfo.servChainIndex + 1:

path[index][1] -= 1

prev\_nbr\_dpid = path[index + 1][0]

path.insert(index + 1, [nbr\_dpid, 1])

# Three or more VNFs were installed

# e.g. Input: [[2 4]]

# Output: [[2 1] [nbr\_dpid 1] [2 2]]

if path[index][1] > 0:

count = path[index][1] - 1

path[index][1] = 1

path.insert(index + 2, [path[index][0], count])

prev\_nbr\_dpid = path[index][0]

index += 1

# Retreive Path between the two switches

intermPath = self.getSrcToDestPath(nbr\_dpid, prev\_nbr\_dpid)

if len(intermPath) > 2:

# First and Last elements are Src and Dst Dpids, do not consider

del intermPath[-1]

del intermPath[0]

for dpid in reversed(intermPath):

path.insert(index+1, [dpid, 0])

# Relax, and merging with Previous Dpid

# e.g. Input : [[1,1] [2,0] [3,0] [1,1]]

# Output : [[1,2]]

tempIndex = index - 1

while tempIndex != 0 and path[tempIndex][1] == 0 and path[tempIndex][0] != path[index][0]:

tempIndex -= 1

# Same dpid having intermediate switches

if tempIndex != 0 and path[tempIndex][0] == path[index][0]:

path[index][1] += path[tempIndex][1]

del path[tempIndex : index]

elif tempIndex == 0 and path[tempIndex][0] == path[index][0]:

del path[1 : index]

# Club duplicate entries, if exists

path = self.mergeDuplicateEntries(path)

sla.compPathOfSLA[src\_mac][dst\_mac]['currPath'] = copy.deepcopy(path)

LOG\_DEBUG("For Hosts %s -> %s: " % (src\_mac, dst\_mac))

LOG\_DEBUG("Previous Path: %s" % sla.compPathOfSLA[src\_mac][dst\_mac]['prevPath'])

LOG\_DEBUG("Current Path: %s" % sla.compPathOfSLA[src\_mac][dst\_mac]['currPath'])

break

# Handles migration of VNF from Current Hypervisor to Nbr Hypervisor

# Scope : Completes all other operations for Migration

def migrateVNF\_RestOp(self, savedContext, new\_dpid):

#pdb.set\_trace()

old\_vnfId = savedContext['old\_vnfId']

new\_vnfId = savedContext['new\_vnfId']

old\_dpid = savedContext['old\_dpid']

oldVNFInfo = gUsedVNFIds[old\_vnfId]

newVNFInfo = gUsedVNFIds[new\_vnfId]

vnfIndexOfServiceChain = newVNFInfo.servChainIndex

sla = self.m\_SLAs[oldVNFInfo.slaIdentifier]

# Update Complete Path of SLA

self.updateCompPathOfSLA(sla, new\_vnfId, new\_dpid)

## Step 3: Update VNF Information w.r.t. New Dpid in all related data-structures

sla.VNFsNetworkDpid[vnfIndexOfServiceChain] = new\_dpid

newVNFInfo.dpid = new\_dpid

del self.m\_mac\_to\_dpid\_port[oldVNFInfo.macAddr]

del self.m\_dpid\_to\_mac\_port[old\_dpid][oldVNFInfo.macAddr]

self.m\_hypVNFStatus[old\_dpid].remove(oldVNFInfo.identifier)

# Map VNFInfo to New Hypervisor

self.m\_hypVNFStatus[newVNFInfo.dpid].append(newVNFInfo.identifier)

self.m\_hypervisorMemStats[new\_dpid]['used'] += newVNFInfo.cpuMemReq

self.m\_hypervisorMemStats[old\_dpid]['used'] -= newVNFInfo.cpuMemReq

## Step 4 : Add New flow rules for this change with Higher Priority

self.updateNewSLAFlowRules(sla, True, [], [])

LOG\_DEBUG("Installed New SLA Flow Rules for the SLA: %s" % sla.identifier)

## Step 5: Stop VNF at the current Hypervisor

# Retrieve Old Hypervisor IP and sockfd

hypIPAddr = self.m\_dpid\_to\_hyp\_ip[old\_dpid]

oldSockfd = ""

for pair in self.m\_hyp\_ip\_sockfd\_pair:

if hypIPAddr == pair.val1[0]:

oldSockfd = pair.val2

break

# Sanity Check

if oldSockfd == "":

LOG\_DEBUG("System communication with Old Hypervisor(%s) is broken. Migration Failed." % (new\_dpid.dpid))

return FAILURE

# Stops VNF at the Old Hypervisor

self.sendStopVNFCommand(oldVNFInfo, oldSockfd)

## Step 6: Remove Previous Flow Rules for all paths

# May also be updated when PacketIn for First Time

for src\_mac in sla.endUsersMac:

for dst\_mac in sla.endUsersMac:

# Sanity Check

if src\_mac == dst\_mac:

continue

if sla.compPathOfSLA[src\_mac][dst\_mac]['prevPath'] != []:

#pdb.set\_trace()

self.removeOldSLAFlowRules(sla, src\_mac, dst\_mac, sla.compPathOfSLA[src\_mac][dst\_mac]['prevPath'])

sla.compPathOfSLA[src\_mac][dst\_mac]['prevPath'] = []

# TODO: Step 7: Update Delay Buffer for the SLA

# Handles migration of VNF from Current Hypervisor to Nbr Hypervisor

# Scope limited to Updating Complete Paths and starting VNF at Nbr Hypervisor

def migrateVNF\_VNFStartOp(self, dpid, new\_dpid, vnfId):

oldVNFInfo = gUsedVNFIds[vnfId]

vnfIndexOfServiceChain = oldVNFInfo.servChainIndex

sla = self.m\_SLAs[oldVNFInfo.slaIdentifier]

newVNFInfo = self.assignVNFResources(sla, oldVNFInfo.servChainIndex, dpid, True)

old\_dpid = sla.VNFsNetworkDpid[vnfIndexOfServiceChain]

# Retrieve New Hypervisor IP and sockfd

hypIPAddr = self.m\_dpid\_to\_hyp\_ip[new\_dpid]

newSockfd = ""

for pair in self.m\_hyp\_ip\_sockfd\_pair:

if hypIPAddr == pair.val1[0]:

newSockfd = pair.val2

break

# Sanity Check

if newSockfd == "":

LOG\_DEBUG("Incorrect Migration at Hypervisor (%s) or System communication with Hypervisor(%s) is broken." % (new\_dpid.dpid, new\_dpid.dpid))

return FAILURE

## Step 1 : Start VNF at the Nbr Hypervisor with same configuration

# Starts VNF at the Hypervisor

self.sendStartVNFCommand(newVNFInfo, newSockfd)

# Rest of the Actions to be taken after VNF comes up at the New Hypervisor

# Ref: migrateVNF\_RestOp

## Step 2a : Save current context

# Sanity Check

if newVNFInfo.macAddr in self.m\_vnfOperationsOnStart:

LOG\_DEBUG("This scenario should not occur. Highly impossible!!!")

return

self.m\_vnfOperationsOnStart[newVNFInfo.macAddr] = {}

self.m\_vnfOperationsOnStart[newVNFInfo.macAddr]['old\_dpid'] = old\_dpid

self.m\_vnfOperationsOnStart[newVNFInfo.macAddr]['old\_vnfId'] = oldVNFInfo.identifier

self.m\_vnfOperationsOnStart[newVNFInfo.macAddr]['new\_vnfId'] = newVNFInfo.identifier

# Handle Node Bottleneck

def handleNodeBottleneck(self, dpid):

priorityScores = {}

# All VNFs installed at the Bottlenecked Hypervisor

for vnfIndex in self.m\_hypVNFStatus[dpid]:

vnfInfo = gUsedVNFIds[vnfIndex]

LOG\_DEBUG("Hypervisor (%s) : VNF (c%s) considered for Migration Algorithm (Case 1)." % (dpid, vnfInfo.identifier))

# Consider all Nbrs of this dpid

# Retrieve Nbrs of the current Dpid

for nbr\_dpid,out\_port in self.m\_graph['edges'][dpid].items():

score,result = self.generatePriorityScore(vnfInfo, dpid, nbr\_dpid)

if result == SUCCESS:

priorityScores[make\_tuple(vnfIndex, nbr\_dpid)] = score

maxPriorityScore = 0.0

maxPSPair = []

for pair in priorityScores:

LOG\_DEBUG("Moving VNF (c%s) from Current (%s) to Nbr (%s) Hypervisor : %s." % (vnfInfo.identifier, dpid, pair.val2, priorityScores[pair]))

# TODO: BCase

if priorityScores[pair] > maxPriorityScore:

maxPriorityScore = priorityScores[pair]

maxPSPair = pair

# Movement to the Nbrs not possible, leads to Case 2

if maxPriorityScore == 0.0:

LOG\_DEBUG("This scenario should not occur (NOW). Programming Error!!!")

self.handleNodeBottleneckCase2()

# Handles migration of VNF from Current Hypervisor to Nbr Hypervisor

self.migrateVNF\_VNFStartOp(dpid, maxPSPair.val2, maxPSPair.val1)

# -----------------------------------------

# Miscellaneous Functions

# -----------------------------------------

# Reads Hypervisor Configuration File

def read\_HYP\_config\_file(self):

dpid\_ip\_pair = {}

file = open(HYPERVISOR\_CONFIGURATION\_FILE, "r")

# Retreive # No.of nodes

nodes = int(file.readline().rstrip())

for i in range(nodes):

line = file.readline().strip()

# Retreive dpid and Hypervisor IP

words = line.split()

dpid\_ip\_pair[int(words[0])] = words[1].strip("\n")

return dpid\_ip\_pair

def LOG\_DEBUG(string):

print str(string) + " - " + str(currentframe().f\_back.f\_lineno)

class make\_tuple():

def \_\_init\_\_(self, value1, value2, value3=None):

self.val1 = value1

self.val2 = value2

self.val3 = value3

def print\_flowpath(path):

print "----------------------------"

print path

x = ""

for item in path:

x += " -> %s" % item

LOG\_DEBUG("Flowpath: %s" % x)

print "----------------------------"

def reverse\_flowpath(path):

retList = []

for item in range(len(path)-1, -1, -1):

retList.append(path[item])

return retList