**ELGY 9333**

**DCN Lab4: SDN Open Virtual Switches**

**Due: 4/9/2016**

Submitted by:

V.Deepti Reddy (N15361877)

Deepak Kadimela (N11094172)

Reports:

Q1) Write a pseudo code to implement spanning tree in SDN network.

Step 1: Let Spanning Tree T defined to be an empty set

Step 2: For each vertex v of Graph G, make the empty set out of v

Step 3: Sort the (u,v) edges of G in the ascending order

Step 4: For each edge (u, v) from the sorted list of Step 3

1. If u and v belong to different sets, then add (u,v) to T
2. Get together u and v in one single set

Step 5: Return T

Reference: <http://www.stoimen.com/blog/2012/11/12/computer-algorithms-kruskals-minimum-spanning-tree/>

Q2) List the advantages of using OpenVSwitch and SDN controller compared to IP networks.

**Advantages of OpenVSwitch:**

1. OpenVSwitch supports both configuring and migrating : slow (configuration) and fast network state between instances.
2. OpenVSwitch state is typed and backed by a real data-model and hence allows for the development of structured automation systems.
3. OpenVSwitch is designed such that managing VM network configuration and monitor state spread across many physical hosts in dynamic virtualized environments.
4. It supports a number of features that allow a network control system to respond and adapt as the environment changes including simple accounting and visibility support like NetFlow.
5. OpenVSwitch includes multiple methods for specifying and maintaining tagging rules, all of which are accessible to a remote process for orchestration.

Advantages of SDN :

1. Software-defined networking is an architecture which is dynamic, manageable, cost- effective, and adaptable, seeking to be suitable for the high-bandwidth, dynamic nature of today's applications.
2. Network control is directly programmable because it is decoupled from forwarding functions.
3. Agile : Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
4. Centrally managed : Network intelligence is logically centralized in SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
5. Programmatically configured : SDN lets network managers configure , manage, secure and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend

**Q3) Controllers Code**

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from ryu.controller.handler import CONFIG\_DISPATCHER, MAIN\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.ofproto import ofproto\_v1\_3

from ryu.ofproto import ether

from ryu.ofproto import inet

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet

from ryu.lib.packet import arp

from ryu.lib.packet import ipv4

from ryu.lib.packet import tcp

from ryu.lib.packet import udp

class SimpleSwitch13(app\_manager.RyuApp):

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

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

super(SimpleSwitch13, self).\_\_init\_\_(\*args, \*\*kwargs)

self.mac\_to\_port = {}

self.arp\_table = {}

@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

match = parser.OFPMatch()

actions = [parser.OFPActionOutput(ofproto.OFPP\_CONTROLLER,

ofproto.OFPCML\_NO\_BUFFER)]

self.add\_flow(datapath, 0, match, actions)

self.arp\_table["10.0.0.1"] = "00:00:00:00:00:01"

self.arp\_table["10.0.0.2"] = "00:00:00:00:00:02"

self.arp\_table["10.0.0.3"] = "00:00:00:00:00:03"

self.arp\_table["10.0.0.4"] = "00:00:00:00:00:04"

if datapath.id == 3:

print "Adding flows to switch 3"

priority =5

self.add\_flowss(datapath,priority,inet.IPPROTO\_UDP,1,2)

self.add\_layer4\_rules(datapath,priority+1,inet.IPPROTO\_UDP,"10.0.0.2","10.0.0.3",None)

if datapath.id == 1:

print "Adding flows to switch 1"

priority =5

self.add\_layer4\_rules(datapath,priority,None,None,"10.0.0.1",3)

self.add\_layer4\_rules(datapath,priority,None,None,"10.0.0.3",4)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_ICMP,None,"10.0.0.2",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_ICMP,None,"10.0.0.4",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_TCP,None,"10.0.0.2",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_TCP,None,"10.0.0.4",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_UDP,None,"10.0.0.2",1)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_UDP,None,"10.0.0.4",1)

match = parser.OFPMatch(eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = inet.IPPROTO\_TCP,

ipv4\_src ="10.0.0.1",

ipv4\_dst = "10.0.0.3")

actions = [parser.OFPActionOutput(ofproto.OFPP\_CONTROLLER,

ofproto.OFPCML\_NO\_BUFFER)]

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

if datapath.id == 2:

print "Adding flows to switch 2"

priority = 5

self.add\_layer4\_rules(datapath,priority,None,None,"10.0.0.2",3)

self.add\_layer4\_rules(datapath,priority,None,None,"10.0.0.4",4)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_ICMP,None,"10.0.0.1",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_ICMP,None,"10.0.0.3",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_TCP,None,"10.0.0.1",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_TCP,None,"10.0.0.3",2)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_UDP,None,"10.0.0.1",1)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_UDP,None,"10.0.0.3",1)

def add\_layer4\_rules(self,datapath,priority=1,ip\_proto=None,ipv4\_src=None,ipv4\_dst=None,fwd\_port=None):

if ip\_proto is None:

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_ICMP,ipv4\_src,ipv4\_dst,fwd\_port)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_TCP,ipv4\_src,ipv4\_dst,fwd\_port)

self.add\_layer4\_rules(datapath,priority,inet.IPPROTO\_UDP,ipv4\_src,ipv4\_dst,fwd\_port)

return

parser = datapath.ofproto\_parser

if ipv4\_src is None and ipv4\_dst is None:

match = parser.OFPMatch(eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = ip\_proto)

elif ipv4\_src is None:

match = parser.OFPMatch(eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = ip\_proto,

ipv4\_dst =ipv4\_dst)

elif ipv4\_dst is None:

match = parser.OFPMatch(eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = ip\_proto,

ipv4\_src =ipv4\_src)

else:

match = parser.OFPMatch(eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = ip\_proto,

ipv4\_src =ipv4\_src,

ipv4\_dst = ipv4\_dst)

if fwd\_port is None:

actions = []

else:

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

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

def add\_flowss(self,datapath,priority,protocol,port1,port2):

parser = datapath.ofproto\_parser

match = parser.OFPMatch(in\_port=port1,

eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = inet.IPPROTO\_UDP)

actions = [parser.OFPActionOutput(port2)]

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

match = parser.OFPMatch(in\_port=port2,

eth\_type = ether.ETH\_TYPE\_IP,

ip\_proto = inet.IPPROTO\_UDP)

actions = [parser.OFPActionOutput(port1)]

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

def add\_flow(self, datapath, priority, match, actions):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS,

actions)]

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

match=match, instructions=inst)

datapath.send\_msg(mod)

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

def \_packet\_in\_handler(self, ev):

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

in\_port = msg.match['in\_port']

#Reference how to identify the packet's protocols

pkt = packet.Packet(msg.data)

eth = pkt.get\_protocols(ethernet.ethernet)[0]

eth\_type = eth.ethertype

# process ARP

if eth\_type == ether.ETH\_TYPE\_ARP:

self.handle\_arp(datapath, in\_port, pkt)

return

elif eth\_type == ether.ETH\_TYPE\_IP:

self.handle\_ip(datapath, in\_port, pkt)

return

else:

return

def handle\_arp(self,datapath,in\_port,pkt):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

eth\_pkt = pkt.get\_protocol(ethernet.ethernet)

arp\_pkt = pkt.get\_protocol(arp.arp)

arp\_resolv\_mac = self.arp\_table[arp\_pkt.dst\_ip]

new\_packet = packet.Packet()

new\_packet.add\_protocol(ethernet.ethernet(ethertype=eth\_pkt.ethertype,

dst=eth\_pkt.src,

src=arp\_resolv\_mac))

new\_packet.add\_protocol(arp.arp(opcode=arp.ARP\_REPLY,

src\_mac=arp\_resolv\_mac,

src\_ip=arp\_pkt.dst\_ip,

dst\_mac=arp\_pkt.src\_mac,

dst\_ip=arp\_pkt.src\_ip))

new\_packet.serialize()

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

out = parser.OFPPacketOut(datapath,

ofproto.OFP\_NO\_BUFFER,

ofproto.OFPP\_CONTROLLER,

actions,

new\_packet.data)

datapath.send\_msg(out)

def handle\_ip(self,datapath,in\_port,pkt):

ofproto=datapath.ofproto

parser = datapath.ofproto\_parser

eth\_pkt = pkt.get\_protocol(ethernet.ethernet)

ipv4\_pkt = pkt.get\_protocol(ipv4.ipv4)

if datapath.id == 1 and ipv4\_pkt.proto==inet.IPPROTO\_TCP:

tcp\_pkt = pkt.get\_protocol(tcp.tcp)

new\_packet = packet.Packet()

new\_packet.add\_protocol(ethernet.ethernet(ethertype=eth\_pkt.ethertype,

dst=eth\_pkt.src,

src=eth\_pkt.dst))

new\_packet.add\_protocol(ipv4.ipv4(dst=ipv4\_pkt.src,

src=ipv4\_pkt.dst,

proto=ipv4\_pkt.proto))

new\_packet.add\_protocol(tcp.tcp(src\_port=tcp\_pkt.dst\_port,

dst\_port=tcp\_pkt.src\_port,

ack=tcp\_pkt.seq+1,

bits=20))

new\_packet.serialize()

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

out = parser.OFPPacketOut(datapath,

ofproto.OFP\_NO\_BUFFER,

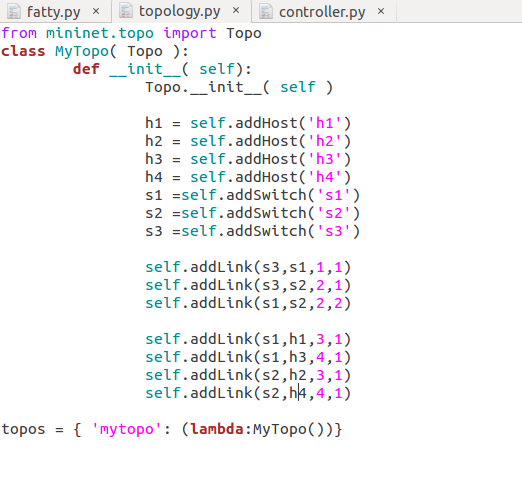
ofproto.OFPP\_CONTROLLER,

actions,

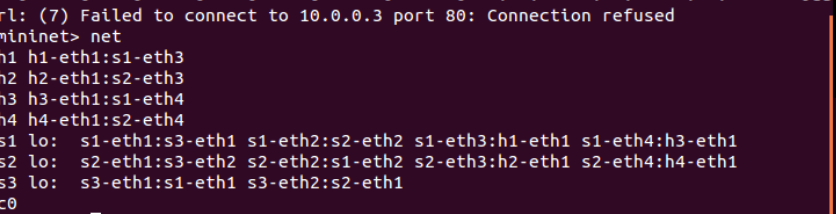
new\_packet.data)

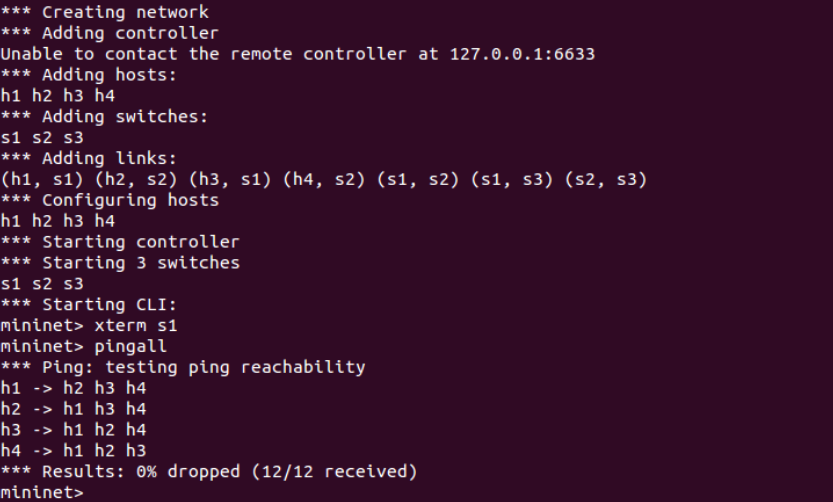
datapath.send\_msg(out)

Q4) Topology file:

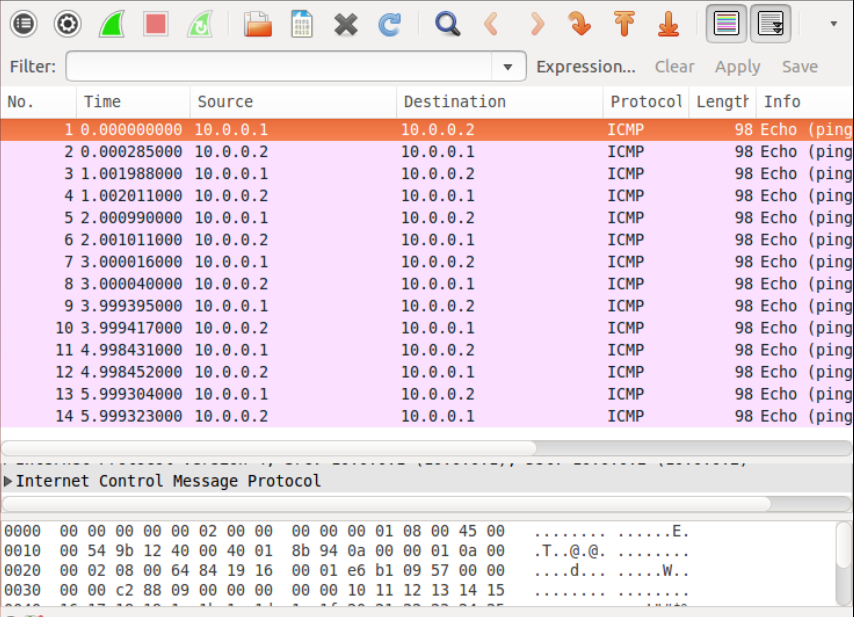


Q5) Screenshots: first, creating networks



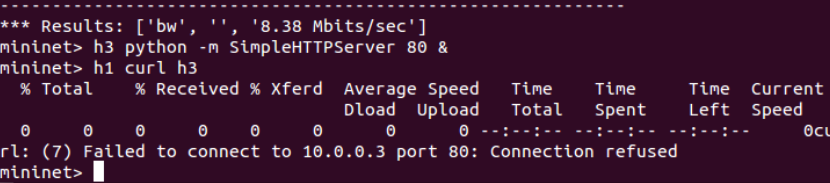
Case1: pingall

Case 2: TCP, UDP and ICMP packets on their respective paths

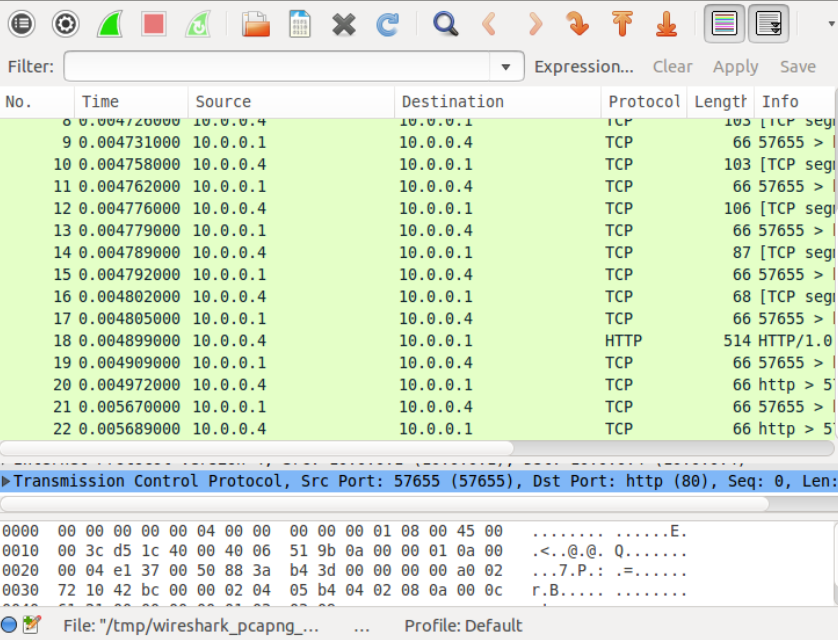
ICMP Packet: take short path via s2

*2. TCP Packets:*

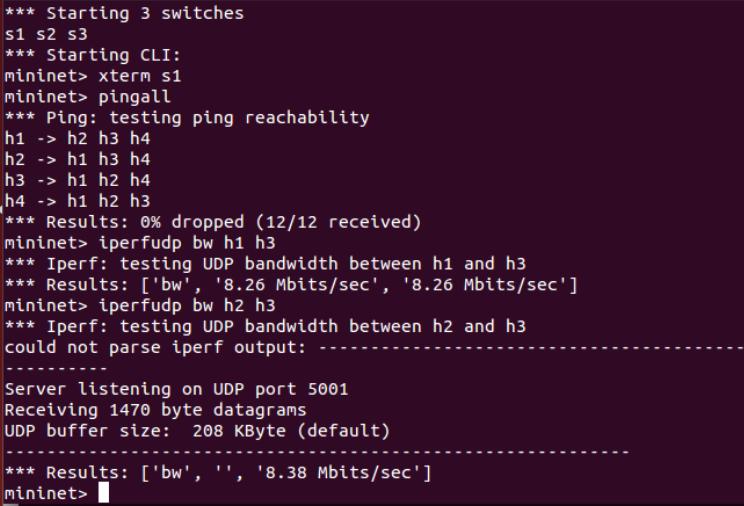
When a TCP connection is initiated by h1 to h3, the connection is refused and the packet is dropped with a RST

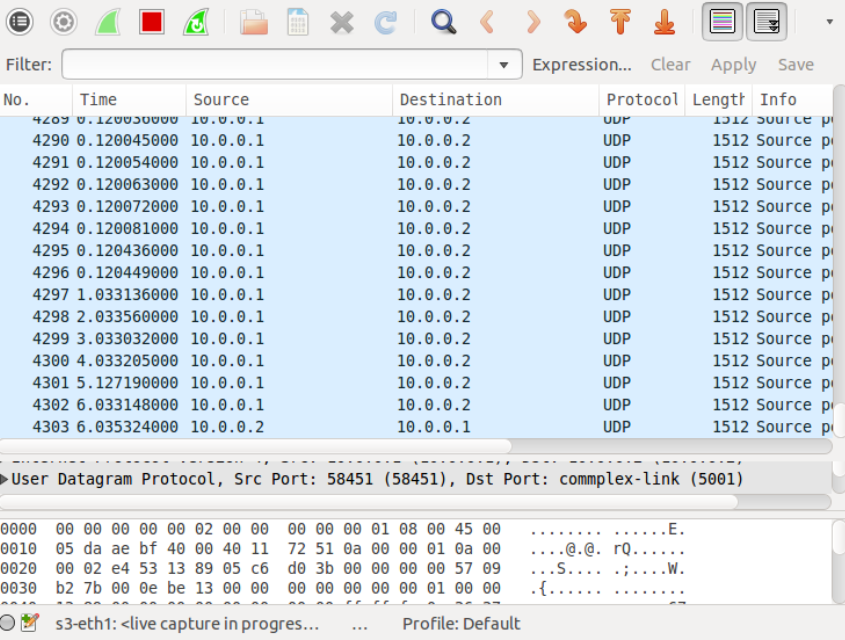


TCP packets taking short path via s2



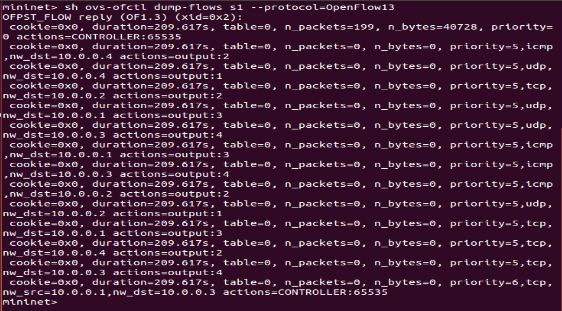
***3. UDP Packets:*** UDP packets taking long path via s3



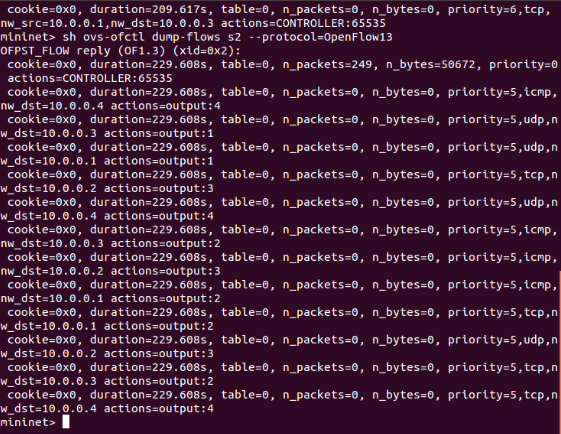


***Case 3: Rules Installed at Each Switch:***

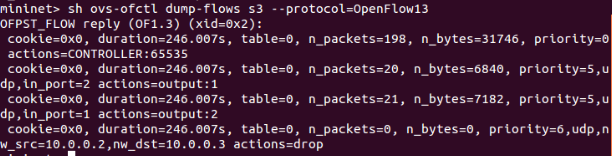
***For Switch1:***



For switch S2:



For switch s3:



Q6) challenges encountered:

1.The primarily issue and difficulty was learning how to program a controller. I went online, researched, looked at a few forums and think how Ryu worked to figure out how to do it.

2. few Syntax errors in l4\_switch-1.py.I think they were Indentation errors, also they were easily resolved by correcting the indentation in the code.