# Sdn网络实验2

目录

[Sdn网络实验2 1](#_Toc131679367)

[自学习交换机 1](#_Toc131679368)

[问题分析 1](#_Toc131679369)

[代码实现 3](#_Toc131679370)

[处理环路广播、 6](#_Toc131679371)

[问题分析 6](#_Toc131679372)

[代码实现 8](#_Toc131679373)

[附加题 11](#_Toc131679374)

[问题分析 11](#_Toc131679375)

[代码实现 11](#_Toc131679376)

[题外问题 13](#_Toc131679377)

## 自学习交换机

在第一个实验中，我们要使用ryu实现一个自学习交换机。

### 问题分析

在本次实验中，给出了部分ryu的代码，其中缺失了部分处理洪泛的代码。其代码如下：

from ryu.base import app\_manager

from ryu.controller import ofp\_event

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

class Switch(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]def \_\_init\_\_(self, \*args, \*\*kwargs):

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

# maybe you need a global data structure to save the mapping

def add\_flow(self, datapath, priority, match,

actions,idle\_timeout=0,hard\_timeout=0):

dp = datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

inst = [parser.OFPInstructionActions(ofp.OFPIT\_APPLY\_ACTIONS, actions)]

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

idle\_timeout=idle\_timeout,

hard\_timeout=hard\_timeout,

match=match,instructions=inst)

dp.send\_msg(mod)

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

match = parser.OFPMatch()

actions =

[parser.OFPActionOutput(ofp.OFPP\_CONTROLLER,ofp.OFPCML\_NO\_BUFFER)]

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

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

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

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

# the identity of switch

dpid = dp.id

self.mac\_to\_port.setdefault(dpid,{})

# the port that receive the packet

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

pkt = packet.Packet(msg.data)

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

# get the mac

dst = eth\_pkt.dst

src = eth\_pkt.src

# we can use the logger to print some useful information

self.logger.info('packet: %s %s %s %s', dpid, src, dst, in\_port)

# you need to code here to avoid the direct flooding

# having fun

# :)

通过观察，我们可以发现代码中：1.需要一个可以保存各个端口对应设备MAC地址的列表来使得在转发时交换机可以精确将包转发给需要的设备而不传给其他设备。

2.需要在转发时判断是否为洪泛来给交换机传递不同的信息。

在代码实现与分析过程中，我参考并使用了以下网站的一部分信息/代码：

1. https://osrg.github.io/ryu-book/zh\_tw/html/switching\_hub.html#ryu （ryu book中文版）
2. <https://www.cnblogs.com/csw5719/p/15412604.html>（原码注释与翻译）

### 代码实现

#此代码与注释源自上列第一、二个网站的修改

from ryu.base import app\_manager

from ryu.controller import ofp\_event

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

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

from ryu.ofproto import ofproto\_v1\_3

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet

from ryu.lib.packet import ether\_types

# 继承ryu.base.app\_manager.RyuApp

class SimpleSwitch13(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION] # 指定OpenFlow 1.3版本

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

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

self.mac\_to\_port = {} # 定义MAC地址列表

# set\_ev\_cls指定事件类别得以接受消息和交换机状态作为参数

# 其中事件类别名称为ryu.controller.ofp\_event.EventOFP+<OpenFlow消息名称>

# 例如：在 Packet-In 消息的状态下的事件名称为EventOFPPacketIn

# 对于交换机的状态来说，可指定以下中的一项

# ryu.controller.handler.HANDSHAKE\_DISPATCHER 交换 HELLO 消息

# ryu.controller.handler.CONFIG\_DISPATCHER 接收SwitchFeatures消息

# ryu.controller.handler.MAIN\_DISPATCHER 一般状态

# ryu.controller.handler.DEAD\_DISPATCHER 连线中断

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

# ev.msg 是用来存储对应事件的 OpenFlow 消息类别实体

# msg.datapath是用来存储OpenFlow交换机的 ryu.controller.controller.Datapath 类别所对应的实体

datapath = ev.msg.datapath

ofproto = datapath.ofproto # ofproto表示使用的OpenFlow版本所对应的ryu.ofproto.ofproto\_v1\_3

parser = datapath.ofproto\_parser # 和ofproto一样，有对应版本ryu.ofproto.ofproto\_v1\_3\_parser

# 下发table-miss流表项，让交换机对于不会处理的数据包通过packet-in消息上交给Ryu控制器

# 匹配数据包

# 若数据包没有 match 任何一个普通 Flow Entry 时，则触发 Packet-In

match = parser.OFPMatch()

# 通过预留端口ofproto.OFPP\_CONTROLLER，将packet-in消息发送给controller，并通过ofproto.OFPCML\_NO\_BUFFE指明Racket-in消息的原因是table miss

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

ofproto.OFPCML\_NO\_BUFFER)]

# 执行 add\_flow() 方法以发送 Flow Mod 消息

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

def add\_flow(self, datapath, priority, match, actions, buffer\_id=None):

# 新增流表项

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

# Apply Actions 是用来设定那些必须立即执行的 action 所使用

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

actions)]

# 通过 Flow Mod 消息将 Flow Entry 新增到 Flow table 中

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)

@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)

# 为了接收处理未知目的地的数据包，需要执行Packet-In 事件管理

msg = ev.msg # 每一个事件类ev中都有msg成员，用于携带触发事件的数据包

datapath = msg.datapath # 已经格式化的msg其实就是一个packet\_in报文，msg.datapath直接可以获得packet\_in报文的datapath结构

# datapath用于描述一个交换网桥，也是和控制器通信的实体单元。

# datapath.send\_msg()函数用于发送数据到指定datapath。

# 通过datapath.id可获得dpid数据。

ofproto = datapath.ofproto # datapath.ofproto对象是一个OpenFlow协议数据结构的对象，成员包含OpenFlow协议的数据结构，如动作类型OFPP\_FLOOD

parser = datapath.ofproto\_parser # datapath.ofp\_parser则是一个按照OpenFlow解析的数据结构。

# 更新Mac地址表

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

pkt = packet.Packet(msg.data)

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

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

# ignore lldp packet

return

dst = eth.dst

src = eth.src

dpid = datapath.id

self.mac\_to\_port.setdefault(dpid, {})

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

# learn a mac address to avoid FLOOD next time.

self.mac\_to\_port[dpid][src] = in\_port

# 判断转发的数据包的连接端口

# 目的 MAC 位址若存在于 MAC 地址表，则判断该连接端口号码为输出。

# 反之若不存在于 MAC 地址表则 OUTPUT action 类别的实体并生成 flooding（ OFPP\_FLOOD ）给目的连接端口使用。

if dst in self.mac\_to\_port[dpid]:

out\_port = self.mac\_to\_port[dpid][dst]

else:

out\_port = ofproto.OFPP\_FLOOD

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

# install a flow to avoid packet\_in next time

if out\_port != ofproto.OFPP\_FLOOD:

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

# 检测是否为洪泛

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

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

return

else:

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

# 在 MAC 地址表中找寻目的 MAC 地址，若是有找到则发送 Packet-Out 讯息，并且转送数据包。

data = None

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

data = msg.data

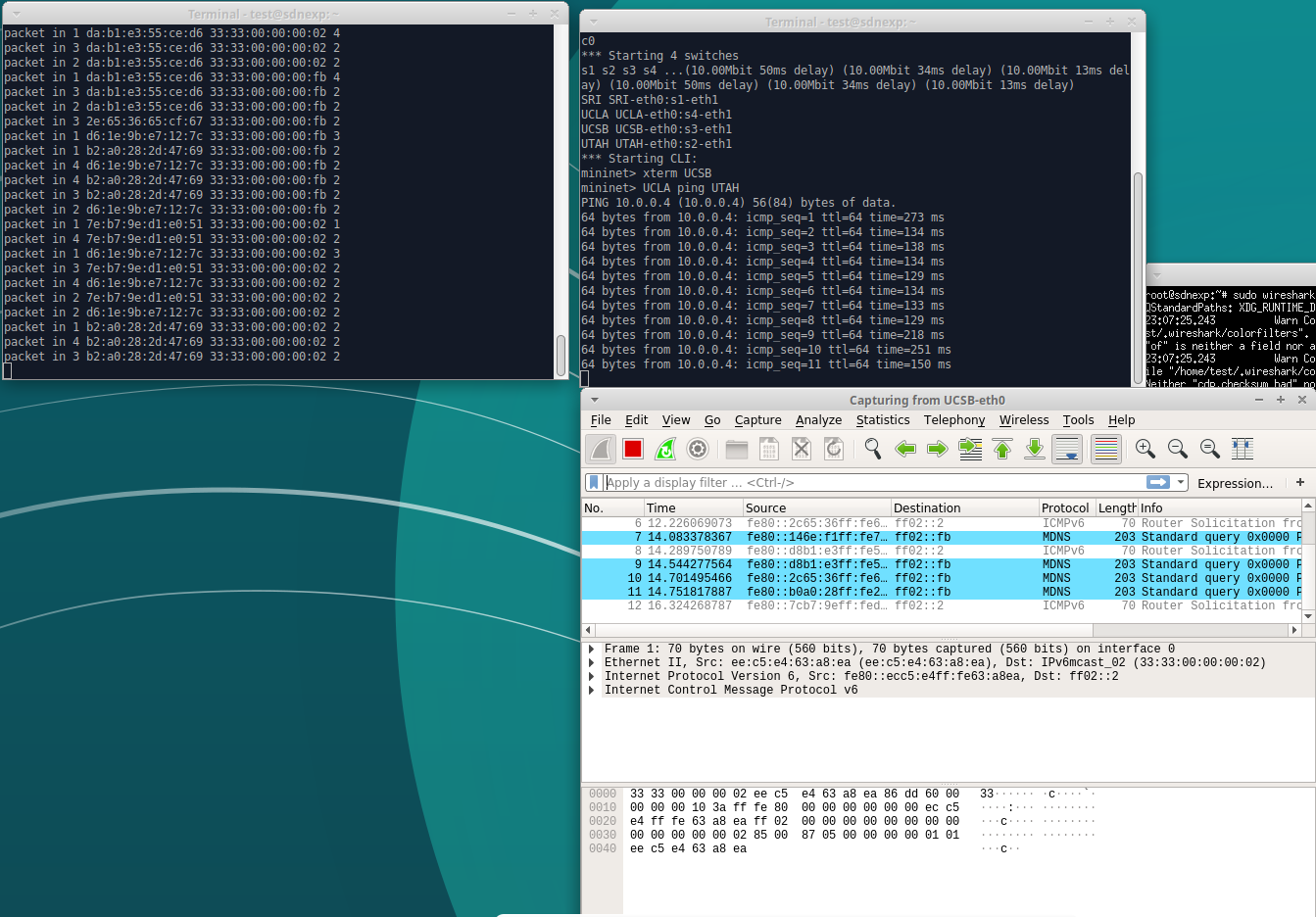
out = parser.OFPPacketOut(datapath=datapath, buffer\_id=msg.buffer\_id,

in\_port=in\_port, actions=actions, data=data)#发送信息

datapath.send\_msg(out)#向交换机发送信息

在上述代码中，实现了基于ryu的二层交换机。

实验结果：可以看到在UCLA ping UTAH的时候UCSB没有收到数据包。



## 处理环路广播、

### 问题分析

通过截图1可以看到确实出现了数据环路问题，包的数量异常的多。

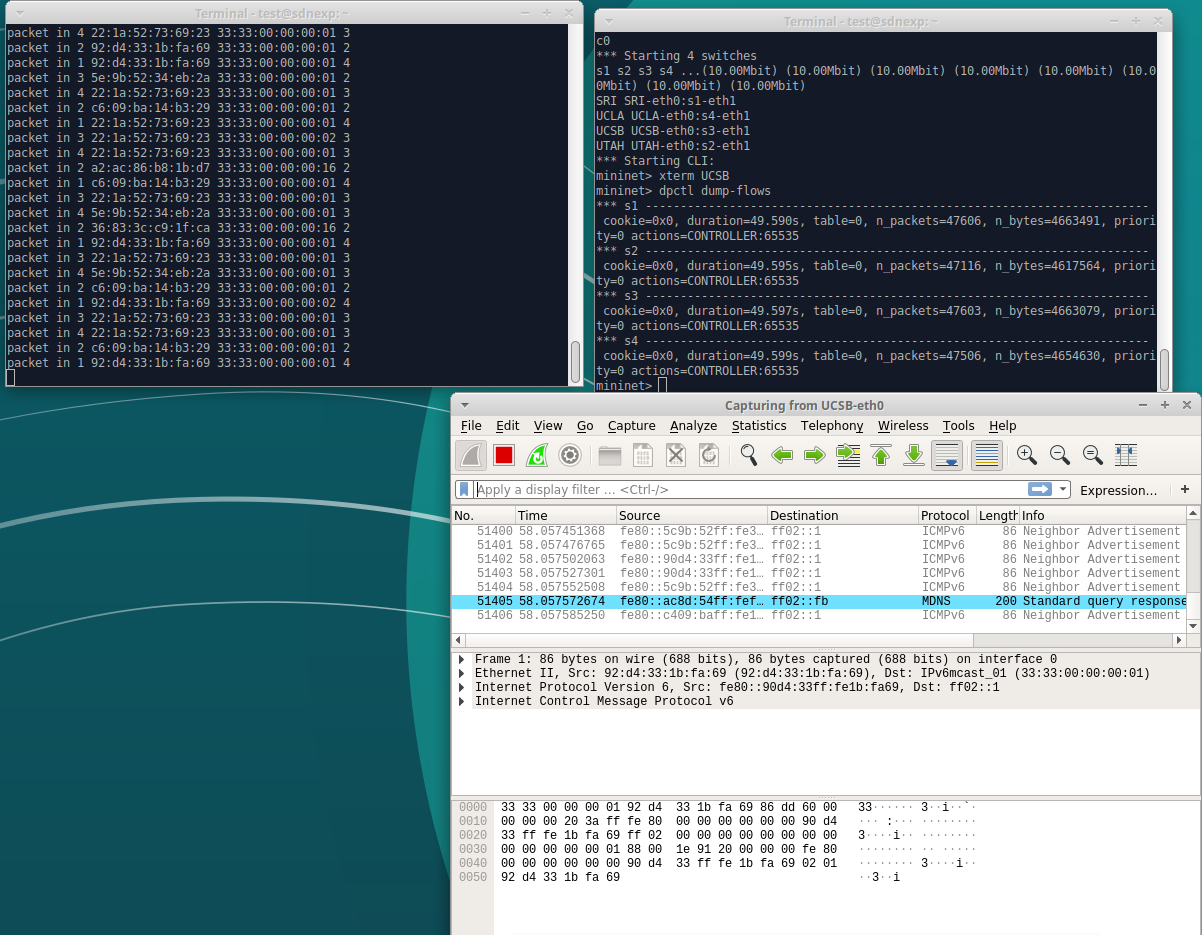
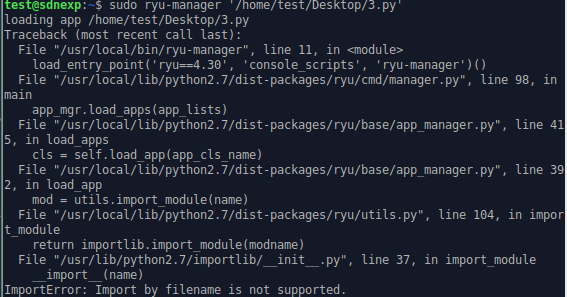
故需对此进行处理。

本次题目中在判断ARP是否属于包中协议处给出的代码框架在给出的实验环境中执行时会报错，需将判决条件改为：

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

if pkt\_arp:

错误如截图2所示：

截图1

截图2

该框架除修正以上错误外，还应添加（源mac地址，目的ip地址，交换机）的字典以防止arp协议的持续广播。ip地址可从arp.dst\_ip中得到。

应注意字典的成员添加为双层的，详情见下列代码。

若sw字典中有成员与包中元素匹配，则丢弃（return），否则添加进字典。

在代码中应注意先判断包是否存在，再去尝试提取包中的成员。字典也一样。

必须注意在丢弃arp数据包时，应使原in\_port接入口的包通过。

### 代码实现

代码实现如下：

from ryu.base import app\_manager

from ryu.controller import ofp\_event

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

from ryu.lib.packet import arp

from ryu.lib.packet import ether\_types

from ryu.lib.packet import ipv4

ETHERNET = ethernet.ethernet.\_\_name\_\_

ETHERNET\_MULTICAST = "ff:ff:ff:ff:ff:ff"

ARP = arp.arp.\_\_name\_\_

class Switch\_Dict(app\_manager.RyuApp):

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

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

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

self.mac\_to\_port = {}

self.sw = {} #(dpid, src\_mac, dst\_ip)=>in\_port, you may use it in mission 2

# maybe you need a global data structure to save the mapping

# just data structure in mission 1

def add\_flow(self, datapath, priority, match, actions, idle\_timeout=0, hard\_timeout=0):

dp = datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

inst = [parser.OFPInstructionActions(ofp.OFPIT\_APPLY\_ACTIONS, actions)]

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

idle\_timeout=idle\_timeout,

hard\_timeout=hard\_timeout,

match=match, instructions=inst)

dp.send\_msg(mod)

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

match = parser.OFPMatch()

actions = [parser.OFPActionOutput(ofp.OFPP\_CONTROLLER, ofp.OFPCML\_NO\_BUFFER)]

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

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

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

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

# the identity of switch

dpid = dp.id

self.mac\_to\_port.setdefault(dpid, {})

self.sw.setdefault(dpid, {})

# the port that receive the packet

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

pkt = packet.Packet(msg.data)

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

if eth\_pkt.ethertype == ether\_types.ETH\_TYPE\_LLDP:

return

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

return

# get the mac

dst = eth\_pkt.dst

src = eth\_pkt.src

self.sw[dpid].setdefault(src, {})#此处即为字典的嵌套

# get protocols

header\_list = dict((p.protocol\_name, p) for p in pkt.protocols if type(p) != str)

p\_arp=pkt.get\_protocol(arp.arp)

if p\_arp:

# you need to code here to avoid broadcast loop to finish mission 2

if p\_arp.dst\_ip in self.sw[dpid][src] and p\_arp.opcode==arp.ARP\_REQUEST:

if self.sw[dpid][src][p\_arp.dst\_ip]!=in\_port:

return

else:

if p\_arp.opcode==arp.ARP\_REQUEST:

self.sw[dpid][src][p\_arp.dst\_ip]=in\_port

self.logger.info("add %s %s %s %s", dpid, src, p\_arp.dst\_ip,p\_arp.opcode)

# self-learning

# you need to code here to avoid the direct flooding

# having fun

# :)

# just code in mission 1

# learn a mac address to avoid FLOOD next time.

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

self.mac\_to\_port[dpid][src] = in\_port

if dst in self.mac\_to\_port[dpid]:

out\_port = self.mac\_to\_port[dpid][dst]

else:

out\_port = ofp.OFPP\_FLOOD

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

# install a flow to avoid packet\_in next time

if out\_port != ofp.OFPP\_FLOOD:

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

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

data = None

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

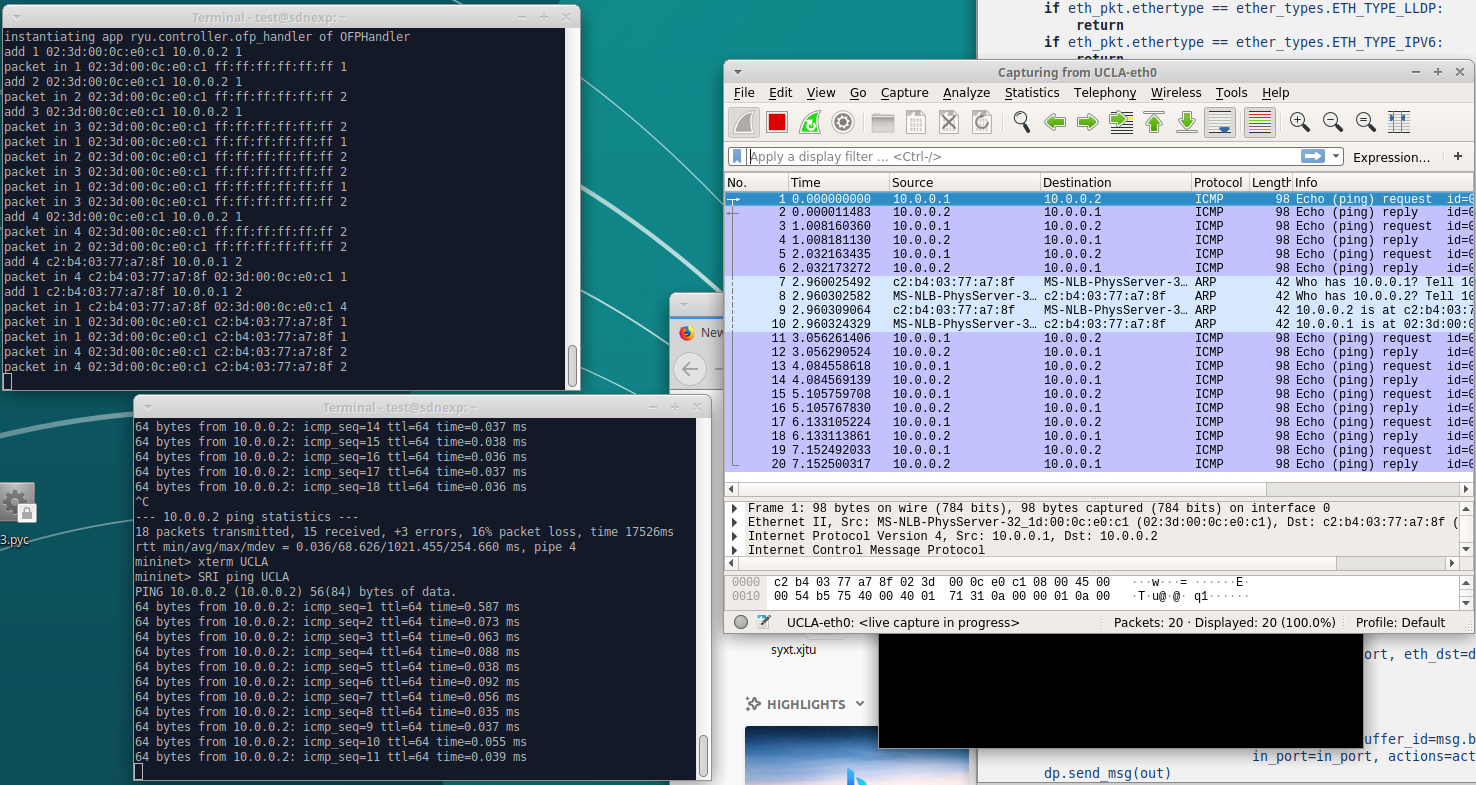
data = msg.data

out = parser.OFPPacketOut(datapath=dp, buffer\_id=msg.buffer\_id,

in\_port=in\_port, actions=actions, data=data)

dp.send\_msg(out)

可以看到，确实在更改代码后ARP包少了不少。



## 附加题

### 问题分析

附加题要求采用另一种方式实现环路广播风暴的避免，故可考虑采用ryu自带的生成树协议，或给每个包加上寿命字段。

在本次实验中，我采用官方给出的stp生成树代码。

代码实现原理可参考<https://osrg.github.io/ryu-book/zh_tw/html/spanning_tree.html#id5>

STP的工作过程如下：首先进行根网桥的选举，其依据是网桥优先级（bridge priority）和MAC地址组合生成的桥ID，桥ID最小的网桥将成为网络中的根桥（bridge root）。在此基础上，计算每个节点到根桥的距离，并由这些路径得到各冗余链路的代价，选择最小的成为通信路径（相应的端口状态变为forwarding），其它的就成为备份路径(相应的端口状态变为blocking)。STP生成过程中的通信任务由BPDU完成，这种数据包又分为包含配置信息的配置BPDU（其大小不超过35B）和包含拓扑变化信息的通知BPDU（其长度不超过4B）。

### 代码实现

#该代码大部分由ryu示例代码更改而来，实现方法也与上方页面介绍一致

import struct

from ryu.base import app\_manager

from ryu.controller.handler import MAIN\_DISPATCHER

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

from ryu.ofproto import ofproto\_v1\_0

from ryu.lib import dpid as dpid\_lib

from ryu.lib import stplib

from ryu.lib.mac import haddr\_to\_str

class SimpleSwitchStp(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_0.OFP\_VERSION]

\_CONTEXTS = {'stplib': stplib.Stp}

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

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

self.mac\_to\_port = {}

self.stp = kwargs['stplib']

def add\_flow(self, datapath, in\_port, dst, actions):

ofproto = datapath.ofproto

wildcards = ofproto\_v1\_0.OFPFW\_ALL

wildcards &= ~ofproto\_v1\_0.OFPFW\_IN\_PORT

wildcards &= ~ofproto\_v1\_0.OFPFW\_DL\_DST

match = datapath.ofproto\_parser.OFPMatch(

wildcards, in\_port, 0, dst,

0, 0, 0, 0, 0, 0, 0, 0, 0)

mod = datapath.ofproto\_parser.OFPFlowMod(

datapath=datapath, match=match, cookie=0,

command=ofproto.OFPFC\_ADD, idle\_timeout=0, hard\_timeout=0,

priority=ofproto.OFP\_DEFAULT\_PRIORITY,

flags=ofproto.OFPFF\_SEND\_FLOW\_REM, actions=actions)

datapath.send\_msg(mod)

def delete\_flow(self, datapath):

ofproto = datapath.ofproto

wildcards = ofproto\_v1\_0.OFPFW\_ALL

match = datapath.ofproto\_parser.OFPMatch(

wildcards, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

mod = datapath.ofproto\_parser.OFPFlowMod(

datapath=datapath, match=match, cookie=0,

command=ofproto.OFPFC\_DELETE)

datapath.send\_msg(mod)

@set\_ev\_cls(stplib.EventPacketIn, MAIN\_DISPATCHER)

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

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

dst, src, \_eth\_type = struct.unpack\_from('!6s6sH', buffer(msg.data), 0)

dpid = datapath.id

self.mac\_to\_port.setdefault(dpid, {})

self.logger.debug("packet in %s %s %s %s",

dpid, haddr\_to\_str(src), haddr\_to\_str(dst),

msg.in\_port)

self.mac\_to\_port[dpid][src] = msg.in\_port

if dst in self.mac\_to\_port[dpid]:

out\_port = self.mac\_to\_port[dpid][dst]

else:

out\_port = ofproto.OFPP\_FLOOD

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

if out\_port != ofproto.OFPP\_FLOOD:

self.add\_flow(datapath, msg.in\_port, dst, actions)

out = datapath.ofproto\_parser.OFPPacketOut(

datapath=datapath, buffer\_id=msg.buffer\_id, in\_port=msg.in\_port,

actions=actions)

datapath.send\_msg(out)

@set\_ev\_cls(stplib.EventTopologyChange, MAIN\_DISPATCHER)

def \_topology\_change\_handler(self, ev):

dp = ev.dp

dpid\_str = dpid\_lib.dpid\_to\_str(dp.id)

msg = '拓扑结构发生变动，更改mac表'

self.logger.debug("[dpid=%s] %s", dpid\_str, msg)

if dp.id in self.mac\_to\_port:

del self.mac\_to\_port[dp.id]

self.delete\_flow(dp)

@set\_ev\_cls(stplib.EventPortStateChange, MAIN\_DISPATCHER)

def \_port\_state\_change\_handler(self, ev):

dpid\_str = dpid\_lib.dpid\_to\_str(ev.dp.id)

of\_state = {stplib.PORT\_STATE\_DISABLE: 'DISABLE',

stplib.PORT\_STATE\_BLOCK: 'BLOCK',

stplib.PORT\_STATE\_LISTEN: 'LISTEN',

stplib.PORT\_STATE\_LEARN: 'LEARN',

stplib.PORT\_STATE\_FORWARD: 'FORWARD'}

self.logger.debug("[dpid=%s][port=%d] state=%s",

dpid\_str, ev.port\_no, of\_state[ev.port\_state])

## 题外问题

在本次实验中，我最开始采用自己的虚拟机与虚拟环境（python3.10，eventlet23.2，pip3），但在下载并启动ryu-manager后，发现无法找到命令。

在将ryu/bin添加到环境变量中后，执行ryu-manager后继续报错，错误为无法找到ryu的一个库文件，将其找到并移动到正确位置后，仍然报错。

错误为在eventlet中一个类没有某属性。经查得eventlet版本过高，无法适配，故将其版本降为20.0，仍然报错。

报错为在python3中修改了一个为常量的属性。经查找，python3.10版本过高，需将python降为3.9版本。在修改版本、修改python软连接与默认python版本后，仍然报错。

错误为ryu未安装。推测为新环境无法找到旧环境的ryu。更改pip版本、默认链接、镜像源后，使用pip install安装时，报错，无法安装。

错误为pip未启用SSH协议，无法安装。经查，需下载openssh-server最新版本。并需更新pip附加工具与pip自身版本，还需重装python。遂放弃，采用提供实验环境。