# Software Define Network Lab2

**实验准备：**

–Construct a network using Mininet

–Use wireshark to capture OpenFlow messages

–Ping and use ovs-ofctl to check the flow tables

**实验要求：**

–Construct the ARPANET-1969 (4 nodes) using Mininet

–Run Ryu as the remote controller

–Let UCLA ping UTAH

–If not reachable, try to solve it by programming with Ryu

**实验环境：**

Windows 10, VMware Workstation Pro, Ubuntu

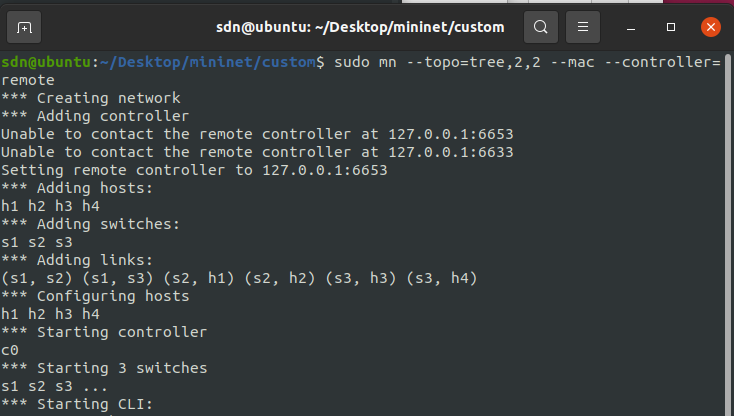
**实验过程：**

Warn-Up:

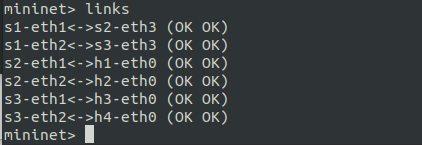
1. 构建一个简单拓扑的Mininet，使用Wireshark捕获OpenFLow协议数据包以及查看交换机流表

我们在mininet/custom目录下打开命令行终端，输入sudo mn --topo=tree,2,2 --mac --controller=remote

remote表示不用’Mininet’自带的控制器，尝试使用’Ryu’等远端控制器。



在mininet命令行中输入links，查看连接情况。

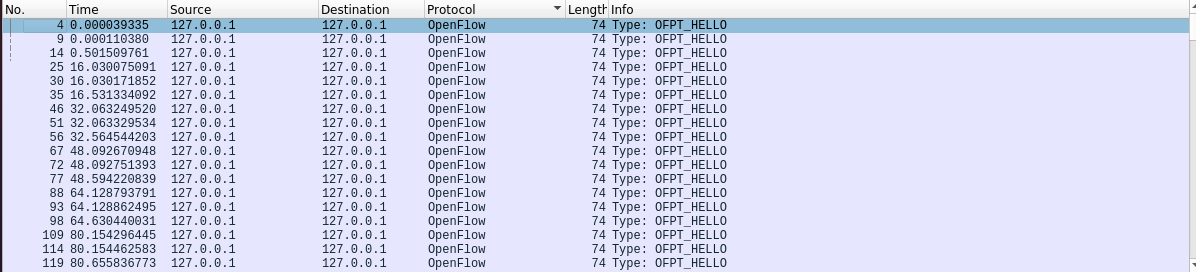


可以看到，主机h1和h2连接交换机s2，主机h3和h4连接交换机s3，交换机s2和s3连接交换机s1，构成一个树形拓扑结构。

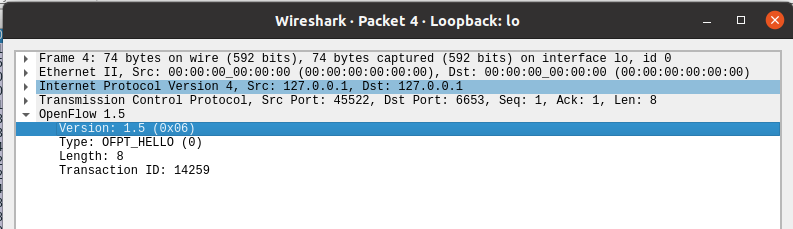
输入sudo ovs-ofctl dump-flows s1，查看s1的流表表项。



输入sudo wireshark，启动Wireshark，选择Loopback:lo端口，开始捕获。



我们捕获到不少OpenFlow协议报文，选择一条查看详细内容。



这是序号为4的报文，协议为OpenFlow，Version为1.5，类型为OFPT\_Hello，长度为8，ID为14259。

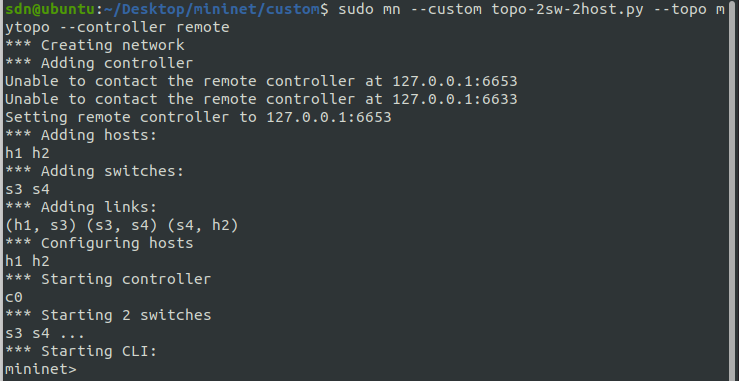
通过查找相关资料我们得知，OpenFLow协议报文类型有三种分类，即同步，异步和Controller to Switch类型，其中OFPT\_Hello类型属于同步类型，当连接启动时交换机和控制器会发送Hello消息进行交互。资料链接如下：

<https://www.h3c.com/cn/d_201811/1131080_30005_0.htm>

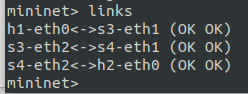
2. 使用Ryu APP查看拓扑

我们在mininet/custom目录下打开命令行终端，输入sudo mn --custom topo-2sw-2host.py --topo mytopo --controller remote

remote表示不用’Mininet’自带的控制器，尝试使用’Ryu’等远端控制器。

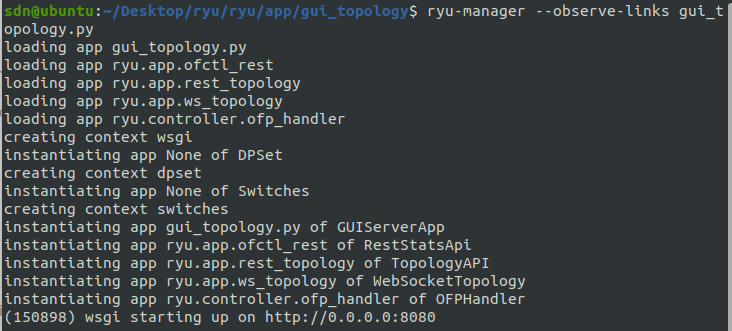


在mininet命令行中输入links，查看连接情况。

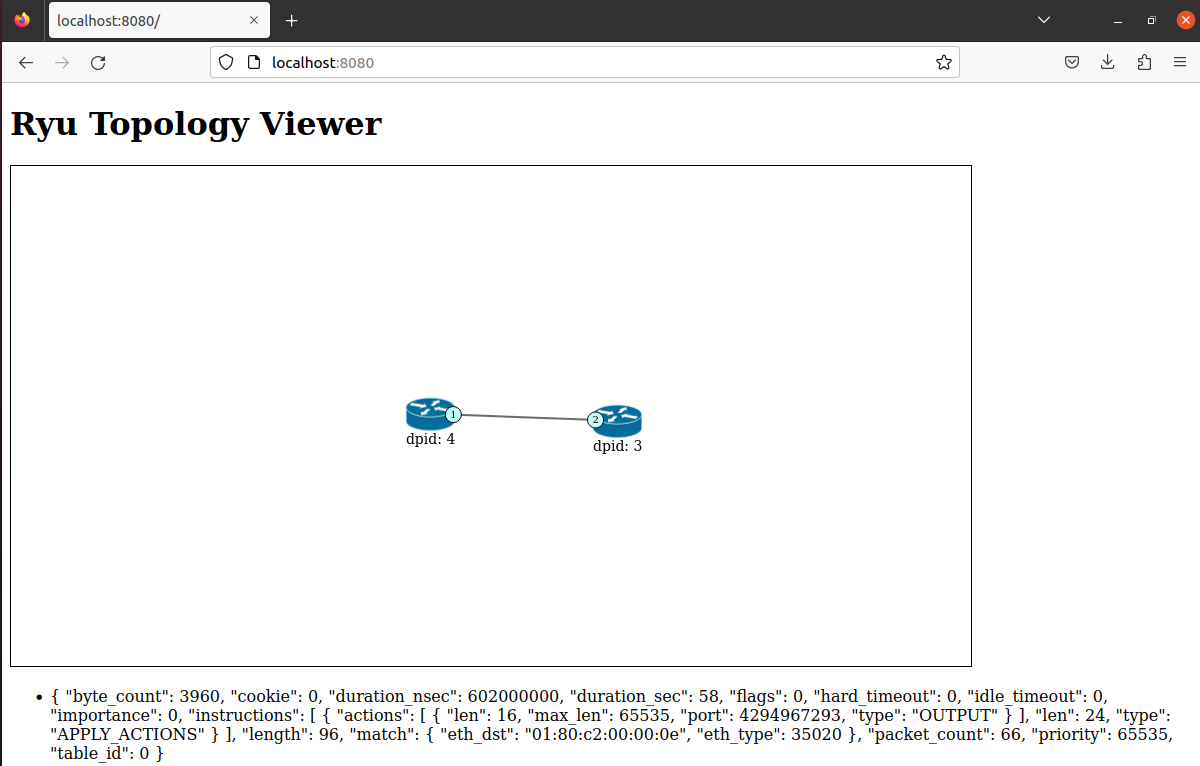


可以看到，主机h1连接交换机s3，交换机s3连接交换机s4，交换机s4连接主机h2，构成拓扑结构。

我们在ryu/ryu/app/gui\_topology目录下，打开命令行终端，输入ryu-manager --observe-links sdn/ryu/ryu/app/gui\_topology/gui\_topology.py

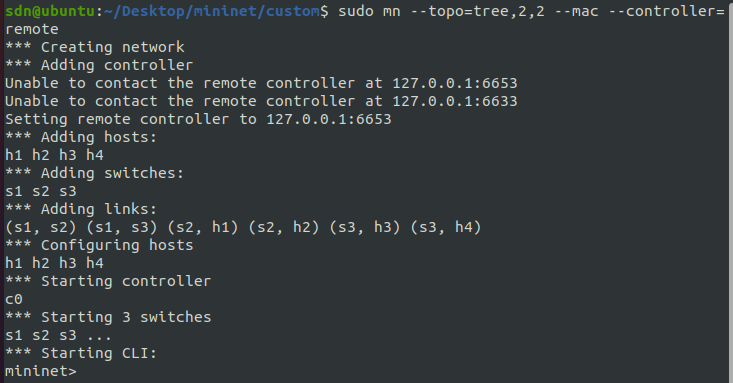


在浏览器中访问localhost:8080以查看拓扑，点击交换机即可查看流表。

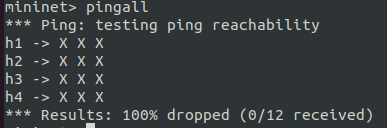


3. 实现Ryu简单交换机

输入sudo mn --controller remote --mac --topo=tree,2,2，创建树形拓扑。



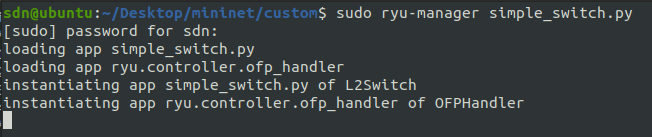
在mininet命令行中输入pingall，测试各节点之间的连通性。



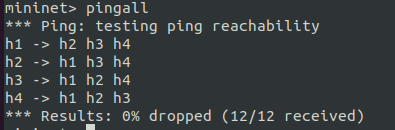
发现无法ping通，这是因为我们还没有启动ryu-manager。

我们将实验指导书里的ryu简单交换机的参考代码保存，命名为simple\_switch.py，保存在mininet/custom目录下，代码详见实验指导书。

输入sudo ryu-manager simple\_switch.py，启动Ryu控制器。

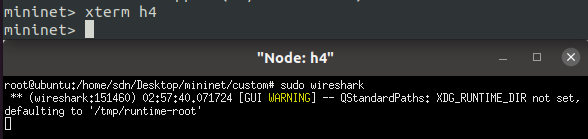


等待一段时间，直到ryu-manager光标不再闪烁之后，在mininet命令行中再次输入pingall，测试各节点连通性。

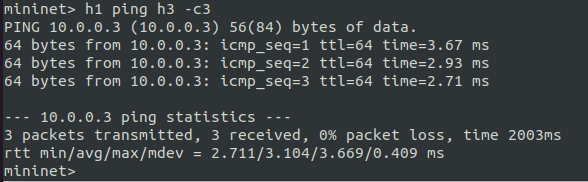


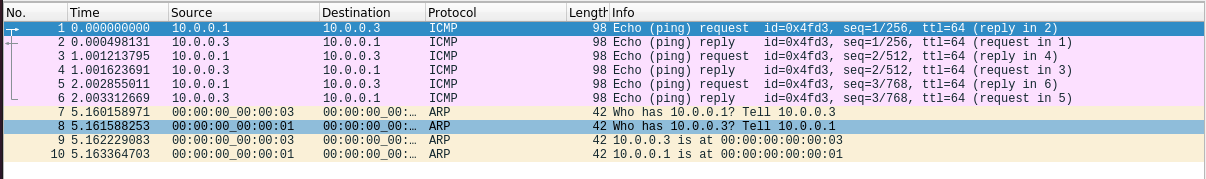
这次我们可以发现，各个节点之间都能互相ping通。

在mininet命令行中输入xterm h4，打开h4的xterm终端，启动Wireshark捕获端口h4-eth0。



在mininet命令行中输入h1 ping h3 -c3，使主机h1 ping主机h3三次，同时开始h4-eth0端口的捕获。





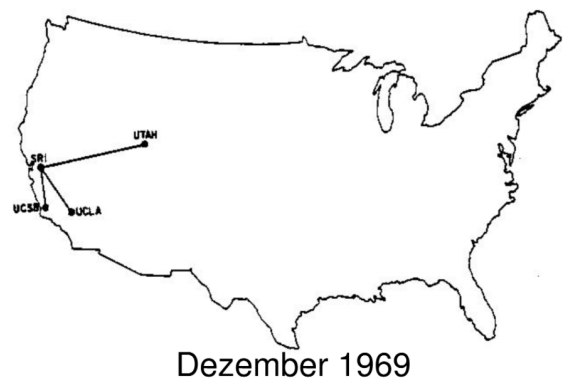
我们可以看到，抓包结果显示该交换机存在验证缺陷：packet\_in\_handler()函数会将数据包洪泛到交换机的所有端口，故h1和h3通讯时，h4也会收到所有的包。

Task:

4. 实现Ryu自学习交换机

首先我们需要实现1969年的Arpanet拓扑结构，1969年的ARPANET非常简单，仅由四个结点组成。假设每个结点都对应一个交换机，每个交换机都具有一个直连主机。前文给出的简单交换机洪泛数据包，虽然能初步实现主机间的通信，但会带来不必要的带宽消耗，并且会使通信内容泄露给第三者。需要在简单交换机的基础上实现二层自学习交换机，避免数据包的洪泛。

1969年的Arpanet拓扑结构示意图如下：



自学习交换机的实现框架如下：

（1）控制器为每个交换机维护一个mac-port映射表。

（2）控制器收到packet\_in消息后，解析其中携带的数据包。

（3）控制器学习src\_mac - in\_port映射。

（4）控制器查询dst\_mac，如果未学习，则洪泛数据包；如果已学习，则向指定端口转发数据包(packet\_out)，并向交换机下发流表项(flow\_mod)，指导交换机转发同类型的数据包。

拓扑结构和mininet构建实现于topo\_1969\_1.py文件之中，代码详见实验指导书。

基于实验指导书给出的自学习交换机框架，我们完善代码实现自学习交换机，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

        self.mac\_to\_port = {}

    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

        # :)

        # Save dpid and src of in\_port to dict mac\_to\_port, learning

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

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

            # Setting direction according to the table

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

        else:

            out\_port = ofp.OFPP\_FLOOD  # Flood

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

        if out\_port != ofp.OFPP\_FLOOD:  # Add flow

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

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

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

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

        dp.send\_msg(out)

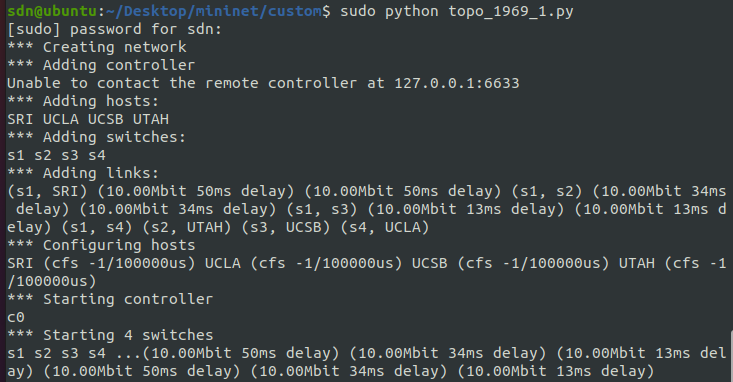
代码分析：

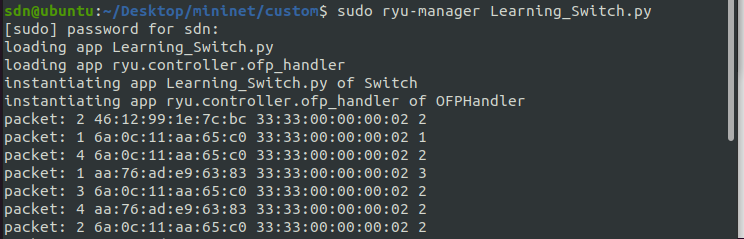
我们在Switch()类型初始化函数\_\_init\_\_()函数中，选择初始化一个mac\_to\_port字典(Dictionary)，用于记录MAC地址和端口等信息，初始化为空。Python的字典数据类型的资料如下：

<https://www.runoob.com/python/python-dictionary.html>

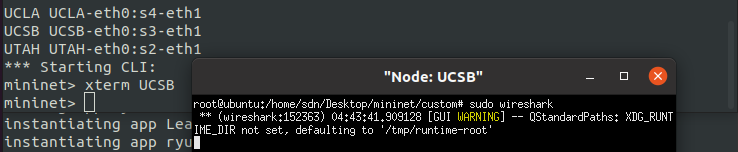
我们主要需要修改代码里的packer\_in\_handler()函数，在得到了源MAC地址和目的MAC地址之后，我们将in\_port对象内的信息记录到mac\_to\_port字典里的dpid和src键下，如果目的MAC地址已经记录在了字典里的dpid键下，我们可以根据字典确定out\_port的转发方向，如果字典里没有已经存在的记录，则洪泛转发。根据选择的转发方式，确定Output时的动作(Action)。同时如果不是洪泛转发，则需要添加流表。最后在完成流表的添加后，再进行转发数据包的封装与实施转发。

修改后的代码命名为Learning\_Switch.py，在命令行终端输入sudo python topo\_1969\_1.py，构建mininet网络拓扑；在另一个命令行终端输入sudo ryu-manager Learning\_Switch.py，启动Ryu控制器。

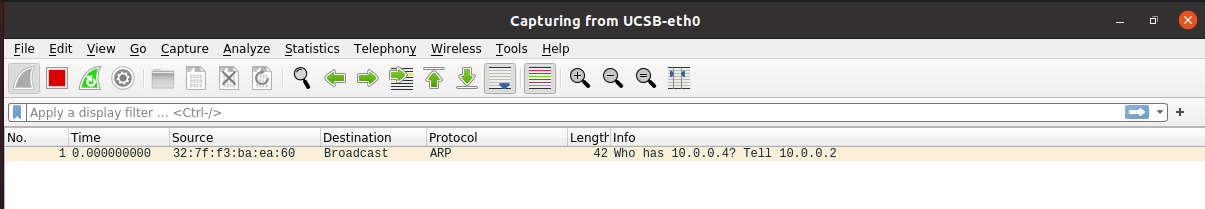


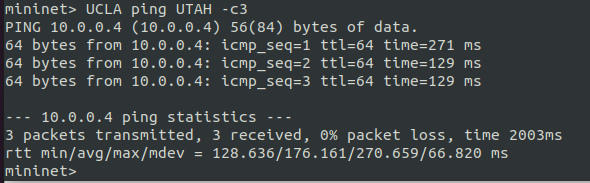


在mininet命令行中输入xterm UCSB，启动UCSB主机的xterm终端，在终端内输入sudo wireshark，启动Wireshark进行数据包捕获。

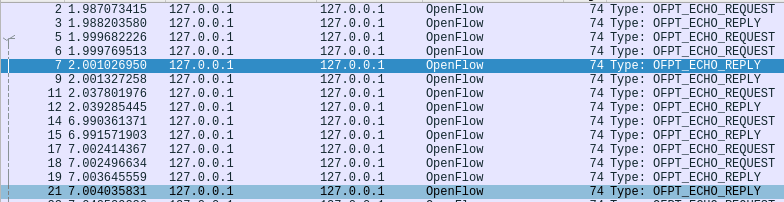


在Wireshark中选择UCSB-eth0端口进行捕获，同时在mininet命令行中输入UCLA ping UTAH -c3，测试两主机之间的连通性，查看捕获结果。





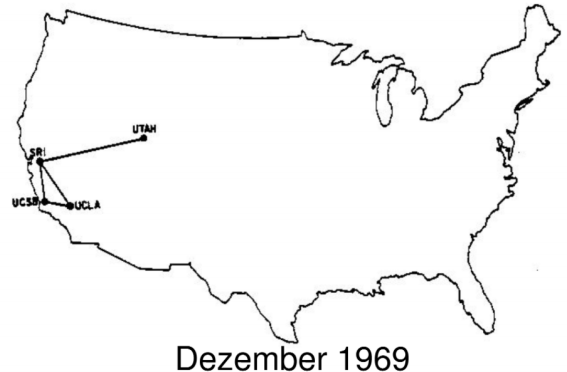
可以看到，UCLA成功ping通UTAH，而且UCSB没有收到相关数据包，成功实现了自学习交换机。



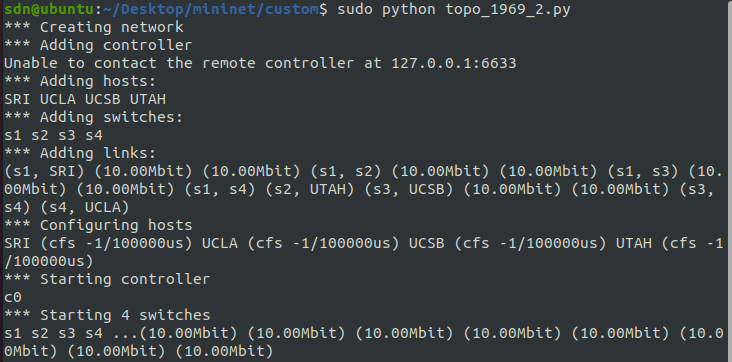
5. 处理环路广播的自学习交换机

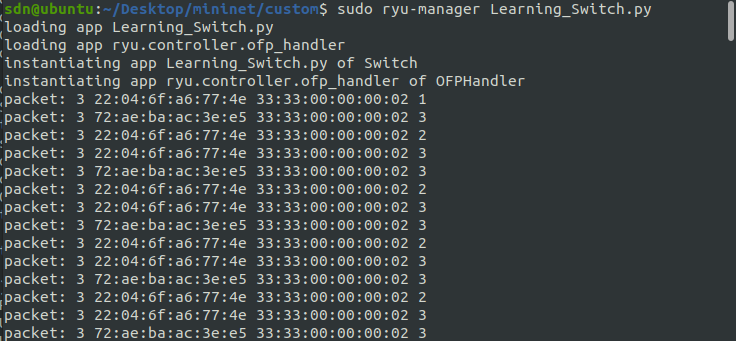
UCLA和UCSB通信频繁，两者间建立了一条直连链路。在新的拓扑topo\_1969\_2.py中运行自学习交换机，UCLA和UTAH之间无法正常通信。分析流表发现，源主机虽然只发了很少的几个数据包，但流表项却匹配了上千次。

新的Arpanet拓扑结构示意图如下：



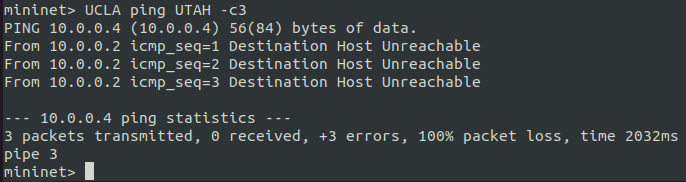
在命令行终端内输入sudo python topo\_1969\_2.py，构建mininet网络拓扑；在另一个命令行终端内输入sudo ryu-manager Learning\_Switch.py，启动Ryu控制器。





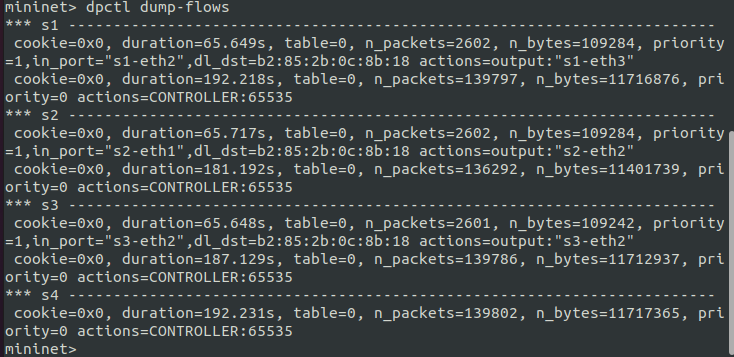
可以看到，在ryu-manager中出现了数目相当大的流表记录，且同一个数据包多次出现，只有in\_port端口的键值发生了变化。

在mininet命令行中输入UCLA ping UTAH -c3，测试两节点之间的连通性。



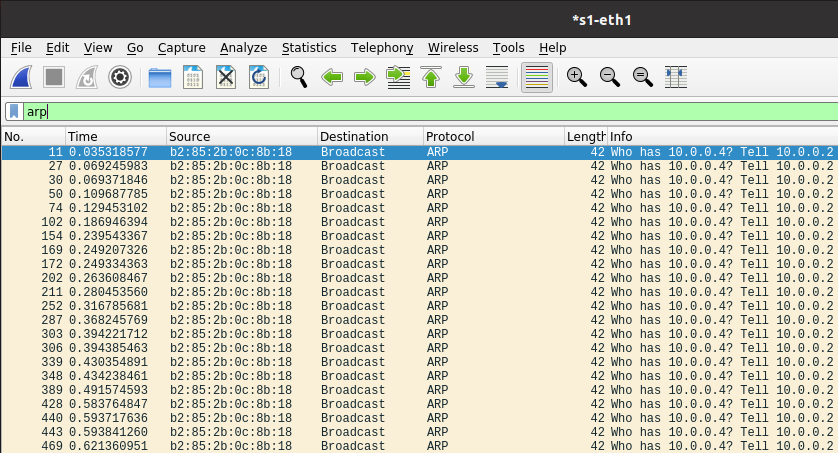
可以看到，两节点之间无法ping通，显示主机不可达。

在mininet命令行终端内输入dpctl dump-flows，查看与各主机直接相连的交换机的流表表项。



可以看到，虽然UCLA与UTAH主机之间只ping了3次，但流表项却匹配了约140000次，出现了异常情况。

在命令行终端内输入sudo wireshark，启动Wireshark进行数据包捕获。可以随意选择一个交换机端口，例如我选择了s1-eth1端口进行捕获。



可以看到，WireShark也截取到了数目异常大的相同报文，这些报文均为ARP请求报文，UCLA主机（IP地址为10.0.0.2）发送ARP请求报文，请求UTAH主机（IP地址为10.0.0.4）的MAC地址，而这些ARP请求报文没有得到应答，所以UCLA主机持续发送请求报文，且由于拓扑结构中存在环路，形成了ARP广播风暴。这实际上是ARP广播数据包在环状拓扑中洪泛导致的，传统网络利用生成树协议解决这一问题。在SDN中，不必局限于生成树协议，可以通过多种新的策略解决这一问题。

处理环路广播的思路如下：

当序号为dpid的交换机从in\_port第一次收到某个src\_mac主机发出，询问 dst\_ip的广播ARP Request数据包时，控制器记录一个映射 (dpid, src\_mac, dst\_ip)->in\_port。下一次该交换机收到同一(src\_mac, dst\_ip)但in\_port不同的ARP Request数据包时直接丢弃，否则洪泛。

基于实验指导书给出的处理环路广播的自学习交换机框架，我们完善代码实现处理环路广播的自学习交换机，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

from ryu.lib.packet import arp

from ryu.lib.packet import ether\_types

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)

        # (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

        self.mac\_to\_port = {}

        self.arp\_table = {}

    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)

        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

        # we can use the logger to print some useful information

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

        # get protocols

        header\_list = dict((p.protocol\_name, p)

                           for p in pkt.protocols if type(p) != str)

        if dst == ETHERNET\_MULTICAST and ARP in header\_list:

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

            dst\_ip = header\_list[ARP].dst\_ip

            # set logger to show useful information

            self.logger.info("ARP Learning: %s %s %s %s",

                             dpid, src, dst\_ip, in\_port)

            # If info is already in ARP table

            if (dpid, src, dst\_ip) in self.arp\_table:

                # The same info comes from another port, Just Drop it

                if self.arp\_table[dpid, src, dst\_ip] != in\_port:

                    out = parser.OFPPacketOut(datapath=dp, buffer\_id=ofp.OFPCML\_NO\_BUFFER,

                                              in\_port=in\_port, actions=[], data=None)  # Drop

                    dp.send\_msg(out)

                    return

            # If info is not in ARP table, Learn and FLood it

            else:

                # Arp table learning

                self.arp\_table[(dpid, src, dst\_ip)] = in\_port

                actions = [parser.OFPActionOutput(ofp.OFPP\_FLOOD)]  # Flood

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

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

                dp.send\_msg(out)

        # self-learning

        # you need to code here to avoid the direct flooding

        # having fun

        # :)

        # just code in mission 1

        # Save dpid and src of in\_port to dict mac\_to\_port, learning

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

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

            # Setting direction according to the table

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

        else:

            out\_port = ofp.OFPP\_FLOOD  # Flood

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

        if out\_port != ofp.OFPP\_FLOOD:  # Add flow

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

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

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

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

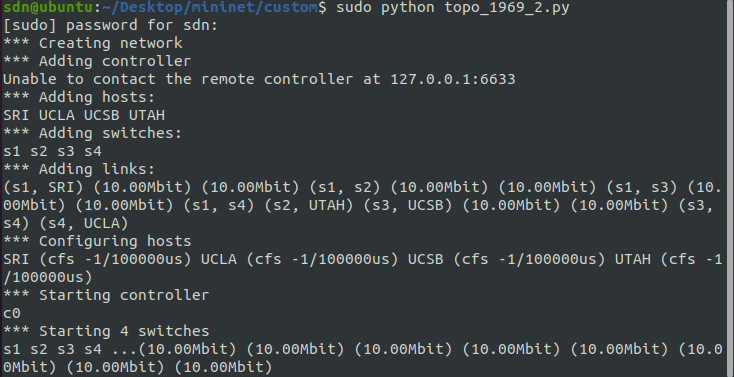
        dp.send\_msg(out)

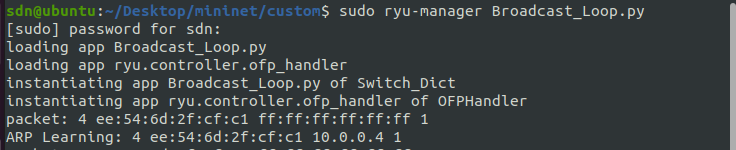
代码分析：

在自学习交换机的基础上，我们在Switch()类型初始化函数\_\_init\_\_()函数中，除了选择初始化一个mac\_to\_port字典，还初始化了一个arp\_table字典，用于记录dpid，源MAC地址，目的IP地址等信息。

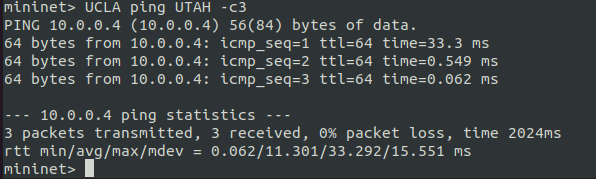
我们还需要修改代码里的packet\_in\_handler()函数，如果header\_list里的键值符合ARP报文的特征，即目的MAC地址(dst)是以太网组播(ETHERNET\_MULTICAST)并且header\_list的键为ARP类型，则记录header\_list对象里的dst\_ip，同时使用logger显示有帮助的信息。如果表项的内容已经存在于arp\_table中并且arp\_table内记录的信息与in\_port对象不完全相同，即符合src\_mac, dst\_ip相同但in\_port不同的条件，我们就丢弃这个数据包，否则若表项内容不存在于arp\_table中，则洪泛转发数据包。随后就是自学习交换机中已经实现了的功能。

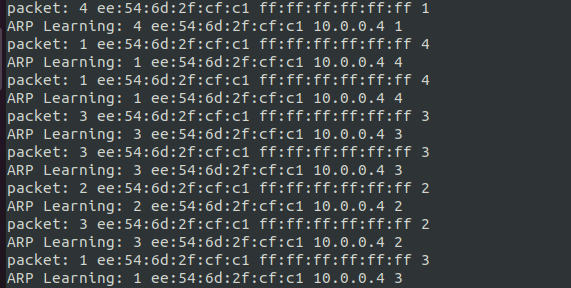
修改后的代码命名为Broadcast\_Loop.py，在命令行终端内输入sudo python topo\_1969\_2.py，构建mininet网络拓扑；在另一个命令行终端内输入sudo ryu-manager Broadcast\_Loop.py，启动Ryu控制器。





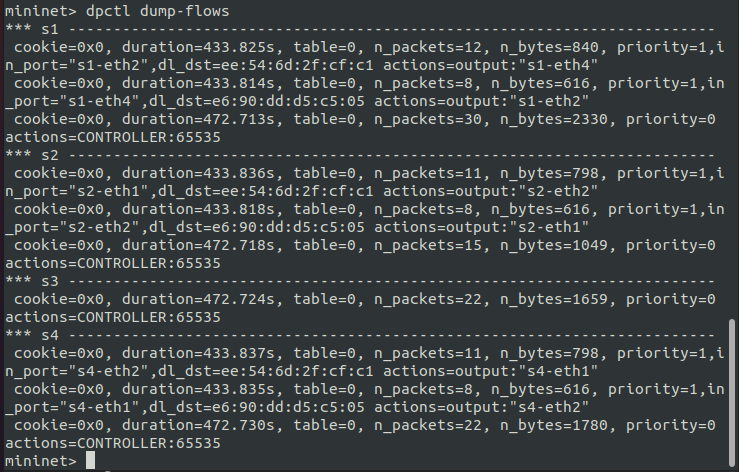
在mininet命令行中输入UCLA ping UTAH -c3，测试两节点之间的连通性。





可以看到，UCLA主机成功ping通UTAH主机。

在mininet命令行终端内输入dpctl dump-flows，查看与各主机直接相连的交换机的流表表项。



可以看到，流表表项的匹配次数明显减少。

**实验结果：**

1. 学会了使用Ryu远程控制器，以及使用Ryu控制器查看网络拓扑结构。

2. 学会了使用命令行指令查看流表表项以及使用Wireshark捕获控制平面报文信息。

3. 学会了使用Ryu远程控制器来配置网络拓扑中的交换机，学习了简单洪泛交换机的代码，功能与缺陷。

4. 学会了实现Ryu自学习交换机，熟悉了Ryu控制器中的数据结构。

5. 学会了基于Ryu自学习交换机处理环路广播，了解了交换机的各种操作。