

# SDN Experiment 1

---

## SDN Experiment 1

- 1 前言
- 2 实验环境
  - 2.1 virtualbox & xubuntu
    - 2.1.1 virtuablbox
    - 2.1.2 xubuntu
  - 2.2 Wireshark
  - 2.3 VS Code
  - 2.4 Mininet
    - 2.4.1 常用命令
    - 2.4.2 查看流表
    - 2.4.3 自定义拓扑
  - 2.5 Ryu
    - 2.5.1 源码目录
    - 2.5.2 查看拓扑图
    - 2.5.3 Ryu APP
- 3 实验内容
  - 3.1 实验1
    - 题目
    - 说明
    - 示例
  - 3.2 实验2
    - 题目
    - 说明
    - 示例
- 4 总结
- 5 扩展资料

## 1 前言

---

- 《软件定义网络》课程实验总计四次，这是第一次的实验指导书
- 实验完成情况可当场验收，可提交实验报告，鼓励当场验收
- 实验虚拟机请提前在[moodle](#)下载，建议使用个人电脑运行
- 实验内容要求各位提前准备，机房现场主要负责答疑和验收
- 实验需独自完成，鼓励互相学习和交流，严禁抄袭
- 关于实验部分的疑问或反馈或Anything请发送邮件至：[sdnexp2019@outlook.com](mailto:sdnexp2019@outlook.com)

标题格式：

cs60-小胖-关于xxx

## 2 实验环境

---

本次实验主要用到的工具如下所示，提供安装好所需工具的虚拟机，也可自行参考文档手动安装

- 虚拟机及系统

virtualbox (free, GPL)

xubuntu (xubuntu配置要求低，自行配置环境选择任意Linux发行版均可)

- 网络模拟

Mininet

- 控制器

Ryu

- 抓包工具

Wireshark (或tcpdump)

- 文本编辑器

虚拟机中安装了VS Code，自选均可

以下内容为上述工具的基本教程，有熟悉的章节自行跳过

### 2.1 virtualbox & xubuntu

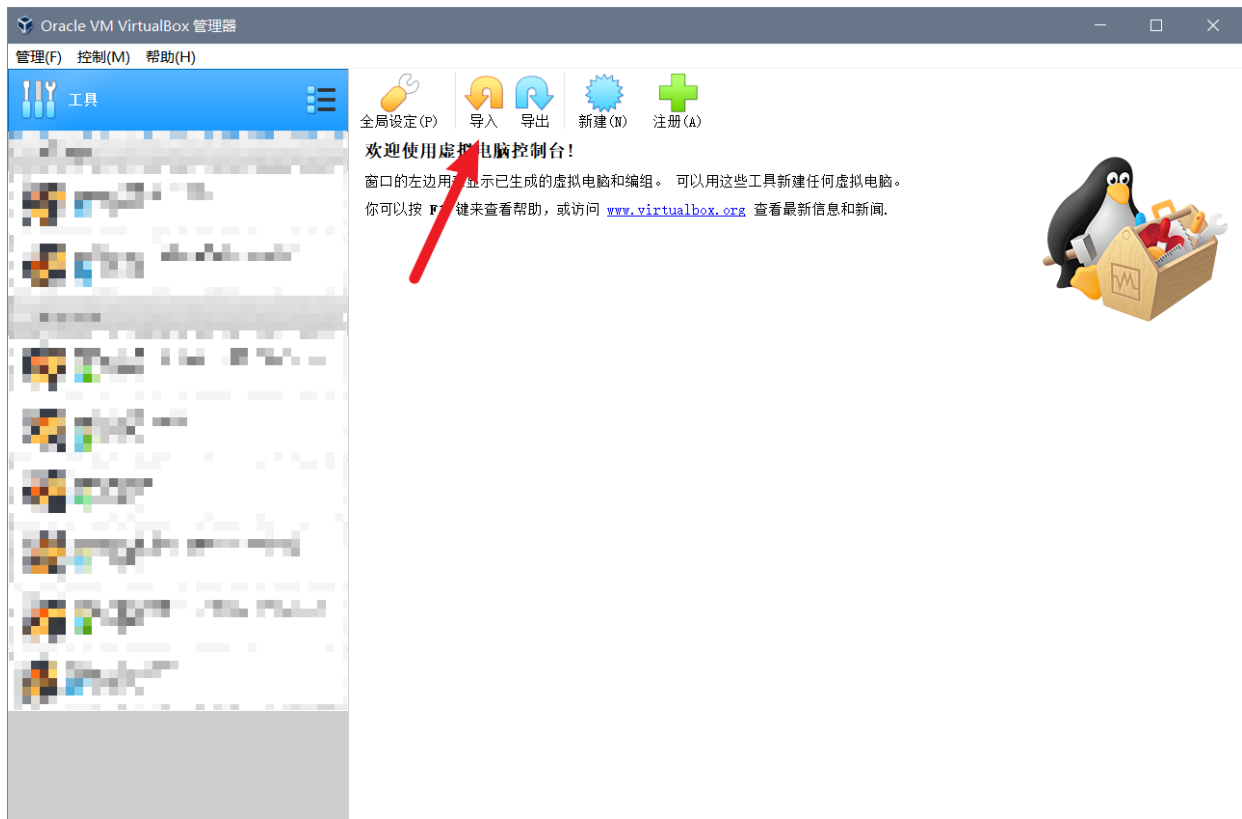
#### 2.1.1 virtualbox

- 安装

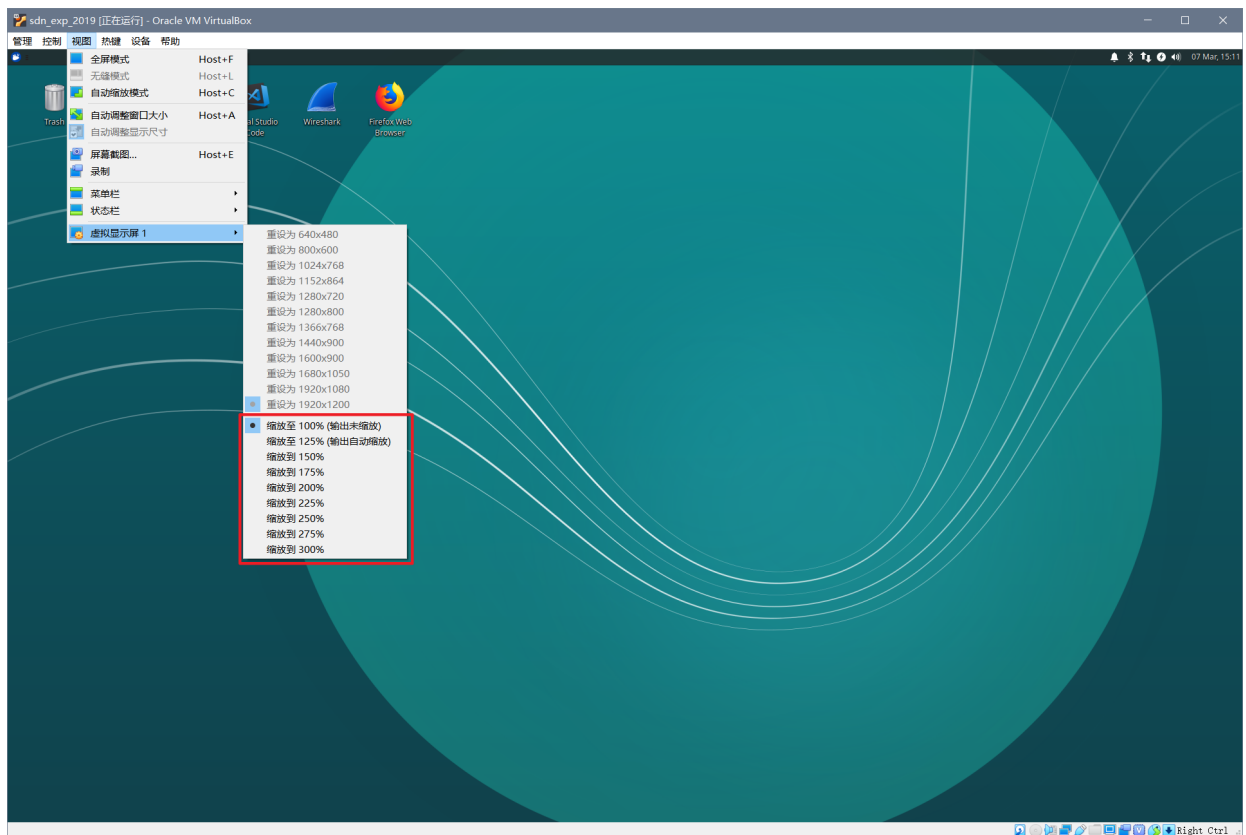
根据自己的操作系统选择[virtualbox下载](#)，默认选项安装（以下教程在6.0.4版本测试通过）

- 虚拟机导入

从[moodle](#)下载的虚拟机，按照下图所示导入。可根据电脑配置在设置中分配更多的核心数和内存，其余选项默认



导入后根据显示器的分辨率调整缩放至合适的大小，也可安装virtualbox的增强工具自适应调整分辨率



- 虚拟机导出

在机房电脑进行实验的同学每次结束后需将虚拟机关闭后导出，保存到自己的U盘中，下次实验再次导入继续实验

## 2.1.2 xubuntu

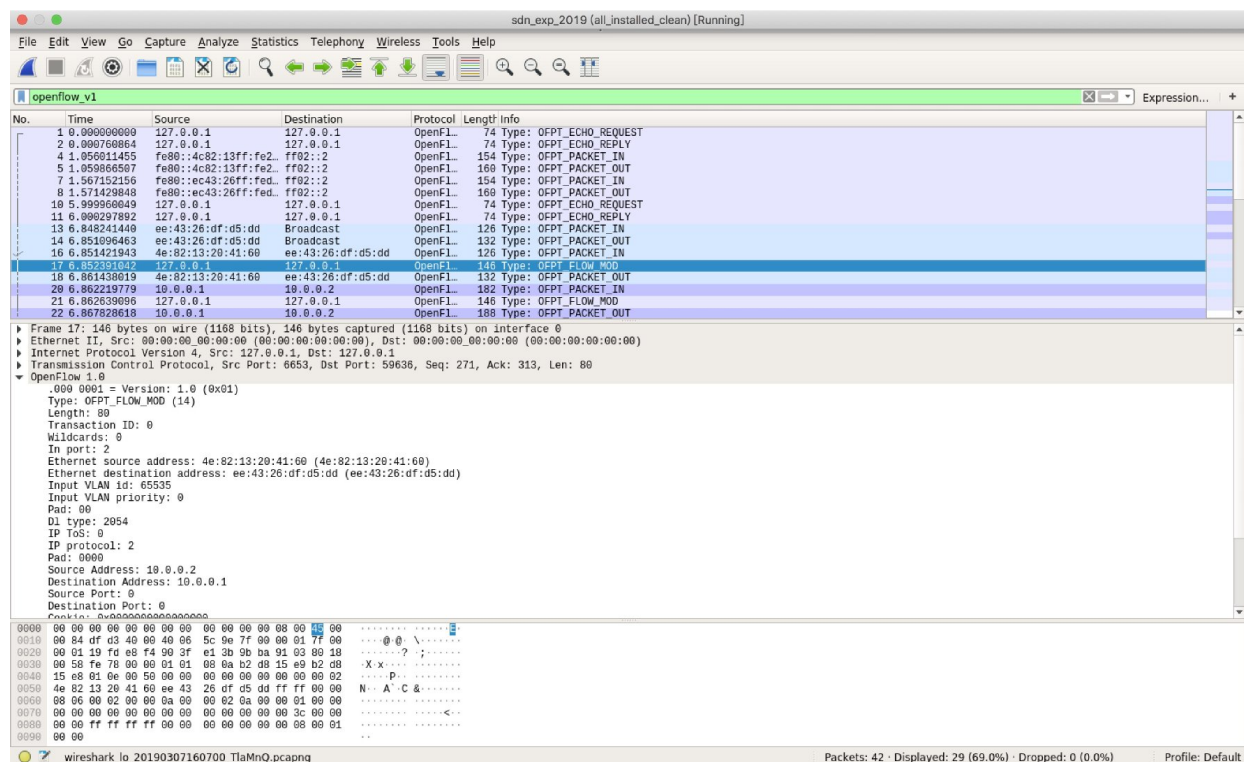
实验虚拟机镜像基于xubuntu18.04 64位制作，各工具均安装在 `~/sdn`

用户名: `test`，密码: `1`

## 2.2 Wireshark

学习Mininet和Ryu的过程中，Wireshark可以抓取控制器和交换机通讯的数据包，并且支持OpenFlow协议的解析

- 由于mininet利用linux的namespace在本地虚拟的网络结构，抓取的端口应选择loopback 抓取loopback时需root权限，故 `sudo wireshark` 启动
- 输入openflow\_xx过滤指定版本的OpenFlow协议报文



## 2.3 VS Code

这里选择自己习惯的文本编辑器即可

若选择VS Code

- 建议安装推荐的Python Extension
  - Python Interpreter切换至 `/usr/bin/python`
  - mininet的开发目录建议在 `mininet/example`，ryu的开发目录建议在 `ryu/ryu/app`
- 这样练习时可以参考提供的示例，同时VSC可以提供舒服的命令补全

## 2.4 Mininet

Mininet的基本教程请阅读官网提供的[walkthrough](#)，安装方式推荐源码安装

## 2.4.1 常用命令

```
# shell prompt
mn -h # 查看mininet命令中的各个选项
sudo mn -c # 不正确退出时清理mininet

# 下面的命令可以在 'sudo mn' 新建的简单拓扑上查看运行结果
# mininet CLI
net # 显示当前网络拓扑
dump # 显示当前网络拓扑的详细信息
xterm h1 # 给节点h1打开一个终端模拟器
sh [COMMAND] # 在mininet命令行中执行COMMAND命令
h1 ping -c3 h2 # 即h1 ping h2 3次
pingall # 即ping all
h1 ifconfig # 查看h1的网络端口及配置
h1 arp # 查看h1的arp表
link s1 h1 down/up # 断开/连接s1和h1的链路
exit # 退出mininet CLI

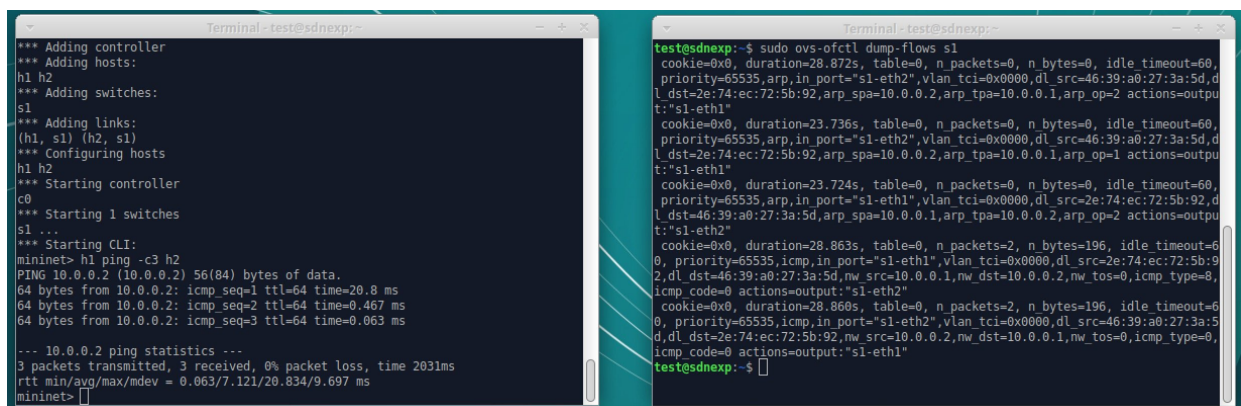
# ovs(run in shell prompt)
sudo ovs-ofctl show s1 # 查看交换机s1的基本信息
sudo ovs-ofctl dump-flows s1 # 查看s1的流表
sudo ovs-ofctl -O OpenFlow13 dump-flows # 查看s1中OpenFlow1.3版本的流表信息
```

## 2.4.2 查看流表

新建简单拓扑查看对应的流表项

```
sudo mn
mininet> h1 ping -c3 h2

sudo ovs-ofctl dump-flows s1
```



The image shows two terminal windows. The left window shows the output of 'sudo mn' and 'mininet> h1 ping -c3 h2', displaying network setup details and ping results. The right window shows the output of 'sudo ovs-ofctl dump-flows s1', displaying the flow table for switch s1 with various flow entries including ICMP and ARP requests.

## 2.4.3 自定义拓扑

- 简单的写法

示例位于 `sdn/mininet/custom/topo-2sw-2host.py` 如下：

```

from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def build( self ):
        "Create custom topo."

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )

topos = { 'mytopo': ( lambda: MyTopo() ) }

```

运行拓扑的命令为：

```

cd ~/sdn/mininet/custom

sudo mn --custom topo-2sw-2host.py --topo mytopo

```

输入 `pingall` 测试连通性如下：

```

test@sdnexp:~/sdn/mininet/custom$ sudo mn --custom topo-2sw-2host.py --topo mytopo
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s3 s4
*** Adding links:
(h1, s3) (s3, s4) (s4, h2)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 2 switches
s3 s4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet>

```

- 更推荐的写法

Mininet的[wiki](#)中推荐了另一种较复杂的写法，但简化了命令行命令

```
# sudo python topo_recommend.py
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel

class S1H2(Topo):
    def build(self):
        s1 = self.addSwitch('s1')
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')

        self.addLink(s1, h1)
        self.addLink(s1, h2)

def run():
    topo = S1H2()
    net = Mininet(topo)

    net.start()
    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel('info') # output, info, debug
    run()
```

运行拓扑的命令为：

```
sudo python topo_recommend.py
```

## 2.5 Ryu

Ryu的基本教程需要阅读文档[Ryu docs](#)，阅读前两个部分[Getting Started](#)和[Write Application](#)即可

### 2.5.1 源码目录

Ryu的源码目录如下：

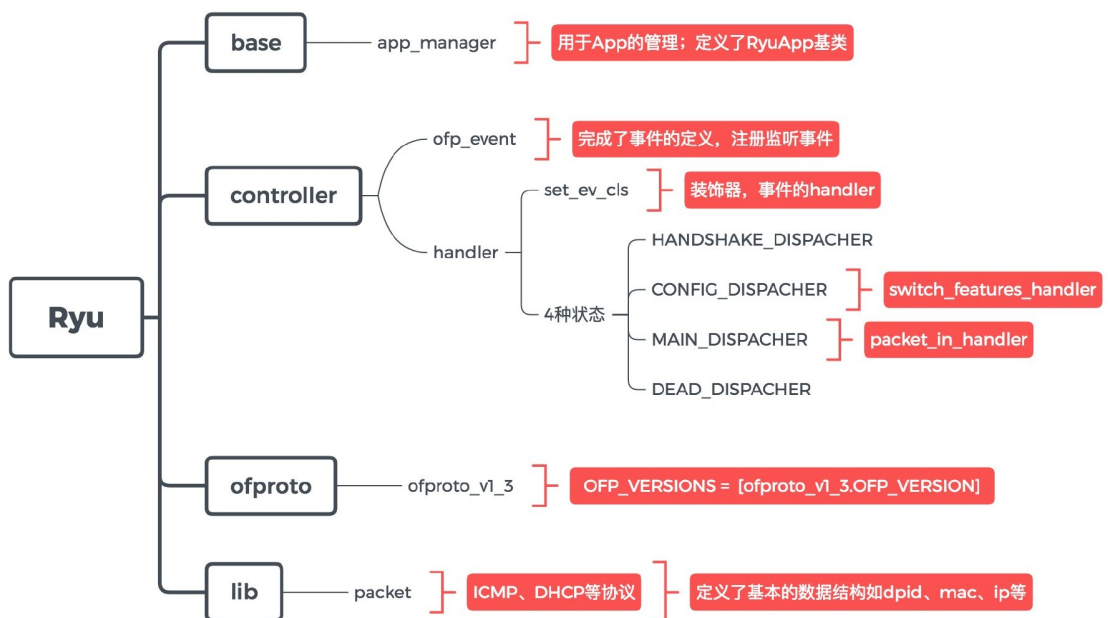
```
.
├─ app
│   └─ gui_topology
│   └─ ofctl
└─ base
```

```

├─ cmd
├─ contrib
├─ controller
├─ lib
│   └─ netconf
│   └─ of_config
│   └─ ovs
│   └─ packet
│   └─ xflow
├─ ofproto
├─ services
│   └─ protocols
├─ tests
│   └─ integrated
│   └─ mininet
│   └─ packet_data
│   └─ packet_data_generator
│   └─ packet_data_generator2
│   └─ packet_data_generator3
│   └─ switch
│   └─ unit
└─ topology

```

主要的文件目录及其作用参考下图



## 2.5.2 查看拓扑图

Ryu提供了查看拓扑图的APP，路径在 `ryu/ryu/app/gui_topology/gui_tology.py`

1. mininet新建拓扑



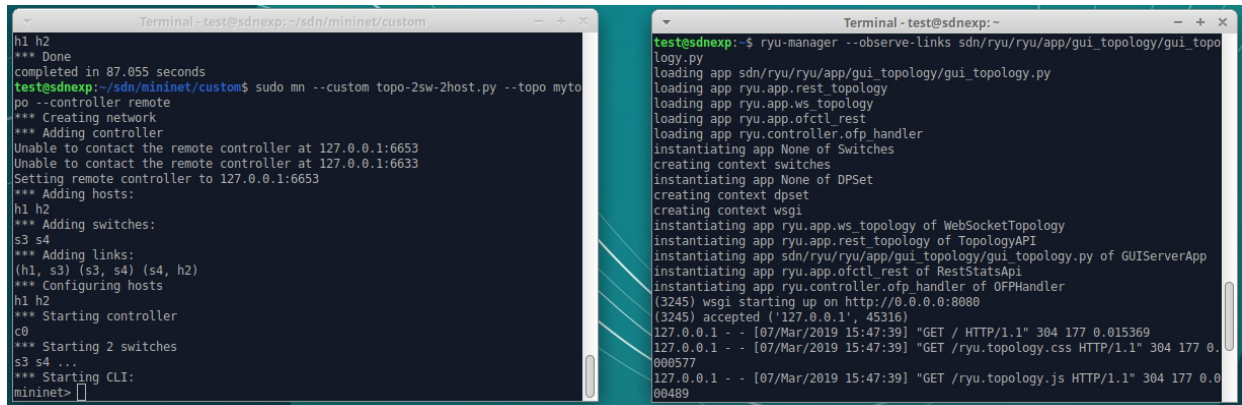
```
cd ~/sdn/mininet/custom
```

```
sudo mn --custom topo-2sw-2host.py --topo mytopo --controller remote
```

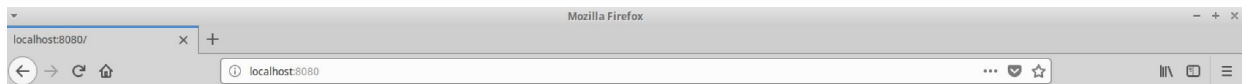
`--controller remote` 指定控制器为远程控制器即下面的Ryu，而不是mininet自带的控制器

## 2. 启动Ryu提供的APP:

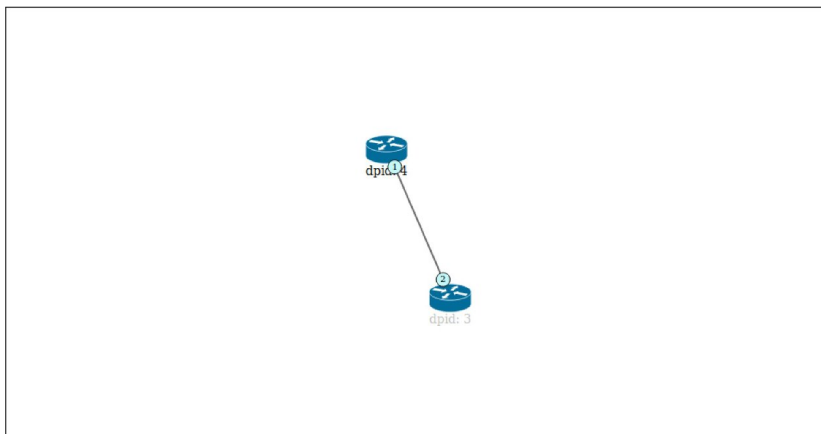
```
ryu-manager --observe-links sdn/ryu/ryu/app/gui_topology/gui_topology.py
```



## 3. 打开浏览器输入Ryu的IP地址，端口号为 8080 即可查看拓扑，本例中为 localhost:8080 点击交换机可查看对应的流表，交



### Ryu Topology Viewer



- { "priority": 65535, "length": 96, "hard timeout": 0, "byte count": 3360, "idle timeout": 0, "duration nsec": 727000000, "packet count": 56, "importance": 0, "duration sec": 49, "flags": 0, "cookie": 0, "table id": 0, "match": { "eth\_dst": "01:80:c2:00:00:0e", "eth\_type": 35020 }, "instructions": [ { "type": "APPLY\_ACTIONS", "len": 24, "actions": [ { "max\_len": 65535, "type": "OUTPUT", "port": 4294967293, "len": 16 } ] } ] }

## 2.5.3 Ryu APP

下面为Ryu文档中实现的交换机示例，我们给他增加了下发默认流表的函数，同时协议版本改为 OFP1.3:

```
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

class L2Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

    def add_flow(self, datapath, priority, match, actions):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
actions)]
        mod = parser.OFPFlowMod(datapath=dp, priority=priority,
match=match, instructions=inst)
        dp.send_msg(mod)

    # add default flow table which sends packets to the controller
    @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)

    # handle packet_in message
    @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

        actions = [parser.OFPActionOutput(ofp.OFPP_FLOOD)]
        out = parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id,
in_port=msg.match['in_port'], actions=actions, data=msg.data)
        dp.send_msg(out)

```

具体解释如下：

- `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
- `msg.datapath`  
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
- `datapath.ofproto`  
定义了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`
- `@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)`  
装饰器,第一个参数表示希望接收的事件，第二个参数告诉函数在该交换机的状态下被调用  
即 `packet_in_handler` 函数在`packet in`事件发生时被调用，且仅发生在交换机处于协商完毕的状态时  
四种状态在`ryu/ryu/controller/handler.py`中有详细的注释
- `actions`是一个列表，用于存放`action list`，可在其中添加动作
- `ofp_parser`类可以构造OFP的数据包
- 通过`datapath.send_msg()`函数发送OpenFlow数据结构，Ryu将把这个数据发送到对应的`datapath`

运行及抓包如下：

1. mininet按如下命令生成3个交换机连接线性拓扑，控制器设为remote

```
sudo mn --topo linear,3 --controller remote
```

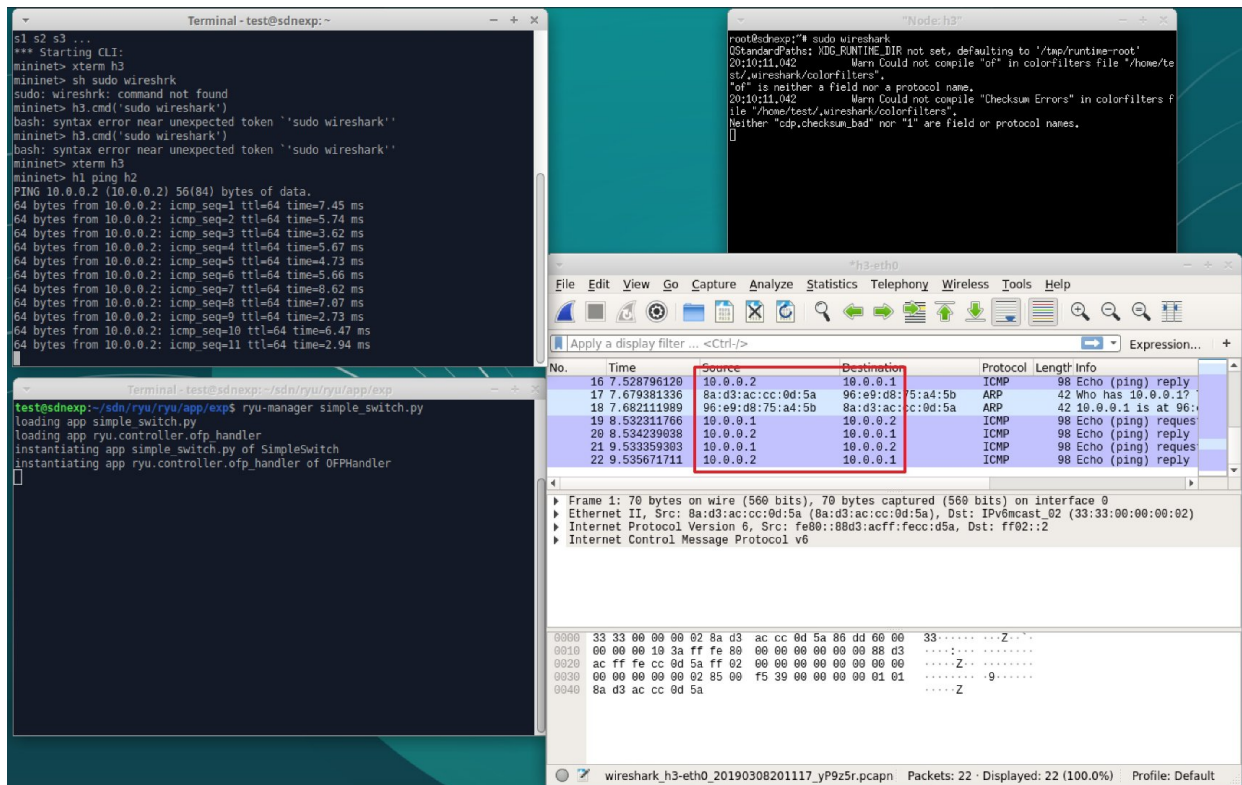
2. 将上面代码保存为 `simple_switch.py`，启动控制器

```
sudo ryu-manager simple_switch.py
```

3. 在mininet的CLI中打开h3的xterm，启动wireshark抓取端口 `h3-eth0`

```
mininet> xterm h3  
  
root@sdnexp:~# sudo wireshark
```

4. 在mininet中 `h1 ping h2`，查看wireshark中关于 `h3` 的抓包情况



从抓包结果中，我们可以发现该交换机的实现中明显的缺点：

在 `packet_in_handler` 中将数据包洪泛到交换机的所有端口，故 `h1` 和 `h2` 通讯时，`h3` 也会收到所有的包

对于更实际的情况，交换机端口更多时，数据包的洪泛将造成网络的拥堵，严重影响网络性能

我们将在后面的实验中由各位改进这一点

## 3 实验内容

第一次实验主要为验证性实验，要求各位在学习第2部分内容的基础上完成下面两道题目

### 3.1 实验1

#### 题目

选取自己名字中的任意一个字，按照汉字的结构生成mininet拓扑并连接Ryu，通过Ryu的GUI界面截取拓扑图

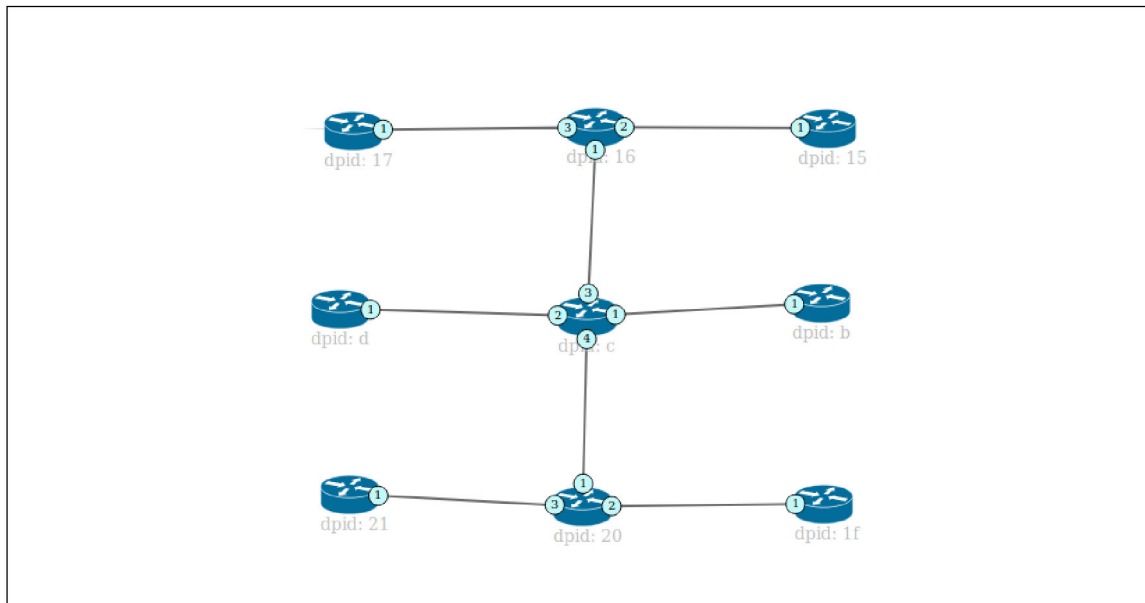
#### 说明

- 可选择其他汉字，笔画适中、结构简单即可
- 尽量不选择环路(“回”)，断路(“二”)等结构的汉字，或者灵活处理使其能够连通且不成环
- 验收
  - 当场验收：请指导老师查看运行结果和代码即可
  - 提交报告：报告中需包括姓名学号、代码、拓扑截图及总结，命名格式 `cs60-小胖-exp11.pdf`

#### 示例

- “王”

## Ryu Topology Viewer



## 3.2 实验2

### 题目

在2.5.3实现的交换机基础之上实现二层自学习交换机，避免数据包的洪泛

SDN环境下，二层自学习交换机的学习策略可以理解为：对于每个交换机，我们可以学习收到的数据包mac和交换机port的映射，进而下发流表以指导包的转发，从而避免向所有port洪泛

为避免各位不熟悉Ryu的API，我们在2.5.3的基础上补充了做了一些补充

```
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 LearningSwitch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(LearningSwitch, self).__init__(*args, **kwargs)
        # maybe you need a global data structure to save the mapping

    def add_flow(self, datapath, priority, match, actions):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
```

```

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
actions)]
        mod = parser.OFPFlowMod(datapath=dp, priority=priority,
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
    # 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
    # :)

    out = parser.OFPPacketOut(
        datapath=dp, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=msg.data)
    dp.send_msg(out)

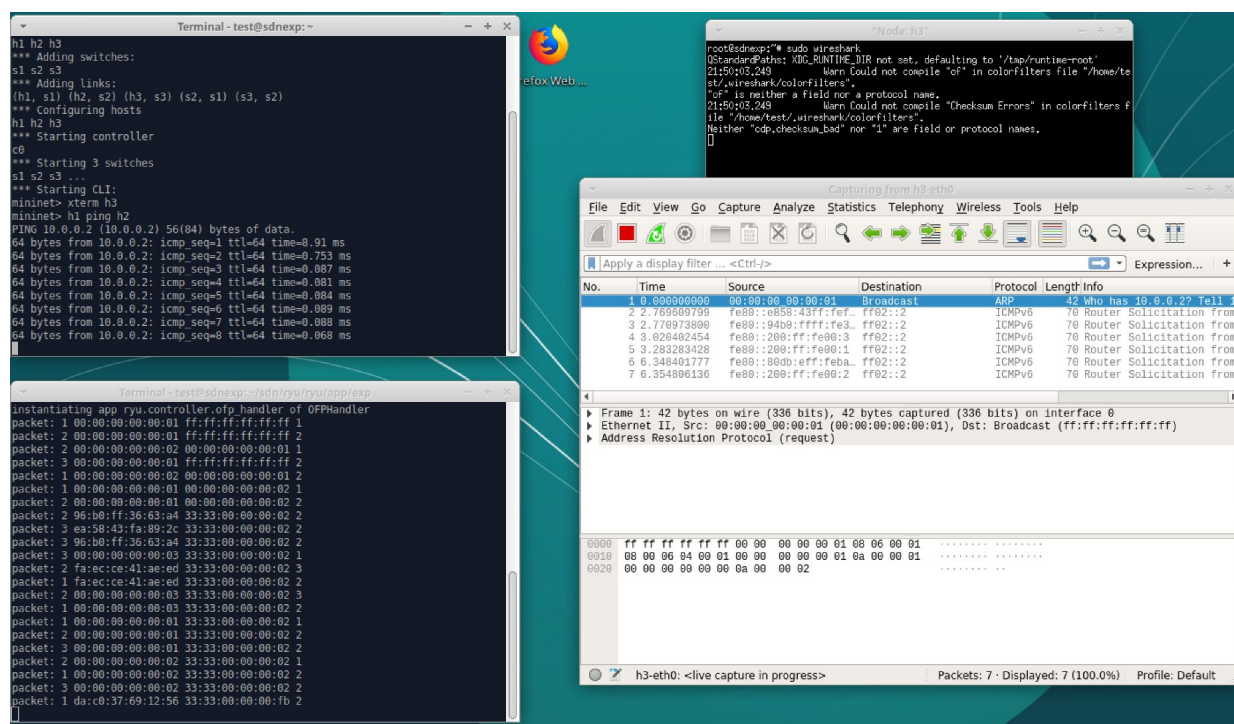
```



## 说明

- 可不考虑交换机对数据包的缓存
- 对比你实现的自学习交换机和2.5.3中的交换机（即在2.5.3的拓扑中，h1 和 h2 的通讯不再转发到 h3 ）
- 验收
  - 当场验收：请指导老师查看运行结果和代码，讲解如何避免洪泛
  - 提交报告：报告中需包括姓名学号、代码、wireshark抓包截图及总结，命名格式 cs60-小胖-exp12.pdf

## 示例



h3-eth0 端口不再受到 h1 和 h2 通讯的影响

## 4 总结

希望本次实验大家能掌握以下知识点：

- 能够通过流表和抓包分析SDN中的网络流
- 利用Mininet的API自定义网络拓扑
- Ryu API的基本使用
- 二层自学习交换机的基本原理

## 5 扩展资料

- sdn论坛：[sdnlab](#)
- 关于Mininet的更多资料：[Mininet Doc](#), [Mininet API](#)
- 关于Ryu APP开发的更多资料：[Ryu Book](#)
- SDN网络的部署案例：[Google P4](#)