SDN Experiment 1

实验环境

- 网络模拟器

  Mininet

- 控制器

  Ryu

- 抓包工具

  Wireshark/tcpdump

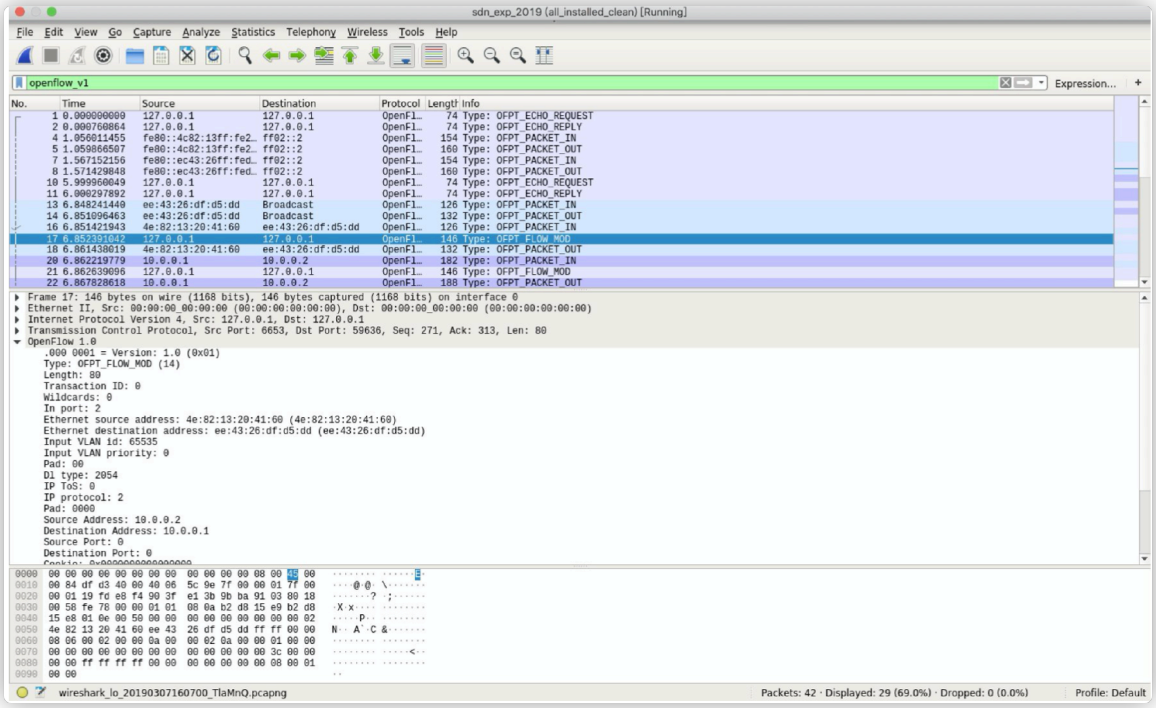
Wireshark

在网络实验中，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



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

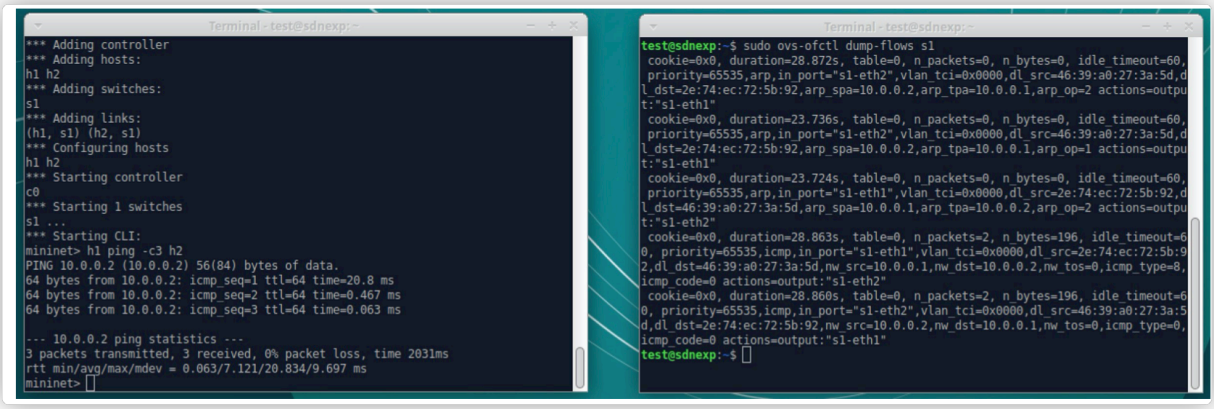
- 常用命令

```
# 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
```

- 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
```



- 简单的写法

示例位于`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
```

- 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
```

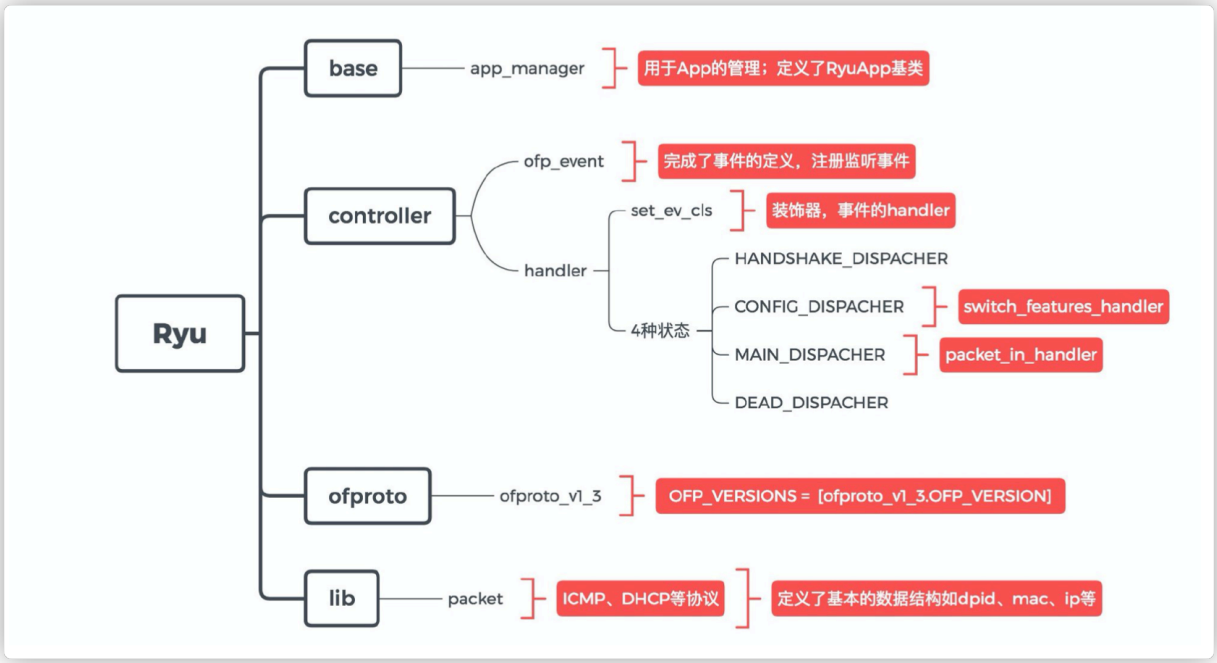
Ryu

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

- Ryu的源码结构

```
.
├── 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



- 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

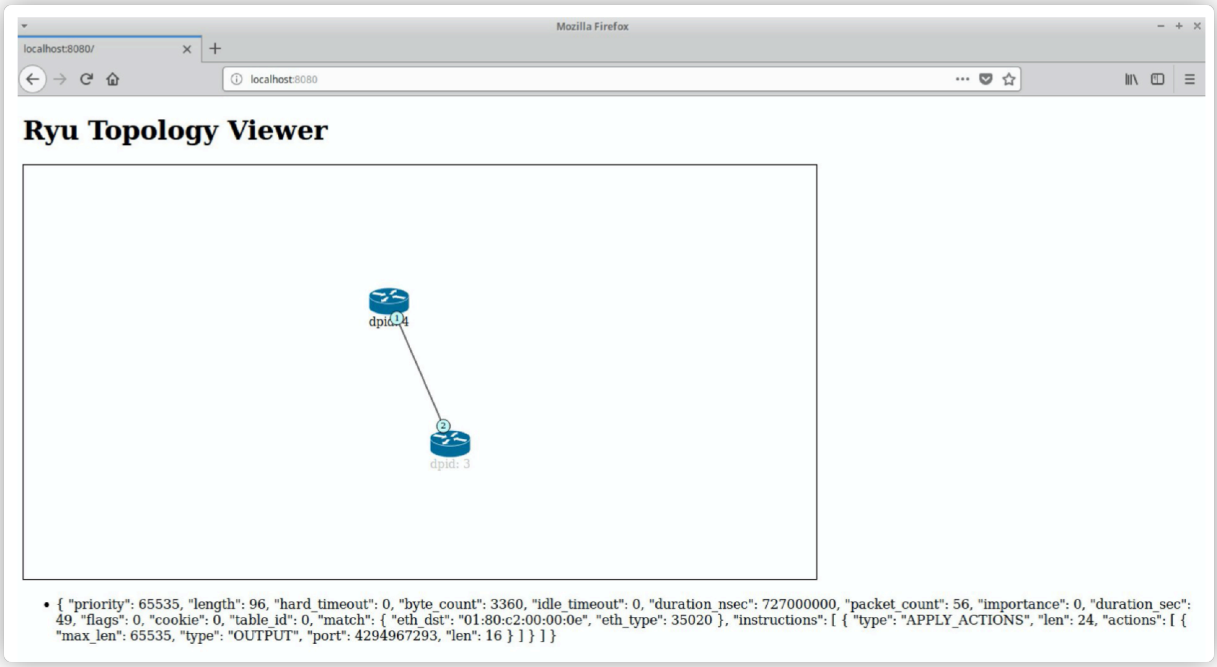
```
cd ~/sdn/mininet/custom

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

2. 启动Ryu提供的APP：

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



- 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
```

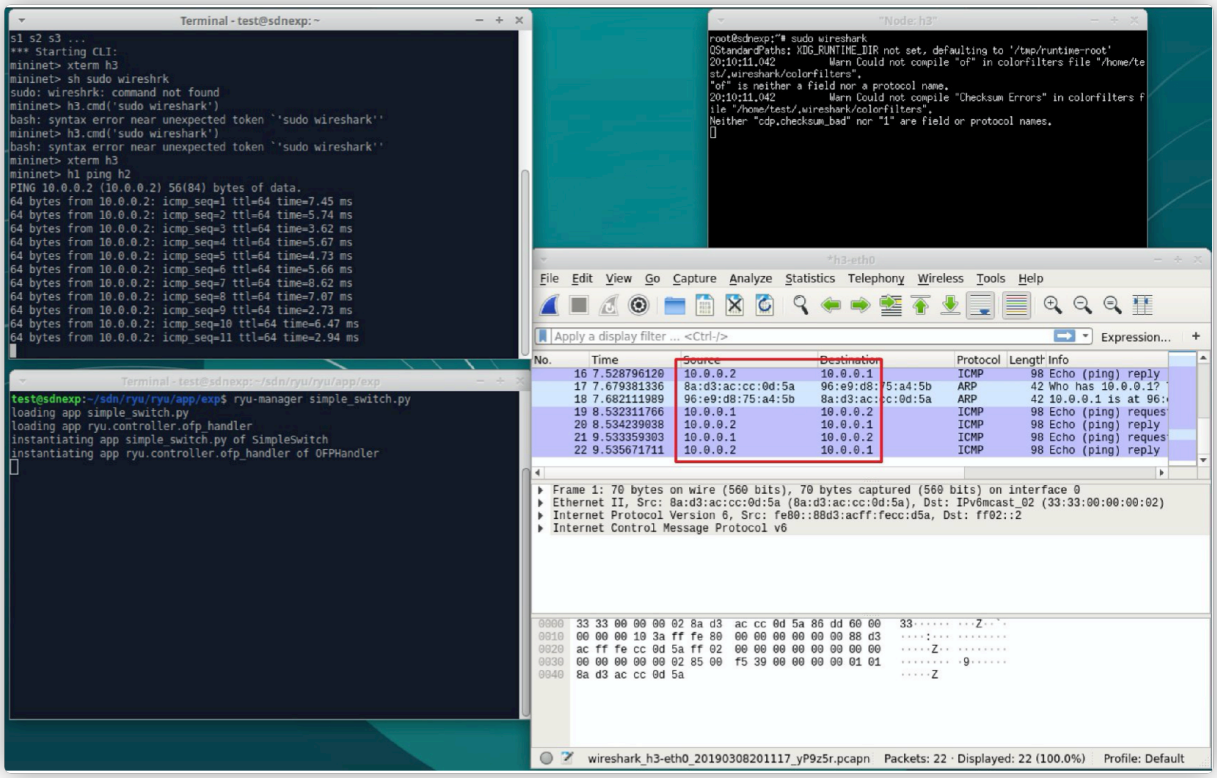
2. 将上面代码保存为`simple_switch.py`，启动控制器

```
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

我们将在第二次的实验中由各位改进这一点

实验内容

第一次实验主要为验证性实验，帮助各位在熟悉Mininet，Ryu及Wireshark等工具的使用，大家可在前面说明的基础上完成下面题目

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的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，我们给出了框架，大家只需补充关键的若干行实现即可

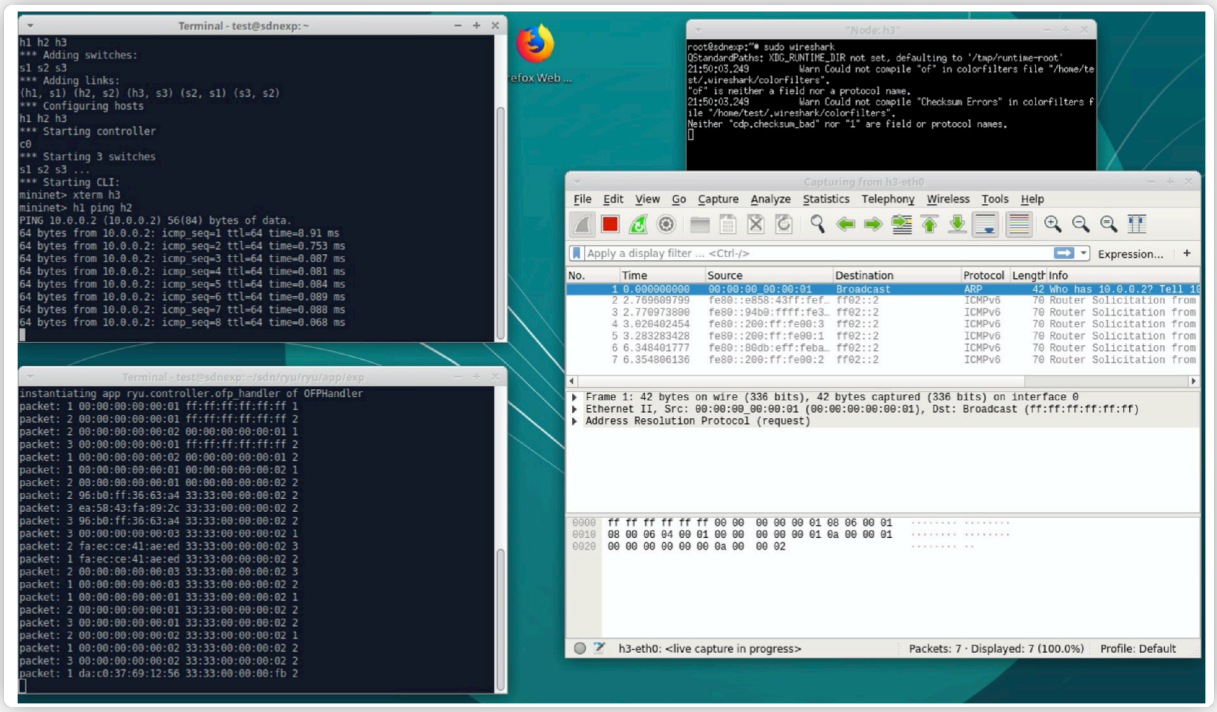
```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

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

示例



`h3-eth0`端口不再受到`h1`和`h2`通讯的影响

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网络中的数据流
- 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

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