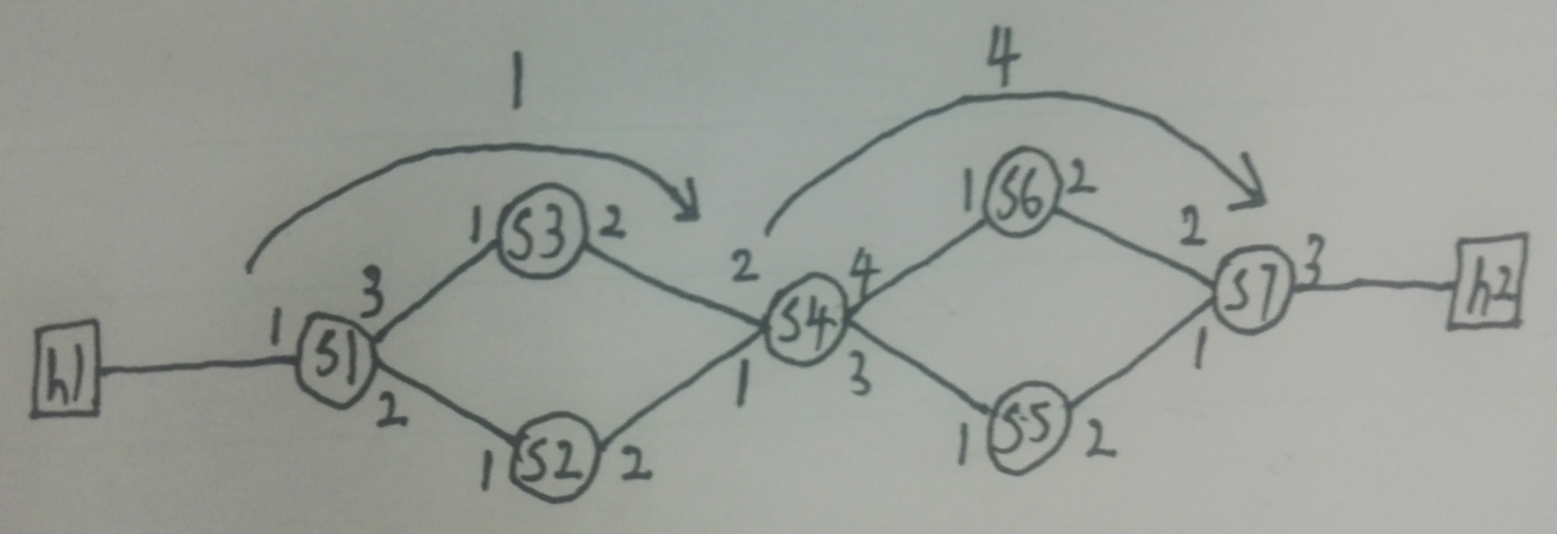
# Paco协议新流实验报告

## 一、实验设计

实验拓扑如下图所示（跟之前的实验一样），不同的是每个交换机都再与控制器建立一个连接。



实验中会创建一个h3来模拟控制器，每个交换机都与控制器建立连接

新流的操作只在s1处实现（可以在每个交换机都实现，这里是当时实验的设计），也就是说对s2~s7的操作不做修改（之后在资源说明中会说明如何做）

## 二、paco协议功能验证

2.1 进行实验

实验所需资源在”新流实验.zip”中

资源介绍（前10条黑色字体的内容均与验证实验相同，红色字体为需要修改的内容，从11条开始为新增的内容）

* 1. build.sh

用来编译p4代码生成中间表示json文件，其中前两行bmv2和p4c-bm的路径需要自行设置

* 1. run\_demo.sh

用来生成此实验环境的，其中前两行也需要像build.sh里面一样自行配置

此实验环境的运行是采用的python脚本的方式，在run\_demo.sh里面运行了topo.py，topo.py稍后会介绍

* 1. cleanup

每次做完实验后清理残留程序

* 1. topo.txt（与验证实验作用相同，但需要添加每个交换机与控制器的连接）

实验拓扑信息，第一行switches后面跟数字代表拓扑中交换机的数目，第二行同理代表主机的数目。剩下每一行代表一条链路，例如h1 s1代表h1和s1之间有一条链路。由topo.py读取建立实验拓扑

* 1. p4src文件夹

里面存放了p4代码

* 1. topo.py

用于自动生成实验环境的python脚本

里面调用了P4Switch，P4Host，为P4官方的P4交换机和P4主机

自定义类MyTopo在Mininet的Topo类上添加了P4相关内容

read\_topo函数实现了从topo.txt中读取拓扑信息

main函数中实现了读取并建立拓扑结构，执行bmv2 CLI并下发表项（第110行开始的for循环）（需要执行新流下发操作的交换机，例如本次的s1交换机，这里不下发相应的流表，需要做相应的修改）。下发表项的命令存放在commands文件夹中，供topo.py读取，每个交换机的命令单独存放，详细的bmv2 CLI的命令格式见附录1，这里举个例子，比如下图的命令，前面table\_add代表添加表项，forward\_ipv4为表项的名字，ipv42paco代表匹配以后执行的动作，=>之前是匹配的内容，回到之前的代码中可以看到forward\_ipv4表项中匹配的是源、目的IP地址的精确值，所以这里有两个参数分别带别源IP地址和目的IP地址，=>后面为执行动作时候用到的参数，回到代码中可以看到ipv42paco动作有两个参数，所以这里也有两个参数



* 1. commands文件夹（与验证实验相同，但是需要执行新流操作的交换机的相关命令不会被topo.py执行）

用来存放各个交换机的表项下发命令文件，由topo.py读取，需要说明的是每个交换机都要对应一个文件，即使不需要下发任何命令，也要有一个空文件

* 1. send.py

在建立好实验拓扑之后，在虚拟主机上发送数据，使用的是python库scapy

验证实验是无限发送数据包的，这里需要根据一次实验的发送次数调整相关代码，并且每次发送数据包的ip地址是不同的（由于要触发新流的操作），详情见代码

* 1. receive.py

在建立好实验拓扑之后，在虚拟主机上接受数据，使用的是python库scapy

比验证实验多输出了数据包的源ip地址信息

* 1. paco.json为build.sh生成的p4语言中间表示
  2. commands\_new文件夹

存放控制器收到交换机的请求以后需要下发的命令，没个ip对应一次新流操作，这些文件由creat\_files.py生成，由cpu.py调用

* 1. creat\_files.py

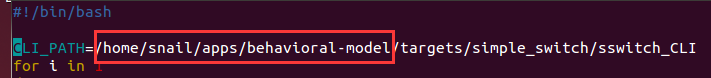
用来批量生成commands\_new中的文件，其中command变量为控制器收到交换机的请求后向交换机安装的命令，需要根据具体情况修改

* 1. default\_commands文件夹

用于存放需要执行新流操作的交换机（例如本次的s1，也可能是多个交换机）初始时的默认流表的下发命令，由default\_commands.sh下发

* 1. default\_commands.sh

下发默认流表的脚本，需要将CLI\_PATH的前半部分的bmv2路径修改为自己的bmv2路径



* 1. cpu.py控制器实现脚本

#!/usr/bin/python

from scapy.all import sniff, sendp, hexdump, DADict, IP, ICMP

from scapy.all import Packet, Ether

from scapy.all import XByteField, DestMACField, SourceMACField, XShortEnumField, XShortField, XIntField, LongField, BitField, MACField

from mininet.cli import CLI

import subprocess

import sys

import struct

#定义以太类型，添加paco协议类型

ETHER\_TYPES = DADict(\_name="/etc/ethertypes")

ETHER\_TYPES['paco'] = 0x0037

ETHER\_TYPES['IP'] = 0x0800

#定义cpu\_header头部

class Cpu(Packet):

name = "cpu"

fields\_desc = [

XByteField("device\_id",1)

]

#定义以太头部

class MyEther(Packet):

name = "MyEther"

fields\_desc = [

DestMACField("dst"),

SourceMACField("src"),

XShortEnumField("ethertype", 0x0800, ETHER\_TYPES)

]

#定义paco头部

class Paco(Packet):

name = "paco"

fields\_desc = [

XIntField("ids",0x0),

XShortField("ori\_ethertype",0x0800)

]

#安装流表，其中的bm\_cli变量前半部分的bmv2路径需要修改为自己的bmv2路径

def install\_table(device\_id, ip\_number):

command\_file = "commands\_new/" + ip\_number + ".txt"

bm\_cli = "/home/snail/apps/behavioral-model/tools/runtime\_CLI.py"

json = "paco.json"

cmd = [bm\_cli, "--json", json,

"--thrift-port", str(22222 + device\_id - 1)]

with open(command\_file, "r") as f:

try:

output = subprocess.check\_output(cmd, stdin = f)

print output

except subprocess.CalledProcessError as e:

print e

print e.output

#分析数据包

def deal\_ip(pkt):

#将抓到的数据包转换为string类型

pkt\_str = str(pkt)

#解析以太报头并验证

# MyEther

myether = MyEther(pkt\_str[1:])

mac\_dst = myether.sprintf("%MyEther.dst%")

mac\_src = myether.sprintf("%MyEther.src%")

ethertype = myether.sprintf("%MyEther.ethertype%")

if ethertype != "IP":

return

#解析IP报头并验证

# IP

ip = IP(pkt\_str[15:])

ip\_dst = ip.sprintf("%IP.dst%")

ip\_src = ip.sprintf("%IP.src%")

ip\_number = ip\_src[7:]

raw = ip.sprintf("%Raw.load%")[1:-1]

if ip\_src[:7] != '10.0.0.' or ip\_dst != '10.0.0.2':

return

print "s1 receive."

install\_table(1, ip\_number)

#将原数据包去掉cpu\_header头部重新发回交换机

# send

interface = "s1-eth3"

sendp(MyEther(dst=mac\_dst,src=mac\_src,ethertype=ETHER\_TYPES[ethertype]) / IP(dst=ip\_dst,src=ip\_src) / ICMP() / raw, iface=interface)

def handle\_pkt(pkt):

pkt\_str = str(pkt)

#检测device\_id，如果不在范围内，则不做处理，’\x07’代表7（16进制）

if pkt\_str[0] > '\x07' or pkt\_str[0] < '\x01':

return

#提取device\_id

device\_id = struct.unpack("b",pkt\_str[0])[0]

#根据device\_id做相应的处理

if device\_id == 1:

deal\_ip(pkt)

else:

return

def main():

#在s1的3端口（s1与控制器相连的端口）抓包，如果抓到包，将数据包交给handle\_pkt()函数去处理

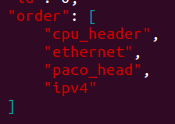
sniff(iface="s1-eth3",prn = lambda x: handle\_pkt(x))

if \_\_name\_\_ == '\_\_main\_\_':

main()

实验步骤：

1. 运行build.sh编译p4程序，编译成功后，编辑paco.p4：搜索order，将order下面的内容改成如下图所示，如果已经一样，则不用修改



1. 运行run\_demo.sh搭建实验环境
2. 新开一个终端，运行default\_commands.sh安装默认流表
3. 运行cpu.py启动控制器
4. 在mininet的CLI中输入xterm h1 h2会分别弹出主机h1和主机h2的控制台
5. 在主机h2上运行reveive.py,在主机h1上运行send.py，如果成功，几秒后h2的控制台会有信息输出。
6. 分析数据包是否按照预期的方式被处理

此处同验证实验，打开wireshark抓包分析即可

1. 实验完成后，关闭wireshark，ctrl+c停止控制器脚本，关闭h1和h2的控制台，然后在mininet的CLI界面输入ctrl+D结束mininet，之后再输入sudo mn –c