# Paco协议验证实验报告

## 一、如何实现paco协议机制

在以太帧中有协议字段，我们定义该字段的值为0x0037的时候为paco协议。

当一个交换机收到一个以太数据包的时候，首先检查协议字段，如果协议字段为0x0800，则说明为IPv4协议，这时我们匹配源、目的IP地址，根据匹配到的流表执行添加paco头以及对应的pathlet\_ids字段的值，然后指定出端口。

如果协议字段是0x0037，说明是paco协议，这时我们匹配pathlet\_ids字段，根据匹配结果以及交换机位置执行相关操作。其中，指定出端口是所有交换机都要执行的操作。分段倒数第二个交换机额外执行pathlet\_ids字段移位操作，以实现pop出前一个标签，暴露出下一个标签。若是最后一个分段，那么分段倒数第二个交换机不执行移位操作，而是执行删除paco头部的操作。

p4中实现paco协议的代码说明如下（/\* \*/包含的部分为注释）：

/\*

定义paco协议头部格式，pathlet\_ids字段由路径中经过的段id按顺序组成，ori\_etherType为收到的以太帧中原始etherType字段的值

\*/

header\_type paco\_head\_t {

fields {

pathlet\_ids: 32;

ori\_etherType: 16;

}

}

/\*实例化一个paco协议头部\*/

header paco\_head\_t paco\_head;

/\*

定义paco协议与控制器（cpu）交互时所需的头部，device字段用来标识当前设备id

\*/

header\_type cpu\_header\_t {

fields {

device: 8;

}

}

/\*实例化一个cpu\_header实例\*/

header cpu\_header\_t cpu\_header;

/\*

定义以太帧头部格式，dstAddr字段为目的MAC地址，srcAddr字段为源MAC地址，etherType字段为协议类型

\*/

header\_type ethernet\_t {

fields {

dstAddr : 48;

srcAddr : 48;

etherType : 16;

}

}

/\*实例化一个以太帧头部实例\*/

header ethernet\_t ethernet;

/\*

定义ipv4协议头部格式，srcAddr为源IP地址，dstAddr为目的IP地址

\*/

header\_type ipv4\_t {

fields {

version : 4;

ihl : 4;

diffserv : 8;

totalLen : 16;

identification : 16;

flags : 3;

fragOffset : 13;

ttl : 8;

protocol : 8;

hdrChecksum : 16;

srcAddr : 32;

dstAddr: 32;

}

}

/\*实例化一个ipv4协议头部\*/

header ipv4\_t ipv4;

/\*

解析程序入口

\*/

parser start {

/\*

根据接收到的数据，检测从第0位开始的128位数据，如果为0，那么转到parse\_cpu\_header，否则转到parse\_ethernet

\*/

return select(current(0,128)) {

0 : parse\_cpu\_header;

default : parse\_ethernet;

}

}

/\*

解析cpu\_header

\*/

parser parse\_cpu\_header {

/\*按照定义的cpu\_header的格式，将当前数据解析到前面实例化的cpu\_header实例中\*/

extract(cpu\_header);

/\*进入ingress\*/

return ingress;

}

/\*这里类似与C语言，宏定义\*/

#define ETHERTYPE\_IPV4 0x0800

#define ETHERTYPE\_PACO 0x0037

#define CPU\_MIRROR\_SESSION\_ID 250

/\*以太帧解析\*/

parser parse\_ethernet{

extract(ethernet);

return select(ethernet.etherType){

ETHERTYPE\_IPV4: parse\_ipv4;

ETHERTYPE\_PACO: parse\_paco;

}

}

/\*ipv4解析\*/

parser parse\_ipv4 {

extract(ipv4);

return ingress;

}

/\*paco解析\*/

parser parse\_paco {

extract(paco\_head);

return ingress;

}

/\*

动作：指定下一跳的出口

\*/

action next\_hop(output\_port) {

/\*

修改metadata里面egress\_spec域的值为output\_port

\*/

modify\_field(standard\_metadata.egress\_spec, output\_port);

}

/\*

动作：ipv4转换到paco

\*/

action ipv42paco(ids, output\_port){

/\*给当前数据添加头部\*/

add\_header(paco\_head);

modify\_field(paco\_head.ori\_etherType, ethernet.etherType);

modify\_field(ethernet.etherType, 0x0037);

modify\_field(paco\_head.pathlet\_ids, ids);

modify\_field(standard\_metadata.egress\_spec, output\_port);

}

/\*

动作：段中间转发，位于段中间的路由器转发paco分组时执行的动作

\*/

action pathlet\_mid\_forward(output\_port) {

modify\_field(standard\_metadata.egress\_spec, output\_port);

}

/\*

动作：匹配到pathlet\_ids为0之后执行的转发动作

\*/

action pathlet\_NULL\_forward(output\_port) {

modify\_field(standard\_metadata.egress\_spec, output\_port);

modify\_field(ethernet.etherType, paco\_head.ori\_etherType);

remove\_header(paco\_head);

}

/\*

动作：段末尾转发，位于段末尾的路由器转发时的动作

\*/

action pathlet\_tail\_forward(output\_port) {

modify\_field(standard\_metadata.egress\_spec, output\_port);

/\*shift\_left(p1,p2,p3)：将p2的数据左移p3位赋值给p1\*/

shift\_left(paco\_head.pathlet\_ids, paco\_head.pathlet\_ids, 8);

}

/\*

Binding-ID

\*/

action pathlet\_multi\_forward(number, new\_ids, output\_port){

/\*bit\_and(p1,p2,p3)：将p2与p3的值赋值给p1。与是指bit与\*/

bit\_and(paco\_head.pathlet\_ids, paco\_head.pathlet\_ids, 0x00FFFFFF);

/\*shift\_right与shift\_left类似\*/

shift\_right(paco\_head.pathlet\_ids, paco\_head.pathlet\_ids, 8 \* number);

/\*bit\_or与bit\_and类似\*/

bit\_or(paco\_head.pathlet\_ids, paco\_head.pathlet\_ids, new\_ids);

modify\_field(standard\_metadata.egress\_spec, output\_port);

}

/\*

域列表

\*/

field\_list copy2cpu\_fields {

standard\_metadata;

}

/\*

动作：复制到cpu，将当前数据复制一份发送给控制器

\*/

action copy2cpu() {

/\*我也不太懂其中的原理\*/

clone\_ingress\_pkt\_to\_egress(CPU\_MIRROR\_SESSION\_ID, copy2cpu\_fields);

}

action forward() {

/\*什么都不执行，但是不能删，还不知道原因\*/

}

/\*

动作：添加cpu\_header的头部，参数device\_id为发送该数据包的设备id

\*/

action do\_cpu\_encap(device\_id) {

add\_header(cpu\_header);

modify\_field(cpu\_header.device, device\_id);

}

/\*

Ipv4转发表：reads里面列出的是匹配项，以ipv4.srcAddr : exact;为例，意思是匹配ipv4的源ip地址，格式为exact（这里是具体值，还有其他的，比如lmp，举个例子，0

x11223344/4代表与0x11223344的前4位匹配），actions为动作列表，为匹配到数据包以后，可能执行的动作列表

\*/

table forward\_ipv4 {

reads {

ipv4.srcAddr : exact;

ipv4.dstAddr : exact;

}

actions {

ipv42paco;

next\_hop;

copy2cpu;

}

}

table forward\_paco{

reads {

paco\_head.pathlet\_ids : lpm;

}

actions{

pathlet\_mid\_forward;

pathlet\_tail\_forward;

pathlet\_NULL\_forward;

pathlet\_multi\_forward;

copy2cpu;

}

}

table redirect {

reads {

standard\_metadata.instance\_type : exact;

}

actions {

forward;

do\_cpu\_encap;

}

}

/\*

入口控制，根据数据包中解析出的数据，决定哪些table起作用

\*/

control ingress {

if (ethernet.etherType == ETHERTYPE\_PACO){

apply(forward\_paco);

}

if (ethernet.etherType == ETHERTYPE\_IPV4){

apply(forward\_ipv4);

}

}

/\*出口控制。我也不太懂\*/

control egress {

apply(redirect);

}

/\*

附加说明：p4代码中写了很多actions和table，但是某个具体的路由器可能只会用到某个或者某几个，但是之所以要写全，是因为所有路由器用的都是同一份p4代码

\*/

## 二、paco协议功能验证

实验拓扑如图2-1所示。

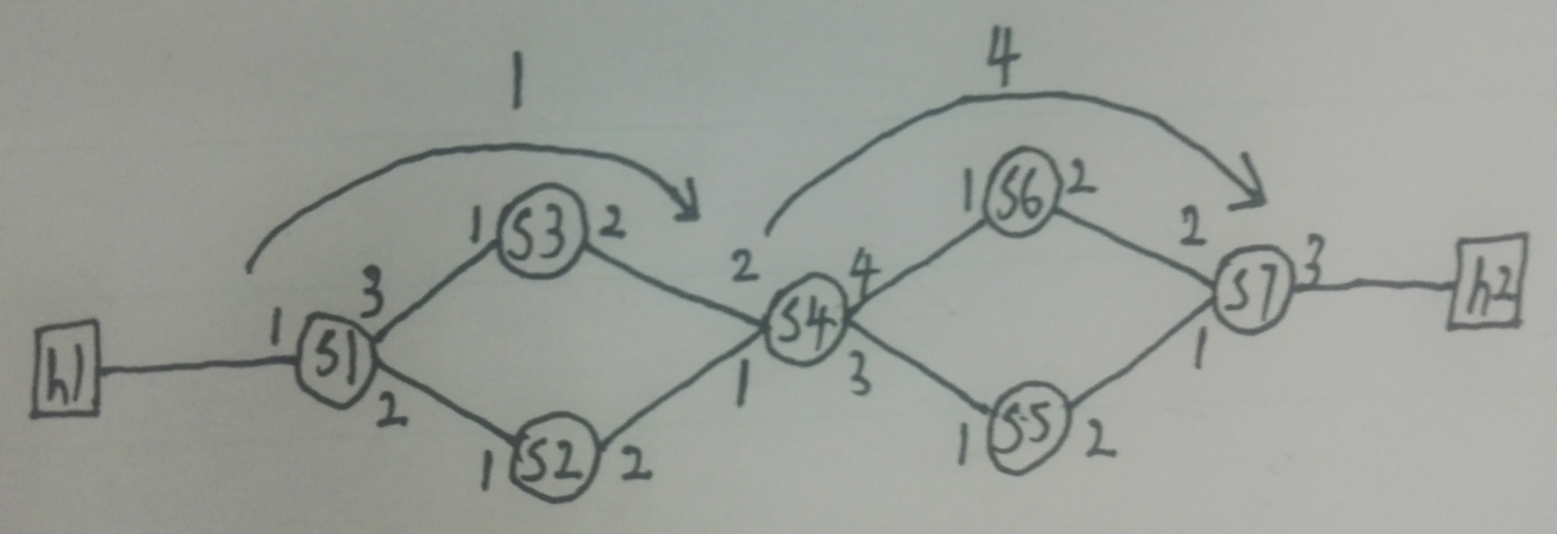


图2-1 实验拓扑

2.1 实验环境搭建

本实验在Ubuntu14.04 LTS下进行。

注：前两步也可以参考下面的链接自行搭建

https://github.com/p4lang/tutorials/tree/master/SIGCOMM\_2015#obtaining-required-software

（1）下载安装bmv2

$ git clone <https://github.com/p4lang/behavioral-model.git>

$ cd behavioral-model

$ git checkout 1.2.0

$ ./install\_deps.sh

$ ./autogen.sh

$ ./configure

$ make

$ [sudo] make install

测试：

$ make check

（2）下载安装p4c-bm

$ git clone <https://github.com/p4lang/p4c-bm.git>

$ cd p4c-bm

$ git checkout 1.2.0

$ sudo pip install -r requirements.txt

$ sudo python setup.py install

（3）下载安装mininet

$ git clone <https://github.com/mininet/mininet.git>

$ cd mininet

$ git checkout 2.2.1

$ ./util/install.sh

（4）下载安装OpenVswitch

$ wget <http://openvswitch.org/releases/openvswitch-2.4.0.tar.gz>

$ tar -zxvf openvswitch-2.4.0.tar.gz

$ cd openvswitch-2.4.0

检查已有版本并删除，注意每个人的版本可能不同

$ lsmod | grep openvswitch

$ rmmod openvswitch

$ find / -name openvswitch.ko –print

$ rm /lib/modules/3.5.0-17-generic/extra/openvswitch.ko

编译安装ovs

$ sh boot.sh

$ ./configure --with-linux=/lib/modules/`uname -r`/build

$ make

$ make install

$ make modules\_install

$ /sbin/modprobe openvswitch

$ mkdir -p /usr/local/etc/openvswitch

$ ovsdb-tool create /usr/local/etc/openvswitch/conf.db vswitchd/vswitch.ovsschema 2>/dev/null

$ ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock \

--remote=db:Open\_vSwitch,Open\_vSwitch,manager\_options \

--private-key=db:Open\_vSwitch,SSL,private\_key \

--certificate=db:Open\_vSwitch,SSL,certificate \

--bootstrap-ca-cert=db:Open\_vSwitch,SSL,ca\_cert \

--pidfile --detach

$ ovs-vsctl --no-wait init

$ ovs-vswitchd --pidfile --detach

这样安装完，以后每次启动要执行最后三条命令

2.2 进行实验

实验所需资源在”验证实验.zip”中

资源介绍

* 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

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）。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

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

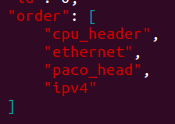
在建立好实验拓扑之后，在虚拟主机上发送数据，使用的是python库scapy

* 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

* 1. paco.json为build.sh生成的p4语言中间表示

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. 等mininet启动换成后在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
3. 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。
4. 分析数据包是否按照预期的方式被处理

通过wireshark抓包分析paco头部是否被正确添加或者正确处理。

Wireshark抓包验证：

在实验环境搭建完成以后，启动wireshark，这里有一点要说明，所有主机和交换机的虚拟网卡号是按照topo.txt中该链路出现的顺序从1依次编号的。选择某个主机或交换机需要抓包的虚拟网卡，开始抓包，在过滤栏中输入icmp or llc后回车，之后会有很多包被抓到，需要根据具体情况去分析。

以下是针对图2-1拓扑进行实验的分析，其他实验类似：

图2-2为使用wireshark对s1的1端口抓包获得的数据包，我们可以看到这是一个未经过处理的正常的Ethernet/IP/ICMP数据包。

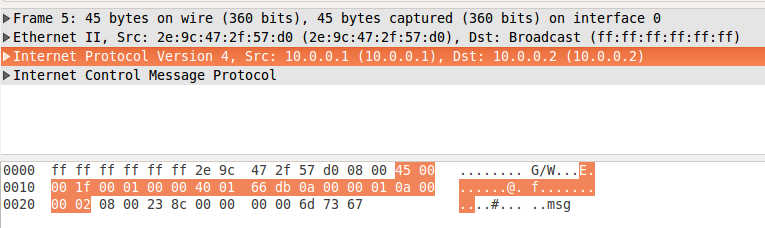


图2-2 wireshark对s1的1端口抓取的数据包

图2-3为使用wireshark对s1的3端口抓包获取的数据包，我们可以看到，与图2-2相比，多出了红线划出的部分，这部分正是我们添加的tag头部，其中前32bits为tags字段，值为初始值1、4、0、0，与预期的相同，后16bits为ori\_etherType字段。而原来以太帧的字段值变为了paco协议的值0x0037

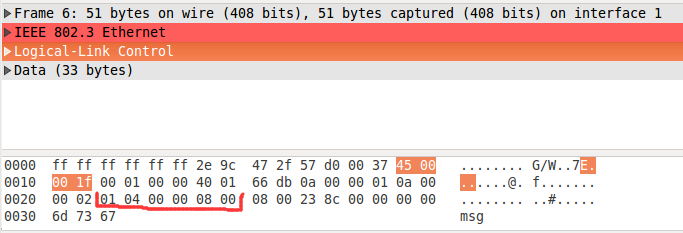


图2-3 wireshark对s1的3端口抓取的数据包

图2-4为使用wireshark对s3的2端口抓包获取的数据包，我们可以看到，pathlet\_ids字段值变为了4、0、0、0，与预期相同。

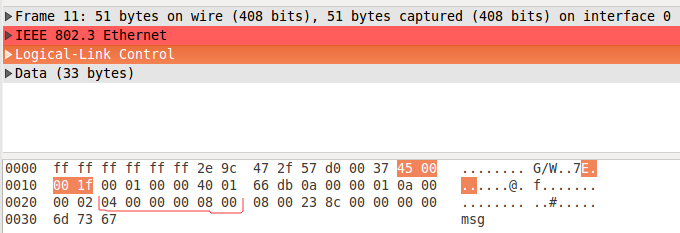


图2-4 wireshark对s3的2端口抓取的数据包

图2-5为使用wireshark对s4的4端口抓包获取的数据包，pathlet\_ids字段的值为4、0、0、0，与预期的相同。

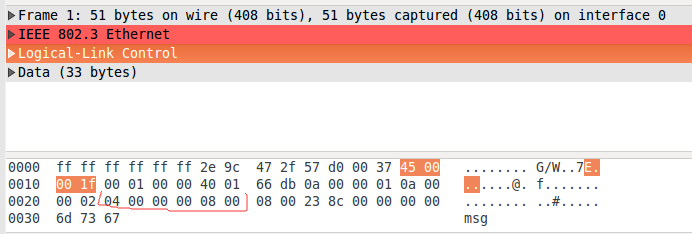


图2-5 wireshark对s4的4端口抓取的数据包

图2-6为使用wireshark对s6的2端口抓包获取的数据包，pathlet\_ids字段的值为0、0、0、0，与预期的相同。

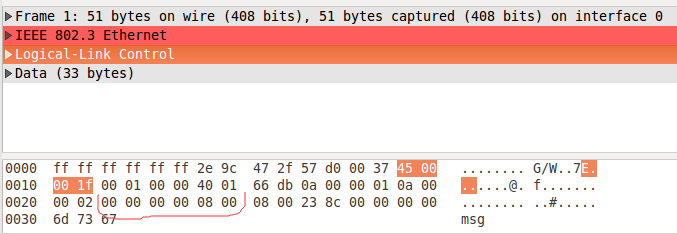


图2-6 wireshark对s6的2端口抓取的数据包

图2-7为使用wireshark对s7的3端口抓包获取的数据包，现在，paco头已经被去掉，又恢复了入口处的数据包。

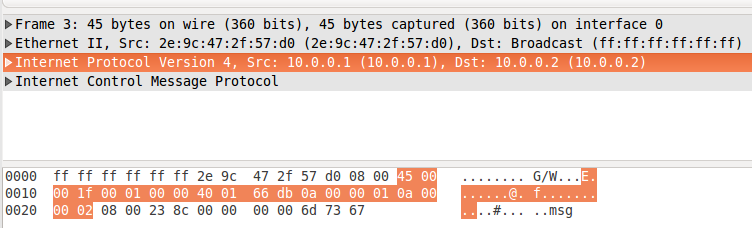


图2-7 wireshark对s7的3端口抓取的数据包

至此，整个功能被成功验证。

1. 实验完成后，关闭wireshark，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

附录1 bmv2 CLI命令

1. 启动bmv2 CLI：

$ python /path/to/bmv2/tools/runtime\_CLI.py --json source\_routing.json --thrift-port 22222

两个参数--json 指定p4代码的中间表示json文件。--thrift-port指定要操作的p4交换的thrift端口。

2. 查看命令：

在bmv2 CLI里可以通过help命令查看所有支持的操作。

使用help table\_add 可以查看table\_add 的具体语法，其他命令也一样。