计算机网络实验四

静态路由的编程实现

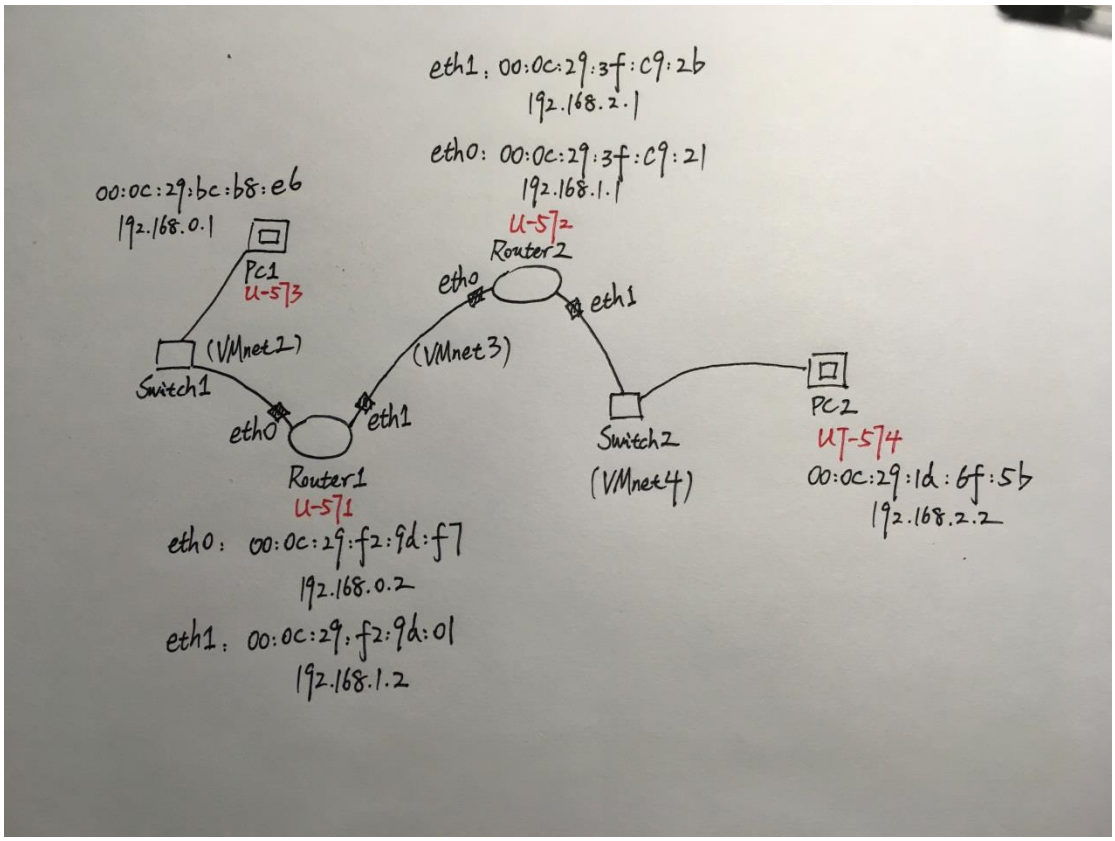
邱梓豪

141130077

1.实验目的

本实验主要目的是设计和实现一个简单的静态路由机制，用以取代 Linux 实现的静态路由方式，进而加深对二三层协议衔接及静态路由的理解。

2.网络拓扑结构



注：每个网卡的 MAC 地址是通过 ifconfig 得到的，而 IP 地址并不是利用 ifconfig 设置的，是写死在程序中的。

3.相关数据结构的定义

(1)静态路由表

```
struct route_item{
    char dest[16];              //  目的 ip
    char gateway[16];           //  网关 ip
    char netmaskl[16];          //  子网掩码
    char interface[16];         //   端口
};
```

(2)ARP 表

```
struct arp_table{
    char ip_addr[16];           // ip 地址
    char mac_addr[18];          // mac 地址
}
```

(3)路由器信息表

```
struct device_info{
    char interface[14];         //  端口
    char mac_addr[18];          //  端口对应 IP
}
```

4.相关表项的配置

Router1 的表项配置：

```
device_info = {'eth0':'00:0c:29:f2:9d:f7','eth1':'00:0c:29:f2:9d:01'}
arp_table = {'192.168.1.1':'00:0c:29:3f:c9:21',
             '192.168.0.1':'00:0c:29:bc:b8:e6'}
routing_table = {'192.168.2.2':['192.168.1.1','255.255.255.0','eth1'],
             '192.168.0.1':['192.168.0.1','255.255.255.0','eth0']}
```

Router2 的表项配置：

```
device_info = {'eth0':'00:0c:29:3f:c9:21','eth1':'00:0c:29:3f:c9:2b'}
arp_table = {'192.168.2.2':'00:0c:29:1d:6f:5b',
             '192.168.1.2':'00:0c:29:f2:9d:01'}
routing_table = {'192.168.2.2':['192.168.2.2','255.255.255.0','eth1'],
             '192.168.0.1':['192.168.1.2','255.255.255.0','eth0']}
```

5.程序运行说明

在本实验中，myPing.py 运行在两台主机上，static_router.py 运行在两台路由器上。两台主机的 ip、mac，两台路由器的 ip、mac 都被写死在程序中。python 程序的运行使用 python2.x，运行时在终端下输入的命令为：sudo python 程序名

mac 地址在程序中被写在这个位置，如果想运行，必须要根据设备的情况修改该源和目的 mac 地址

```
# build frame header, which contains dst mac, src mac and protocol type(IPv4)
# you should this the following mac addr based on your device
frameHeader = struct.pack("!6s6s2s",'\x00\x0c\x29\x3f\xc9\x2b','\x00\x0c\x29\x1d\x6f\x5b','\x08\x00')
```

6.代码说明：

a) myPing.py，程序如下：

```
import socket, struct, select, time

ICMP_ECHO_REQUEST = 8
IP_VERSION = 4

# function to calculate checksum of ip head(copy from blog)
def ip_headchecksum(ip_head):
        checksum = 0
        headlen = len(ip_head)
        i = 0
        while i<headlen:
                temp = struct.unpack("!H", ip_head[i:i+2])[0]
                checksum += temp
                i += 2
        checksum = (checksum>>16) + (checksum&0xffff)
        checksum += checksum>>16

        return (~checksum)&0xffff
```

首先是对一些常量的定义。ip_headchecksum()函数的功能是计算

当前 ip 头（checksum 字段为 0）的校验和

```
# function to calculate checksum of icmp(copy from github)
def checksum(source_string):
    """
    I'm not too confident that this is right but testing seems
    to suggest that it gives the same answers as in_cksum in ping.c
    """
    sum = 0
    countTo = (len(source_string)/2)*2
    count = 0
    while count<countTo:
        thisVal = ord(source_string[count + 1])*256 + ord(source_string[count])
        sum = sum + thisVal
        sum = sum & 0xffffffff # Necessary?
        count = count + 2

    if countTo<len(source_string):
        sum = sum + ord(source_string[len(source_string) - 1])
        sum = sum & 0xffffffff # Necessary?

    sum = (sum >> 16)  +  (sum & 0xffff)
    sum = sum + (sum >> 16)
    answer = ~sum
    answer = answer & 0xffff

    # Swap bytes. Bugger me if I know why.
    answer = answer >> 8 | (answer << 8 & 0xff00)

    return answer
```

checksum()函数是计算当前 icmp 数据报（checksum 字段为 0）的

校验和。

```
# crear raw socket
rawSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))
rawSocket.bind(("eth0", socket.htons(0x0800)))  # change nerwork card if you need
# build frame header, which contains dst mac, src mac and protocol type(IPv4)
frameHeader = struct.pack("!6s6s2s",'\x00\x0c\x29\xf2\x9d\xf7','\x00\x0c\x29\xbc\xb8\xe6','\x08\x00')
# build IP header
ihl_version = (IP_VERSION << 4) + 5 # Version + Header Length
tos = 0                             # Type Of Service
totalLen = 28                       # Total Length
idMark = 0                          # fragment id
offset = 0                          # fragment offset
ttl = 64                            # Time To Live
proto = 1                           # Protocol (1 means ICMP)
checkSum = 0                        # Check Sum
saddr = socket.inet_aton("192.168.0.1")      # src ip, change if you need
daddr = socket.inet_aton("192.168.2.2")       # dst ip, change if you need
ipHeader = struct.pack("!BBHHHBBH4s4s", ihl_version, tos, totalLen,
                                        idMark, offset, ttl, proto, checkSum, saddr, daddr)
ipHeader = struct.pack("!BBHHHBBH4s4s", ihl_version, tos, totalLen,
                                        idMark, offset, ttl, proto, ip_headchecksum(ipHeader), saddr, daddr)
# build ICMP packet
icmp = struct.pack("!BBHHH", 8, 0, 0, 0, 0)  # ping request
icmp = struct.pack("!BBHHH", 8, 0, checksum(icmp), 0, 0)
# build and send the whole packet
packet = frameHeader + ipHeader + icmp
rawSocket.send(packet)
```

之后先创建一个 socket，该 socket 绑定在端口“eth0”。然后依次构建出以太网帧头，ip 头和 icmp 数据报。

```
# build frame header, which contains dst mac, src mac and protocol type(IPv4)
frameHeader = struct.pack("!6s6s2s",'\x00\x0c\x29\xf2\x9d\xf7','\x00\x0c\x29\xbc\xb8\xe6','\x08\x00')
```

以太网帧头部三个成员分别是目的 mac 地址，源 mac 地址和上层协议类型，这里用 python 的 pack 函数进行包装。

```
# build IP header
ihl_version = (IP_VERSION << 4) + 5 # Version + Header Length
tos = 0                             # Type Of Service
totalLen = 28                       # Total Length
idMark = 0                          # fragment id
offset = 0                          # fragment offset
ttl = 64                            # Time To Live
proto = 1                           # Protocol (1 means ICMP)
checkSum = 0                        # Check Sum
saddr = socket.inet_aton("192.168.0.1")      # src ip, change if you need
daddr = socket.inet_aton("192.168.2.2")       # dst ip, change if you need
ipHeader = struct.pack("!BBHHHBBH4s4s", ihl_version, tos, totalLen,
                                        idMark, offset, ttl, proto, checkSum, saddr, daddr)
ipHeader = struct.pack("!BBHHHBBH4s4s", ihl_version, tos, totalLen,
                                        idMark, offset, ttl, proto, ip_headchecksum(ipHeader), saddr, daddr)
```

接下来是创建 ip 头，各字段的含义见程序中的注释。源 ip 设置为当前主机，目的 ip 设置为另一端的主机。

```
# build ICMP packet
icmp = struct.pack("!BBHHH", 8, 0, 0, 0, 0)  # ping request
icmp = struct.pack("!BBHHH", 8, 0, checksum(icmp), 0, 0)
# build and send the whole packet
packet = frameHeader + ipHeader + icmp
rawSocket.send(packet)
```

组装 icmp 数据报，“8”表示为 icmp request。最后将以太网帧头部，ip 首部，icmp 数据报连接起来便得到最后的 packet，发送此 packet。

```
# receive the icmp packet
recSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))

while True:
        packet = recSocket.recvfrom(65565)

        #packet string from tuple
        packet = packet[0]

        #parse ethernet header
        eth_length = 14

        eth_header = packet[:eth_length]
        eth = struct.unpack('!6s6sH' , eth_header)
        eth_protocol = socket.ntohs(eth[2])
        print 'Destination MAC : ' + eth_addr(packet[0:6]) + ' Source MAC : ' + eth_addr(pack
(eth_protocol)

        #Parse IP packets, IP Protocol number = 8
        if eth_protocol == 8 :
                #Parse IP header
                #take first 20 characters for the ip header
                ip_header = packet[eth_length:20+eth_length]

                #now unpack them :)
                iph = struct.unpack('!BBHHHBBH4s4s' , ip_header)

                version_ihl = iph[0]
                version = version_ihl >> 4
                ihl = version_ihl & 0xF

                iph_length = ihl * 4
```

之后再建立一个 socket 进行监听，探测到有数据包则对数据包进行解析。

如果抓到 icmp 数据包，则对其进行解析，看是不是对之前 icmp 请求的回应。

```
                iph_length = ihl * 4

                ttl = iph[5]
                protocol = iph[6]
                s_addr = socket.inet_ntoa(iph[8]);
                d_addr = socket.inet_ntoa(iph[9]);

                print 'Version : ' + str(version) + ' IP Header Length : ' + str(ihl) + ' TTL : ' + str(ttl) + ' Protocol : '
(protocol) + ' Source Address : ' + str(s_addr) + ' Destination Address : ' + str(d_addr)

                #ICMP Packets
                if protocol == 1 :
                        print 'Get icmp reply!!!!'
                        u = iph_length + eth_length
                        icmph_length = 4
                        icmp_header = packet[u:u+4]

                        #now unpack them :)
                        icmph = struct.unpack('!BBH' , icmp_header)

                        icmp_type = icmph[0]
                        code = icmph[1]
                        checksum = icmph[2]

                        print 'Type : ' + str(icmp_type) + ' Code : ' + str(code) + ' Checksum : ' + str(checksum)

                        h_size = eth_length + iph_length + icmph_length
                        data_size = len(packet) - h_size
                else:
                        print 'Other protocol'
```

Trash

在另一台主机上运行的 myPing 和这个大致相同，只不过这个包是 reply 包（type=0 表示 reply），为了简便，在监听 socket 是仅对 icmp 数据包做出相应：

```
rSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))
rSocket.bind(("eth0", socket.htons(0x0800)))
frameHeader = struct.pack("!6s6s2s", '\x00\x0c\x29\x3f\xc9\x2b','\x00\x0c\x29\x
saddr=socket.inet_aton("192.168.2.2")
daddr=socket.inet_aton("192.168.0.1")
ipHeader = struct.pack("!BBHHHBBH4s4s",(4<<4)+5,0,28,0,0,64,1,0,saddr,daddr)
ipHeader = struct.pack("!BBHHHBBH4s4s",(4<<4)+5,0,28,0,0,64,1,ipcheck(ipHeader)
icmp = struct.pack("!BBHHH",0,0,0,0,0)
icmp = struct.pack("!BBHHH",0,0,checksum(icmp),0,0)
packet_reply = frameHeader+ipHeader+icmp

while True:
        packet = lSocket.recvfrom(2048)
        packet = packet[0]
        ipheader = packet[14:20+14]
        iph = struct.unpack('!BBHHHBBH4s4s', ipheader)
        if iph[6]==1:
                rSocket.send(packet_reply)



^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

b) static_router.py 代码如下：

首先利用 python 的字典结构定义了三张表，供路由时查询。

eth_addr()函数的功能是将 mac 地址的数值转换成字符串表示。

```
# -*- coding: utf-8 -*-
import socket
import struct
import binascii

device_info = {'eth0':'00:0c:29:f2:9d:f7','eth1':'00:0c:29:f2:9d:01'}
arp_table = {'192.168.1.1':'00:0c:29:3f:c9:21',
                '192.168.0.1':'00:0c:29:bc:b8:e6'}
routing_table = {'192.168.2.2':['192.168.1.1','255.255.255.0','eth1'],
                '192.168.0.1':['192.168.0.1','255.255.255.0','eth0']}

def eth_addr(a):
        b="%.2x:%.2x:%.2x:%.2x:%.2x:%.2x"%(ord(a[0]),ord(a[1]),ord(a[2]),ord(a[3]),ord(a[4]),ord(a[5]))
        return b
```

```
listenSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))
while True:
        packet = listenSocket.recvfrom(65565)
        packet = packet[0]
        eth_length = 14
        eth_header = packet[:eth_length]
        eth = struct.unpack('!6s6sH', eth_header)
        eth_prot = socket.ntohs(eth[2])
        dst_mac = eth_addr(packet[0:6])
        src_mac = eth_addr(packet[6:12])
        print 'Des mac: '+dst_mac+' Src mac: '+src_mac+' Ptotocol: '+str(eth_prot)
        if dst_mac == device_info['eth0'] or dst_mac == device_info['eth1']:
                print 'The packet was sent to ME '
                if eth_prot == 8:
                        print 'It\'s a IP protocol'
                        ip_header = packet[eth_length:20+eth_length]
                        iph = struct.unpack('!BBHHHBBH4s4s', ip_header)
                        s_addr = socket.inet_ntoa(iph[8])
                        d_addr = socket.inet_ntoa(iph[9])
                        print 'Src ip: '+str(s_addr)+' Dst ip: '+str(d_addr)
                        gateway = routing_table[str(d_addr)][0]
                        eth = routing_table[str(d_addr)][2]
                        new_dst_mac = arp_table[gateway]
                        new_src_mac = device_info[eth]
                        print 'new dst mac: '+new_dst_mac
                        print 'new src mac: '+new_src_mac
                        eth_header = struct.pack('!6s6s2s', binascii.unhexlify(new_dst_mac.replace(':','')), binascii.unhexli
(new_src_mac.replace(':','')), '\x08\x00')

                        sendSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))
                        sendSocket.bind((eth, socket.htons(0x0800)))
                        new_packet = eth_header + packet[eth_length:]
                        bytes = sendSocket.send(new_packet)
                        print 'Have transmit '+str(bytes)+' bytes'
        else:
                print 'It\'s NOT my packet'
```

之后建立一个 socket，并对往来的数据包进行相应的处理。首先从数据包的帧头中拿出源 mac 地址和目的 mac 地址，如果目的 mac 地址是自己的话则进行进一步处理，否则丢弃此包。

如果此包的网络层协议是 ip 的话，则进一步从 ip 头中取出源 ip 和目的 ip。查询路由表可得目的 ip 的网关 gateway 及对应的端口 eth。则路由器可通过查询 arp 表得知 gateway 的 mac 地址，这就是此数据包的新目的 mac 地址；路由器可通过查询 device_info 得知端口的 mac，这就是新源 mac 地址。

```
gateway = routing_table[str(d_addr)][0]
eth = routing_table[str(d_addr)][2]
new_dst_mac = arp_table[gateway]
new_src_mac = device_info[eth]
print 'new dst mac: '+new_dst_mac
print 'new src mac: '+new_src_mac
```

然后重新构造以太网帧头，和数据包的剩余部分组成新的 packet，在对应端口上发送即可。

```
                    eth_header = struct.pack('!6s6s2s', binascii.unhexlify(new_dst_mac.replace(':','')), binascii.unhexlify
(new_src_mac.replace(':','')), '\x08\x00')

                    sendSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))
                    sendSocket.bind((eth, socket.htons(0x0800)))
                    new_packet = eth_header + packet[eth_length:]
                    bytes = sendSocket.send(new_packet)
                    print 'Have transmit '+str(bytes)+' bytes'
```

7.运行结果：

Ip 设置，由于无法将 ip 设为全 0，所以我将所有 ip 设为 0.0.0.1：

Router1：

```
user@ubuntu:~/Downloads$ sudo ifconfig eth0 0.0.0.1 netmask 255.255.255.0
user@ubuntu:~/Downloads$ sudo ifconfig eth1 0.0.0.1 netmask 255.255.255.0
user@ubuntu:~/Downloads$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:f2:9d:f7
          inet addr:0.0.0.1  Bcast:0.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fef2:9df7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9106 errors:0 dropped:0 overruns:0 frame:0
          TX packets:956 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:842126 (842.1 KB)  TX bytes:80895 (80.8 KB)
          Interrupt:19 Base address:0x2000

eth1      Link encap:Ethernet  HWaddr 00:0c:29:f2:9d:01
          inet addr:0.0.0.1  Bcast:0.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fef2:9d01/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13316 errors:0 dropped:0 overruns:0 frame:0
          TX packets:974 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1098546 (1.0 MB)  TX bytes:83666 (83.6 KB)
          Interrupt:19 Base address:0x2080

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:689 errors:0 dropped:0 overruns:0 frame:0
          TX packets:689 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:50456 (50.4 KB)  TX bytes:50456 (50.4 KB)
```

Router2：

```
user@ubuntu:~/Downloads$ sudo ifconfig eth0 0.0.0.1 netmask 255.255.255.0
user@ubuntu:~/Downloads$ sudo ifconfig eth1 0.0.0.1 netmask 255.255.255.0
user@ubuntu:~/Downloads$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:3f:c9:21
          inet addr:0.0.0.1  Bcast:0.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe3f:c921/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9200 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5220 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:848648 (848.6 KB)  TX bytes:343988 (343.9 KB)
          Interrupt:19 Base address:0x2000

eth1      Link encap:Ethernet  HWaddr 00:0c:29:3f:c9:2b
          inet addr:0.0.0.1  Bcast:0.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe3f:c92b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:427296 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1001 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:25932557 (25.9 MB)  TX bytes:87385 (87.3 KB)
          Interrupt:19 Base address:0x2080

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:706 errors:0 dropped:0 overruns:0 frame:0
          TX packets:706 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:51706 (51.7 KB)  TX bytes:51706 (51.7 KB)
```

PC1：

```
user@ubuntu:~/Downloads$ sudo ifconfig eth0 0.0.0.1 netmask 255.255.255.0
user@ubuntu:~/Downloads$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:bc:b8:e6
          inet addr:0.0.0.1  Bcast:0.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:febc:b8e6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9242 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1003 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:851533 (851.5 KB)  TX bytes:86819 (86.8 KB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:692 errors:0 dropped:0 overruns:0 frame:0
          TX packets:692 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:50788 (50.7 KB)  TX bytes:50788 (50.7 KB)
```

PC2：

```
user@ubuntu:~$ sudo ifconfig eth0 0.0.0.1 netmask 255.255.255.0
user@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:1d:6f:5b
          inet addr:0.0.0.1  Bcast:0.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe1d:6f5b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8991 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5903355 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:832105 (832.1 KB)  TX bytes:247945408 (247.9 MB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

PC1：

运行 myPing.py 之后收到一个 icmp 数据包，type=0 说明是 reply。源主机正是另一端的主机，目的主机是自己。

```
user@ubuntu:~/Downloads$ sudo python myPing.py
[sudo] password for user:
Destination MAC : 00:0c:29:bc:b8:e6 Source MAC : 00:0c:29:f2:9d:f7 Protocol : 8
Version : 4 IP Header Length : 5 TTL : 64 Protocol : 1 Source Address : 192.168.
2.2 Destination Address : 192.168.0.1
Get icmp reply!!!!
Type : 0 Code : 0 Checksum : 65535
Destination MAC : 00:00:00:00:00:00 Source MAC : 00:00:00:00:00:00 Protocol : 8
Version : 4 IP Header Length : 5 TTL : 64 Protocol : 17 Source Address : 127.0.0
.1 Destination Address : 127.0.0.1
Other protocol
```

Router1：

可以看到这次 ping 的过程中这个路由器交换了 2 个数据包，一来一回。

```
Des mac: 00:0c:29:f2:9d:f7 Src mac: 00:0c:29:bc:b8:e6 Ptotocol: 8
The packet was sent to ME
It's a IP protocol
Src ip: 192.168.0.1 Dst ip: 192.168.2.2
new dst mac: 00:0c:29:3f:c9:21
new src mac: 00:0c:29:f2:9d:01
Have transmit 60 bytes
Des mac: 00:0c:29:f2:9d:01 Src mac: 00:0c:29:3f:c9:21 Ptotocol: 8
The packet was sent to ME
It's a IP protocol
Src ip: 192.168.2.2 Dst ip: 192.168.0.1
new dst mac: 00:0c:29:bc:b8:e6
new src mac: 00:0c:29:f2:9d:f7
Have transmit 60 bytes
```

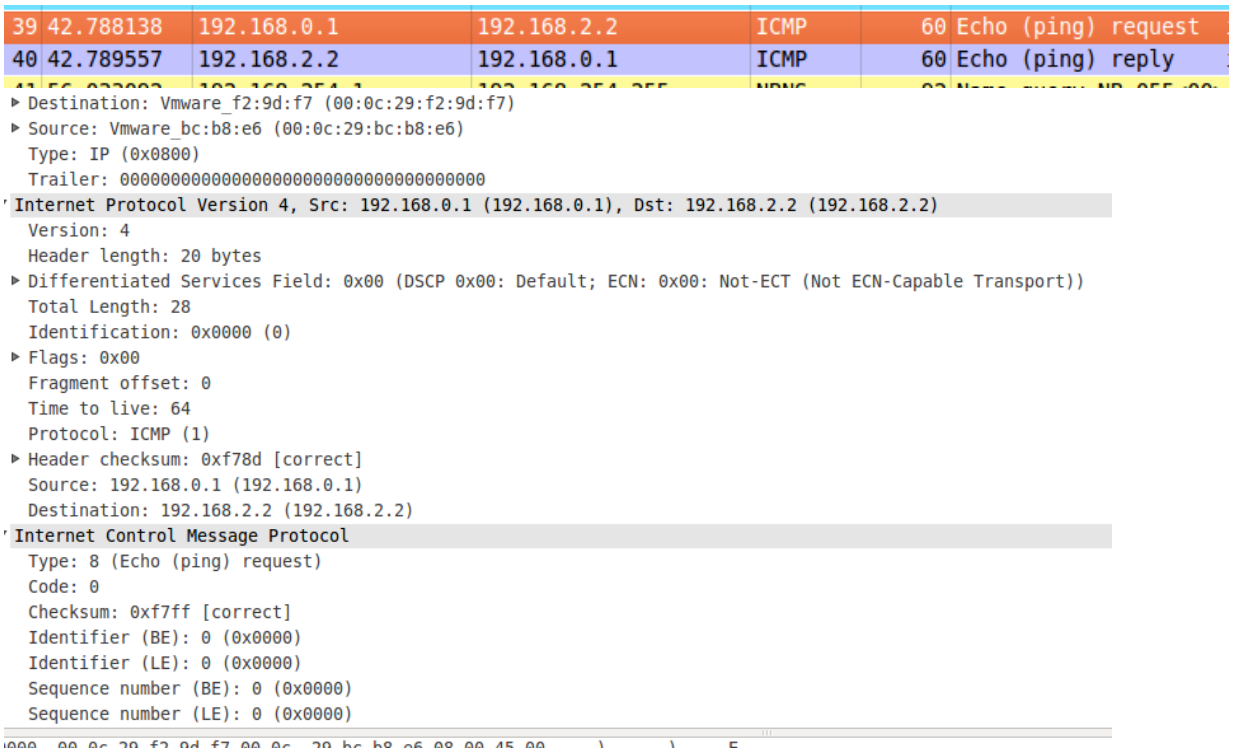
Router2：

情况和 Router1 相似。

```
Des mac: 00:0c:29:3f:c9:21 Src mac: 00:0c:29:f2:9d:01 Ptotocol: 8
The packet was sent to ME
It's a IP protocol
Src ip: 192.168.0.1 Dst ip: 192.168.2.2
new dst mac: 00:0c:29:1d:6f:5b
new src mac: 00:0c:29:3f:c9:2b
Have transmit 60 bytes
Des mac: 00:0c:29:3f:c9:2b Src mac: 00:0c:29:1d:6f:5b Ptotocol: 8
The packet was sent to ME
It's a IP protocol
Src ip: 192.168.2.2 Dst ip: 192.168.0.1
new dst mac: 00:0c:29:f2:9d:01
new src mac: 00:0c:29:3f:c9:21
Have transmit 60 bytes
Des mac: 00:00:00:00:00:00 Src mac: 00:00:00:00:00:00 Ptotocol: 8
It's NOT my packet
Des mac: 00:00:00:00:00:00 Src mac: 00:00:00:00:00:00 Ptotocol: 8
It's NOT my packet
```

8.Wireshark 抓包结果：

再次用 pc1 ping pc2



可以看到，包中填充的内容和我事先填入的内容是一致的。

9.个人思考

通过这次实验，我对路由的过程及其中二三层协议的衔接又有了进一步的认识。我觉得代码仍可以进行改进，比如可以用 python 自动获取本机的 mac，而不用写死在程序中。