路由器程序设计

学号:2111252　　姓名:李佳豪　时间:12.15

路由器程序设计
- 一、实验内容
- 二、测试环境
- 三、程序功能、界面
- 四、程序设计
  - 1、首先考虑核心功能
  - 2、路由表：链表，顺序查找
  - 3、ARP缓存表： 哈希（u_int --> mac），便于直接查找
  - 4、数据包缓冲区：队列，push和pop
  - 5、日志系统
  - 6、Device： 设置全局的pcap_t ahandle
  - 7、捕获线程，流程正如上面说过的：
  - 8、转发线程
  - 9. 命令系统
- 五、过程验证
- 六、问题与思考

一、实验内容

二、测试环境

仓库链接：https://github.com/FondH/NetTech/tree/main/Rooter

网络拓扑如下，R1为自己书写的路由程序，R2启用Window路由功能；PC0和PC1则为两个网段下的设备

```
/*
206.1.1.2       206.1.1.1      206.1.2.1     206.1.2.2   206.1.3.1       206.1.3.2
   PC0      -->   inc0  - [R1] - inc1       -->      [R2]         -->       PC1
*/
```

在模拟上述拓扑时，主机内网下使用四台虚拟机host-only进行组网，其中R1使用win10系统（debug效率更高），其余则仍然使用实验给的win 2003；

三、程序功能、界面

1. 手动维护路由表：通过add、del指令对路由表项管理；print指令输出。

```
route add  `ip`  `mask`  `next-hop`
route del  `ip`  `mask`  `next-hop`
route print
```

2. 转发：对接受到的IP层数据进行转发；转发时涉及对路由表查询以及动态获取下一跳MAC地址。由下指令查看；

```
log print trans
```

3. 日志系统：通过print指令查看路由器捕获、转发等行为； dump保存到本地；

```
log print `filter`  # filter可选 sys 、trans、cap、send
log dump `filename`
```

4. ArpCache、Packetqueue

```
arp-cache  #查看当前arp-cache
packet-queue  #查看当前数据包缓冲区
```

```
@fondH/>
@fondH/>
@fondH/>arp-chche

Unknown command.
@fondH/>arp-cache


ArpCache:
============================================================
IP              MAC             是否有效
206.1.2.2 -- > 00-0c-29-1b-c0-76  Va
206.1.1.2 -- > 00-0c-29-5d-0c-bc  Va
@fondH/>
@fondH/>
@fondH/>
@fondH/>
@fondH/>
@fondH/>
@fondH/>route print


IPv4 Rooting Table
============================================================
网络目标        掩码            下一跳地址      接口
0.0.0.0         0.0.0.0         0.0.0.0         1
206.1.1.0       255.255.255.0   0.0.0.0         0
206.1.2.0       255.255.255.0   0.0.0.0         1
206.1.3.0       255.255.255.0   206.1.2.2       1
@fondH/>
```

```
@fondH/>log print
log entrys:
============================================================
| 3:28:57|2| [CAPTURE] <ICMP> src_ip: 206.1.1.2  dst_ip: 206.1.3.2
            src_mac: 00-0c-29-5d-0c-bc      dst_mac : 00-0c-29-c9-bd-0a
| 3:28:57|0| [SEND] <Arp>  src_ip: src_ip: 206.1.2.1  dst_ip: 206.1.2.2
            src_mac: 00-0c-29-c9-bd-0a      dst_mac : 00-00-00-00-00-00
| 3:28:57|0| [SEND] <Arp>  src_ip: src_ip: 206.1.2.1  dst_ip: 206.1.2.2
            src_mac: 00-0c-29-c9-bd-0a      dst_mac : 00-00-00-00-00-00
| 3:28:58|0| [CAPTURE] <Arp>  src_ip: src_ip: 206.1.2.2  dst_ip: 206.1.2.1
            src_mac: 00-0c-29-1b-c0-76      dst_mac : 00-0c-29-c9-bd-0a
| 3:28:58|0| [CAPTURE] <Arp>  src_ip: src_ip: 206.1.2.2  dst_ip: 206.1.2.1
            src_mac: 00-0c-29-1b-c0-76      dst_mac : 00-0c-29-c9-bd-0a
| 3:28:58|0| [TRANS] <IP >src_ip:206.1.1.2 -> 206.1.2.2 -> 206.1.3.2
            src_mac: 00-00-00-00-28-f3      dst_mac : ad-02-28-f3-ad-02
| 3:28:59|3| [CAPTURE] <ICMP> src_ip: 206.1.3.2  dst_ip: 206.1.1.2
            src_mac: 00-0c-29-1b-c0-76      dst_mac : 00-0c-29-c9-bd-0a
| 3:28:59|0| [SEND] <Arp>  src_ip: src_ip: 206.1.1.1  dst_ip: 206.1.1.2
            src_mac: 00-0c-29-c9-bd-0a      dst_mac : 00-00-00-00-00-00
| 3:28:59|0| [SEND] <Arp>  src_ip: src_ip: 206.1.1.1  dst_ip: 206.1.1.2
            src_mac: 00-0c-29-c9-bd-0a      dst_mac : 00-00-00-00-00-00
| 3:28:60|0| [CAPTURE] <Arp>  src_ip: src_ip: 206.1.1.2  dst_ip: 206.1.1.1
            src_mac: 00-0c-29-5d-0c-bc      dst_mac : 00-0c-29-c9-bd-0a
| 3:28:60|0| [CAPTURE] <Arp>  src_ip: src_ip: 206.1.1.2  dst_ip: 206.1.1.1
            src_mac: 00-0c-29-5d-0c-bc      dst_mac : 00-0c-29-c9-bd-0a
```

四、程序设计

1、首先考虑核心功能

- 接受数据包，过滤目的MAC非路由器本身，根据链路层Type进行拆卸网络层协议。这里仅仅考虑ARP（0x0806）和IP（0x0800）
  - 对于ARP包，若是Request进行忽视，若是require，则根据数据包内的源ip地址和源mac地址的映射关系、存入ArpCache中；而由于是在虚拟机上实现，可能收到跨网段的ARP，直接无视即可。
  - 对于IP包，由于程序实现在pc机器上，因此过滤掉目的ip自己的包；之后校验和通过后，加入全局数据包缓冲区，等转发线程进行转发；并且根据解析得到的上层协议类型、记录到日志系统中；
- 转发数据包，转发线程循环扫描当前全局数据包队列，每次从队列中pop出一个进行转发；
  - 首先根据数据包目的ip，在路由表中获得下一跳ip地址；
  - 再根据下一跳ip地址在Arp-Cache查询下一跳MAC地址，若查询失败，则主动发送Arp广播获得。
  - 最后，修改以太帧头部的源mac和目的mac字段分别为路由器mac和刚刚查询的mac地址。

因此数据结构需要全局的路由表、ARP缓存、数据包缓冲区、device、日志系统。

2、路由表：链表，顺序查找

路由表通过`vector<RouteEntry>`模拟链表，路由表的插入、删除操作；每个`RouteEntry`记录该表项的destination：目的ip地址， mask：掩码，nextHop：下一跳地址，gig：用来发送的INC序号；

注:这里声明对于本地链路上的包，其对应的下一跳地址为0，表示链路直接发送。

```
struct RouteEntry {
    uint32_t destination;
    uint32_t mask;
    uint32_t nextHop;
    int gig;

    bool  operator == (const RouteEntry& r) const  {
        return destination == r.destination && mask == r.mask && nextHop == r.nextHop;
    }
    bool operator > (const RouteEntry& r) {
        return mask > r.mask;
    }
    RouteEntry():destination(0),mask(0),nextHop(0), gig(0){}
    RouteEntry(const string& d,const string& m,const string& n,const string&
g):destination(ipToInt(d)),mask(ipToInt(m)),nextHop(ipToInt(n)),gig(stoi(g)) {}
    RouteEntry(const int& d, const int& m, const int& n, const int& g) :destination(d), mask(m),
nextHop(n), gig(g) {}

    string toString(){
        ostringstream oss;
        oss << left << setw(COLUMN_GAP) << intToIp(destination)
            << setw(COLUMN_GAP) << intToIp(mask)
            << setw(COLUMN_GAP) << intToIp(nextHop)
            <<setw(COLUMN_GAP)<<gig;

        return oss.str();
    }

};
```

```
class RouterTable {
private:
    vector<RouteEntry> routes;
    int n = 0;
public:
    //初始默认网关
    RouterTable() { n = 0; }
    RouterTable(const string d) { routes.push_back(RouteEntry("0.0.0.0","0.0.0.0","0.0.0.0",
d)); }

    // 添加路由
    bool addRoute(const RouteEntry& entry);

    // 删除路由 特定entry
    bool deleteRoute(const RouteEntry& entry);

    //匹配路由 逐条扫描、最长匹配
    RouteEntry findRoute(const uint32_t& d);

    void printTable();
    ~RouterTable() {
        routes.clear();
    }
};
```

对于插入操作，需要考虑重复和可达，即插入的entry不得于现有entry重复、且其下一跳IP在路由器链路层发送的范围内。

```
bool RouterTable::addRoute(const RouteEntry& entry) {
    //去重
    for (const auto& e : routes)
        if (entry == e)
            return 0;

    //检测下一跳地址是否可达
    if ((entry.destination & mask_INC[0]) == (ip_INC[0] & mask_INC[0]))
        return 0;
    if ((entry.destination & mask_INC[1]) == (ip_INC[1] & mask_INC[1]))
        return 0;

    routes.push_back(entry);
    return 1;
}
```

3、ARP缓存表： 哈希（u_int --> mac），便于直接查找

- ARP缓存的意义在于通过路由器查询的下一跳 32 位的 IP，找到对应的目的MAC地址，以便后面修改数据包以太帧头部的`dst_mac`字段；因此查询、更新复杂度 `O(1)`的哈希是最理想的方式。
- 正如上课学到的，ARP缓存记录的应该设置一个计时器，`ArpEntry->stime`正是起到这个作用，后面应该使用单独线程对每一个 Entry 的 stime 进行计时；不过本次实验中所有虚拟机 IP 与 MAC 的对应关系都是静态设置好的，这个计时技术并未做测验；

```
struct ArpEntry {
    u_char  DstMac[6];
    clock_t  stime;
    bool  Valid;
```

```
    string toString() {
        string s = stime < ArpEntryMaxTime ? "Va" : "Fe";
        return arrayToMac(DstMac) + "   " +  s;
    }
};

class ArpCache {

private:
    unordered_map<uint32_t, ArpEntry> cache;

public:
    //根据路由表查询的下一跳dstIp 结果存储dstmac里
    bool lookUp(const uint32_t& dstIp, u_char** mac);
    //更新的new_mac刷新
    void update(uint32_t ip, u_char* new_mac);
    void printArpAache();
    int getSize();
    ~ArpCache() {}
};
```

4、数据包缓冲区：队列，push和pop

- **队列**：当捕获线程捕获到需要转发的包，将包放入队列，转发线程每次从队列拿取、转发；队列是符合这个逻辑的；
- **mutex**：这个队列未来将被两个线程写，因此引入mutex，每一次调用push和pop将先申请mtx，由锁导致的效率问题后续分析。
- **包唯一编码**：为了将来能在日志系统中查看每个包在路由器的流动过程，对每一个包进行编码；

```
int num = 1;
int getNum() {  return num++;}
int retNum() {  return num;}
class PacketQueue {
private:
    queue<u_char*> buffer;
    unordered_map<u_char*, int> map_no;
    mutex mtx;

public:
    PacketQueue() {}
    bool push(const u_char* p, int len);
    u_char* pop();
    //得到当前队列末尾的No
    int getNo(u_char* u);
    //输出缓冲区当前未转发的包、转发效率。
    void printPacketQueue();

    ~PacketQueue() {
        while (!buffer.empty())
            this->pop();
     }
};
```

5、日志系统

- Type声明四种类型的消息，PackType则是包的类型；在程序运行适当时候构造mess插入到Loger的buffer里。便于对日志分类、整理。
- Loger将来则负责将其logerBuffer里的mess结构体进行toString打印或者dump

```
enum Type {Tsys=1,Tcap, Tsend, Ttrans};

enum PackType {Parp, Pip, Picmp, Pudp,Ptcp};

class mess {
    int no;
    time_t now;
    Type type;
    PackType packType;
    u_char src_mac[6];
    u_char dst_mac[6];
    uint32_t src_ip;
    uint32_t dst_ip;
    uint32_t trans_ip;
    string optional;
public:
   string toString();
}
class Loger {

    vector<mess> logerBuffer;
    mutex mtx;
public:
    void push(int n, Type t, PackType p, u_char* s_mac, u_char* d_mac, uint32_t s_ip, uint32_t
d_ip);
    void push(int n, Type t, PackType p, uint32_t s_ip, uint32_t d_ip, uint32_t t_ip);
    void push(string s);
    void print(int filter);
    void dump(const string& name);
};
```

6、Device： 设置全局的pcap_t ahandle

由于使用npcap的pcap_sendpacket发送数据，则需要提取获得路由器各INC对应的ahandle保存，同时记录本机的ip、mac等信息，应用直接调用Device提供的接口调用即可；

这是Device应该初始化的接口。

```
bool boot_root_INC() {
    pcap_if_t* alldevs = NULL;
    get_device_list(&alldevs, errbuf,0);
    if (!open_device(&adhandle, WIN10_NUM, alldevs, errbuf)) {
        cerr << "[Error]: INC device open defeated" << endl;
        return -1;
    }
    pcap_freealldevs(alldevs);

    string mac_string = DEFAULT_PC_MAC;
    sscanf_s(mac_string.c_str(), "%hhx-%hhx-%hhx-%hhx-%hhx-%hhx",
        &mac_INC[0][0], &mac_INC[0][1], &mac_INC[0][2],
        &mac_INC[0][3], &mac_INC[0][4], &mac_INC[0][5]);
```

```
    sscanf_s(mac_string.c_str(), "%hhx-%hhx-%hhx-%hhx-%hhx-%hhx",
        &mac_INC[1][0], &mac_INC[1][1], &mac_INC[1][2],
        &mac_INC[1][3], &mac_INC[1][4], &mac_INC[1][5]);
    cout << "INC Init... \n\n\n" ;


    ip_INC[0] = ipToInt(DEFAULT_INC0_IP);
    ip_INC[1] = ipToInt(DEFAULT_INC1_IP);
    mask_INC[0] = ipToInt("255.255.255.0");
    mask_INC[1] = ipToInt("255.255.255.0");
    return 1;
}
```

7、捕获线程，流程正如上面说过的：

- 对于pcap_next_ex捕获的数据pkt_data，首先根据以太网帧的目的MAC字段，将不是自己的数据过滤;
- 其次，若是IP协议，则先验证校验和，之后过滤IP报文头部目的IP是自己的报文，这部分报文是windows原本自己相应的，我们无法进行处理；之后则将过滤后的数据包Push进PacketQueue，相关日志信息也进行保存。
- 若是ARP协议，先过滤arp请求报文，因为这也是windows自己该响应的，无需我们再进行响应；同时过滤其他网段的ARP（非虚拟机应该不会有这个现象）。最后提取ARP报文头 IP 于 MAC对应关系，加入HASH表、保存日志信息。

```
void _rcvProc(int totalen, const u_char* pkt_data) {
    eth_header* ehtHeader = (eth_header*)pkt_data;
    uint16_t etherType = ntohs(((eth_header*)pkt_data)->eth_type);
    if (!MacIs2Self(ehtHeader->dst_mac))
        return;
    //cout << arrayToMac(ehtHeader->dst_mac)<<endl;
    if (etherType == 0x0800) {//IP
        v4Header* v4header = (v4Header*)(pkt_data + sizeof(eth_header));

        if (!Checksum(v4header))
            return;

        if (ntohl(v4header->destination_address) == ip_INC[0] || ntohl(v4header-
>destination_address) == ip_INC[1])
            return;

        packetBuffer.push(pkt_data, totalen);

        PackType p = Pip;
        switch (v4header->protocol)
        {
        case 1:
            p = Picmp;
           // cout << "ICMP \n";
            break;
        case 11:
            p = Ptcp;
           // cout << "TCP \n";
            break;
```

```
        case 17:
            p = Pudp;
          //  cout << "UDP \n";
            break;
        default:
          //  cout << "IP \n";
            break;
        }
        logger.push(retNum(), Tcap, p, ehtHeader->src_mac, ehtHeader->dst_mac, ntohl(v4header->source_address), ntohl(v4header->destination_address));

    }
    else if (etherType == 0x0806) {//Arp

        ArpPacket* pkt = (ArpPacket*)pkt_data;

        if (ntohs(pkt->arp_head.opcode == OP_REQ)) //请求报文
            return;

        if (!IsArpForSelf((ArpPacket*)pkt_data))  //不在同一网段报文
            return;

        //cout << "ARP\n";
        arpCache.update(ntohl(pkt->arp_head.sender_proto_addr), pkt->arp_head.sender_hw_addr);
        logger.push(0, Tcap, Parp, pkt->arp_head.sender_hw_addr, pkt->arp_head.target_hw_addr, ntohl(pkt->arp_head.sender_proto_addr), ntohl(pkt->arp_head.target_proto_addr));
    }

}
```

8、转发线程

逻辑在1中已经叙述

- packetBuffer.pop()从全局数据包缓存区拿一个包pkt，在_transPacket(pkt)进行转发，转发则根据路由表、ARP缓存，最后调用npcap的发送API。（详细见源码）
- _transPacket()将返回0-3，分别代表成功发送、路由查询失败、ARP不可达、INC发送失败，根据错误类型进行mess保存。

```
DWORD WINAPI tnsThrd(LPVOID lpParam) {
    cout << "Trans Thread started." << endl;
    while (keep_trn_trd) {

        int rst;
        u_char* pkt = packetBuffer.pop();  //阻塞

        rst = _transPacket(pkt);
        if (!rst)
            continue;

        v4Header* v4head = (v4Header*)(pkt + sizeof(eth_header));

        /* 报错信息 */
        string opt = "Error: ";
        if (rst == 1)
            opt += intToIp(ntohl(v4head->destination_address)) + " 路由查询失败";
        else if (rst == 2) {
```

```
            opt += intToIp(ntohl(v4head->destination_address))+ ":" +
arrayToMac(((eth_header*)pkt)->dst_mac) + "下一跳地址查询失败";
            //Send ICMP Timeout
        }
        else if (rst == 3) {
            opt += intToIp(ntohl(v4head->destination_address)) + ":" +
arrayToMac(((eth_header*)pkt)->dst_mac) + "INC 发送数据失败";
        }
        logger.push(opt);
    }
    return 0;
}
```

9. 命令系统

命令系统主要通过访问以上介绍的各个全局变量，输出他们的状态。 `parseCmd()`可以解析的指令已经在文档一开始介绍过。

```
void cmdThrd(){

    cout << "cmd Thread started."<<endl;
    while (true)
    {
        string cmd;
        cout << "@fondH/>";
        getline(cin,cmd);
        if (cmd == "exit")
            break;
        //cout << "\n";
        parseCmd(cmd);
    }

    exit_router();

}
```

效果大概如下：

```
@fondH/>
@fondH/>
@fondH/>arp-chche

Unknown command.
@fondH/>arp-cache


ArpCache:
============================================================
IP              MAC             是否有效
206.1.2.2 -- > 00-0c-29-1b-c0-76  Va
206.1.1.2 -- > 00-0c-29-5d-0c-bc  Va
@fondH/>
@fondH/>
@fondH/>
@fondH/>
@fondH/>
@fondH/>
@fondH/>route print


IPv4 Rooting Table
============================================================
网络目标        掩码            下一跳地址      接口
0.0.0.0         0.0.0.0         0.0.0.0         1
206.1.1.0       255.255.255.0   0.0.0.0         0
206.1.2.0       255.255.255.0   0.0.0.0         1
206.1.3.0       255.255.255.0   206.1.2.2       1
@fondH/>
```

五、过程验证

1. 根据拓扑图网络配置好，手动在R2主机插入路由条目：

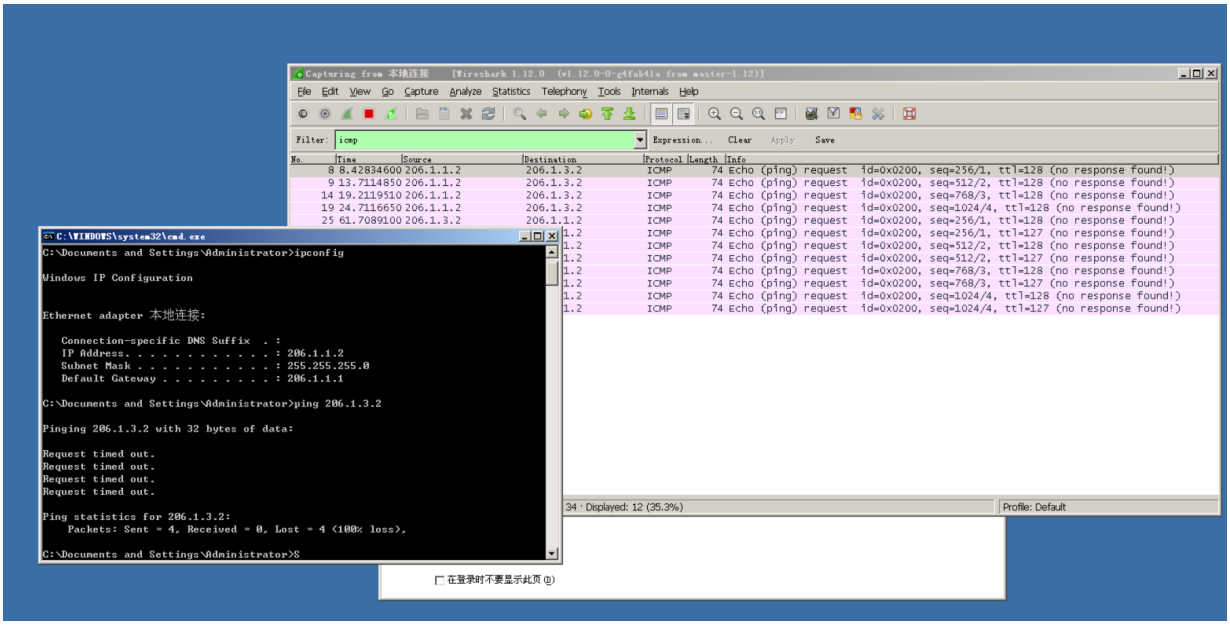
> route add 206.1.1.0 mask 255.255.255.0 206.1.2.1
>
> route print

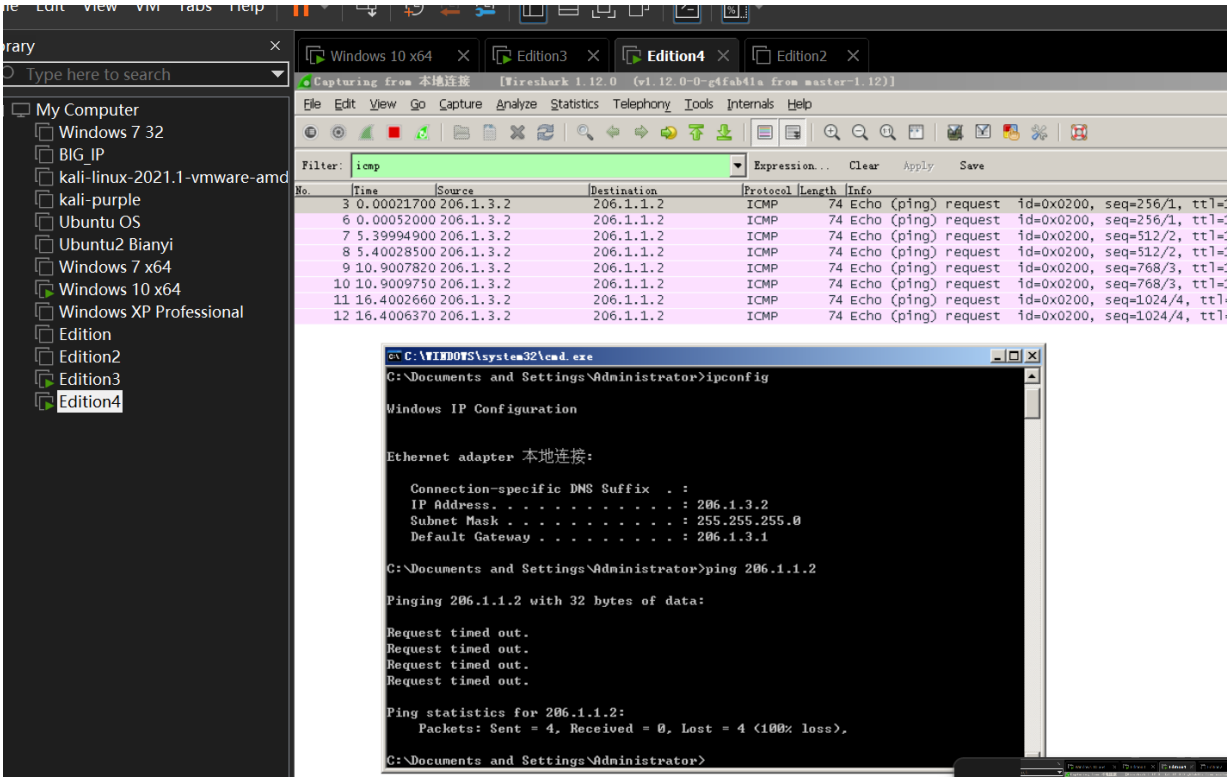
```
C:\Documents and Settings\Administrator>route add 206.1.1.0 mask 255.255.255.0 2
06.1.2.1

C:\Documents and Settings\Administrator>route print

IPv4 Route Table
===========================================================================
Interface List
0x1 ........................... MS TCP Loopback interface
0x10003 ...00 0c 29 1b c0 76 ...... Intel(R) PRO/1000 MT Network Connection
===========================================================================
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
        122.0.0.0        255.0.0.0        122.0.0.1        122.0.0.1      1
        206.1.1.0    255.255.255.0        206.1.2.1        206.1.2.2      1
        206.1.2.0    255.255.255.0        206.1.2.2        206.1.2.2     10
        206.1.2.2  255.255.255.255        127.0.0.1        127.0.0.1     10
      206.1.2.255  255.255.255.255        206.1.2.2        206.1.2.2     10
        206.1.3.0    255.255.255.0        206.1.3.1        206.1.2.2     10
        206.1.3.1  255.255.255.255        127.0.0.1        127.0.0.1     10
      206.1.3.255  255.255.255.255        206.1.3.1        206.1.2.2     10
        224.0.0.0        240.0.0.0        206.1.2.2        206.1.2.2     10
```

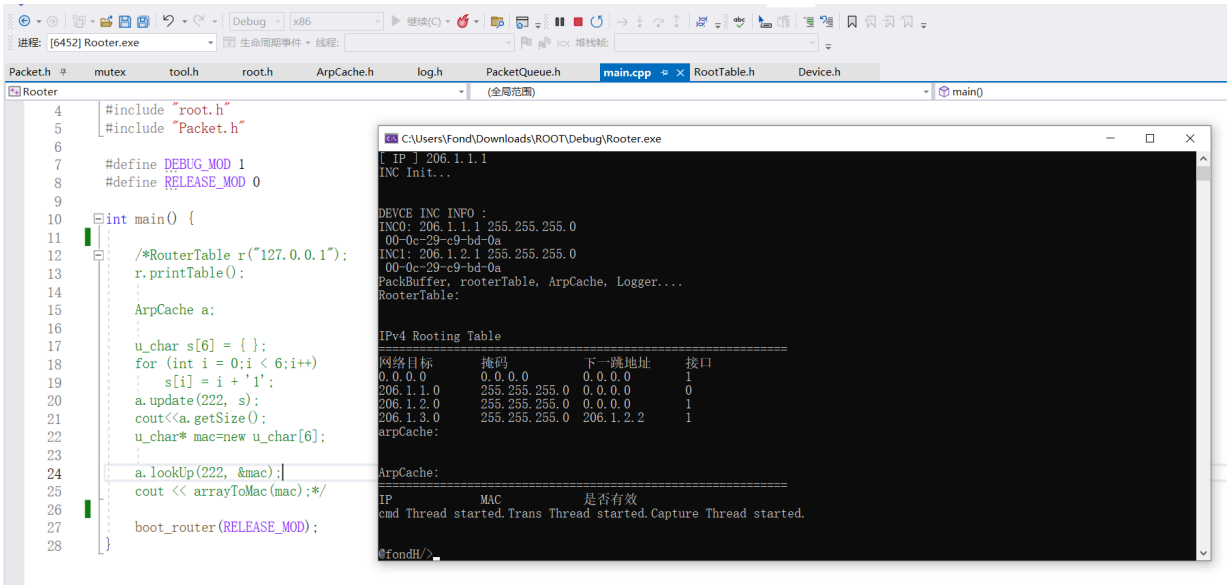
2. 使用PC1 ping PC2 或者 使用PC2 ping PC1，无法ping 通



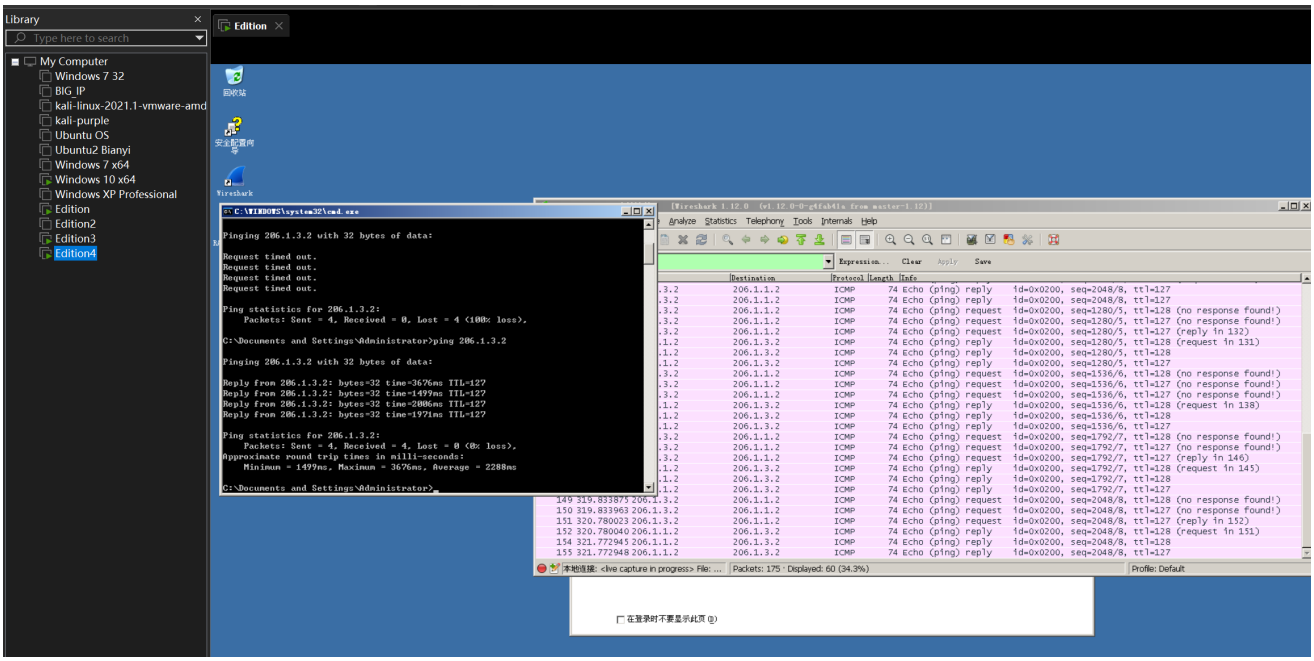


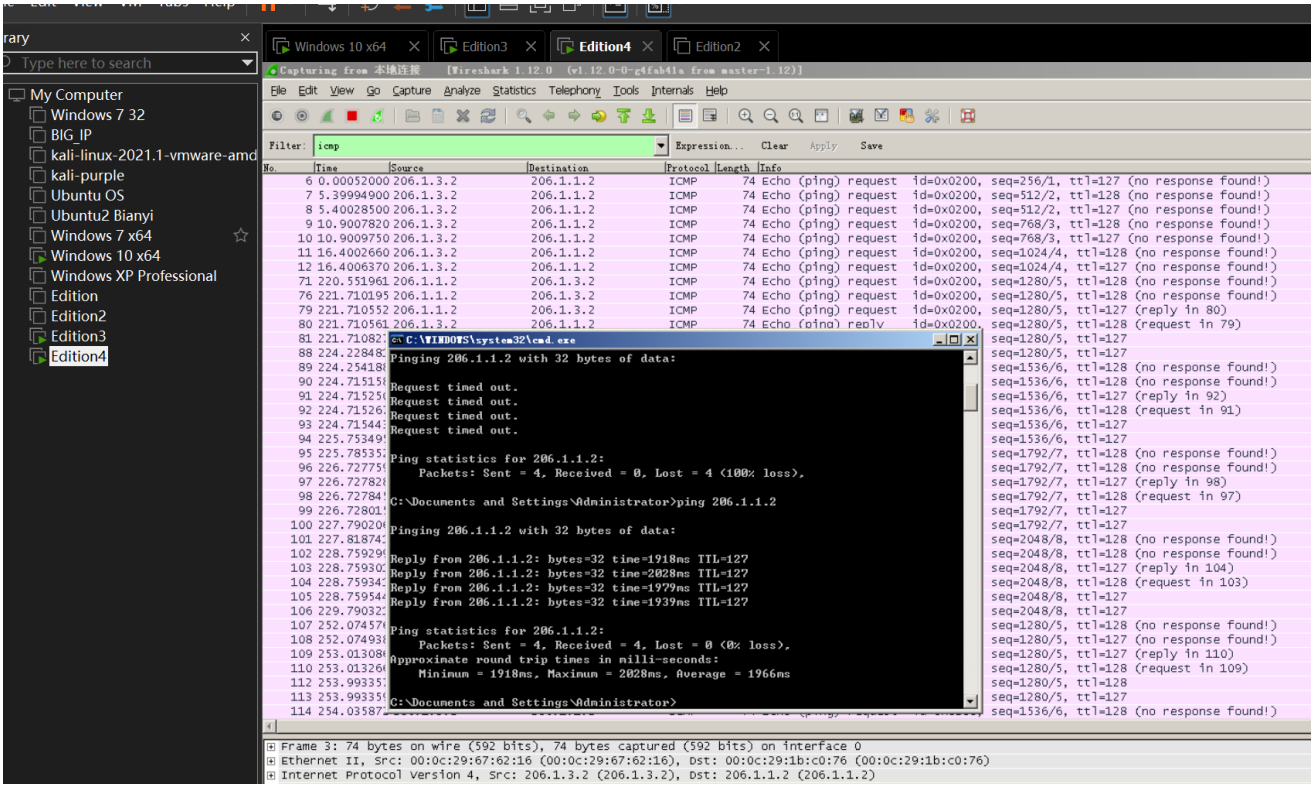
3. 开启路由程序， 此时它的路由表项我已经添加。

```
rooterTable.addRoute(RouteEntry(ip_INC[0]& mask_INC[0],mask_INC[0], 0, 0));
rooterTable.addRoute(RouteEntry(ip_INC[1]& mask_INC[1], mask_INC[1], 0, 1));
rooterTable.addRoute(RouteEntry("206.1.3.0", "255.255.255.0", "206.1.2.2", "1"));
```



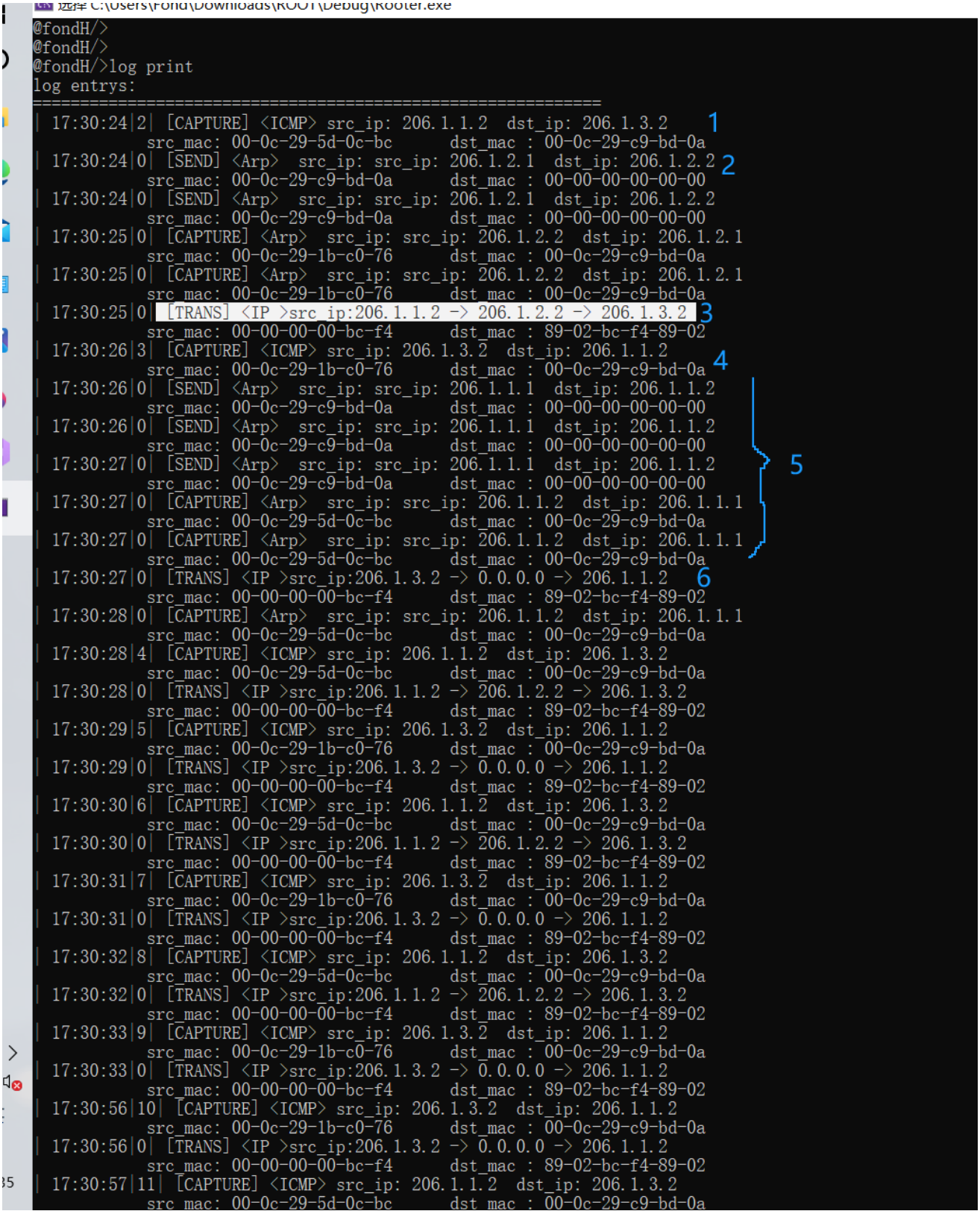
4. 再次互相ping，可以ping通



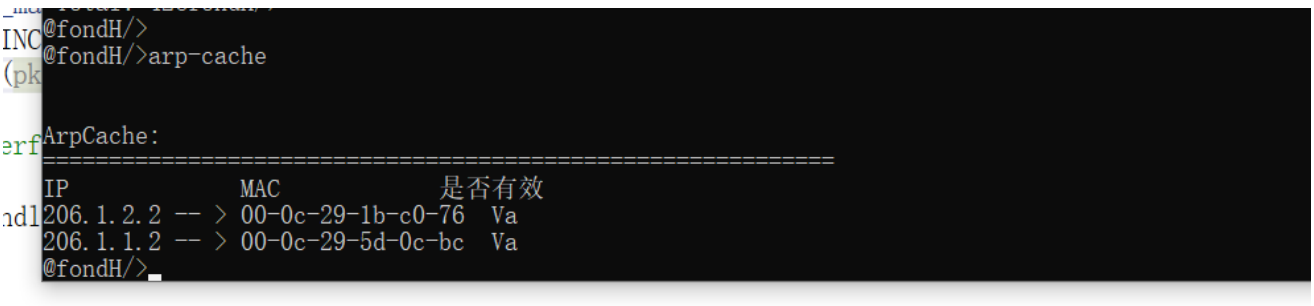


此时路由器日志

- 1：路由收到PC1发送ICMP
- 2：路由转发进行进行转发1中数据，在arpCache查询失败，发送arp得到下一跳MAC（R2的MAC），日志可知知道，在arpCache更新之前，程序发送了两次arp广播，并且都很快顺利得到回复
- 3：更新了arpCache，此时对1的包进行转发
- 4：收到了来自R2的回复，这个回复是R2转发自PC2的。
- 5：由于程序根据路由表查询，arpCache没有下一跳地址（PC1的MAC），进行arp广播，有日志可以知道，发送了三次广播，并且都得到回复。
- 6：更新了arpCache后，此时对4中收到的包进行转发，这个包即对PC1发送的对一个ICMP的回复。



此时的arpCache：



六、问题与思考

1. 多线程导致的容器数据一致性、效率；

> 数据包缓冲区我使用的容器，涉及到转发线程、捕获线程的"写"操作和log线程的"读"操作，而这些写操作均使用了容器自身的迭代器，若不进行特殊处理，即一个线程使用这些迭代器进行写，另一个线程再访问就触发异常。而引入锁机制后，我在每一个push和pop均`lock_guard<mutex> lk(mtx)`，每一次加锁、解锁带来的开销实则跟push、pop本身的开销相同；

我希望能进行改进：

- 每一次push和pop都不在对一个包进行，但有需要在转发、捕获线程维护新的暂存区。
- 或者避免使用容器，仅仅使用单向链表，捕获端负责尾指针，向链表接新的数据包，接收端负责头指针，将链表前面的数据一个个摘走，可以避免使用全局容器带来的异常问题。

2. 路由表相关设置不当导致的**回路**，导致转发流量剧增。

> 实验测试过程中遇见过一类错误，错误根源在于路由表的`LookUp`函数中，每次都返回了指向路由R2的`routeEntry`，导致当PC0发送一个ICMP，迅速WireShark捕捉到上万条源IP是PC1、目的IP是PC2，开始没找见错误原因，查询了大量关于ping命令实现机制的文章，认为是因为PC1由于某些原因自己主动发送的。后续通过wireshark，逐个检查每一个包的目的、源MAC地址，才发现回路这个事情，也耽误许多时间。

3.Demo仅实现IP转发和ARP主动请求，其余均未处理，而在查看日志时，发现许多通过过滤，并且加入数据包缓冲区等待转发的包，但是基本都转发失败，根据错误信息，都是arp无法获得正确的地址（但我路由表的设计，一定可以查询到一个记录的）。