南 开 大 学

计算机与网络空间安全学院

网络技术与应用课程报告

第五次实验报告

学号：2012911
姓名：马永田
年级：2020 级
专业：计算机科学与技术

2022 年 12 月 10 日

第1节 实验内容说明

（一）简单路由器程序设计实验的具体要求为：

1. 设计和实现一个路由器程序，要求完成的路由器程序能和现有的路由器产品（如思科路由器、华为路由器、微软的路由器等）进行协同工作。
2. 程序可以仅实现IP数据报的获取、选路、投递等路由器要求的基本功能。可以忽略分片处理、选项处理、动态路由表生成等功能。
3. 需要给出路由表的手工插入、删除方法。
4. 需要给出路由器的工作日志，显示数据报获取和转发过程。
5. 完成的程序须通过现场测试，并在班（或小组）中展示和报告自己的设计思路、开发和实现过程、测试方法和过程。

第2节 实验准备

(一) 配置实验环境并构建项目

安装 Npcap1.71、npcap-sdk-1.13,并对Visual Studio的项目进行如下配置：

• 在 C/C++ 中添加附加包含目录：D:\npcap\Include;

• 在预处理器中添加预处理器定义：WPCAP;HAVE_REMOTE;

• 在链接器-> 常规中添加附加库目录：D:\npcap\Lib;

• 在链接器-> 输入中添加附加依赖项：Packet.lib;wpcap.lib

(二) 学习 WinPcap 编程方法

一、获取网络接口设备列表： pcap_findalldevs_ex()

函数原型为：

```
int pcap_findalldevs_ex(
    char* source,
    struct pcap_rmtauth* auth,
    pcap_if_t** alldevs,
    char* errbuf)
```

返回值：

0表示查找成功。

-1表示查找失败

参数：

1. source 指定是本地适配器或者远程适配器，它告诉函数必须在哪里进行查找.
2. struct pcap_rmtauth *auth （auth参数可以为NULL.)
3. pcap_if_t **alldevs 该参数用于存放获取的适配器数据。如果查找失败，alldevs的值为NULL.
4. char *errbuf 该参数存放查找失败的信息。

二、打开网络接口：pcap_open()

函数定义如下：

```
pcap_t* pcap_open(
    const char* source,
    int snaplen,
    int flags,
    int read_timeout,
    struct pcap_rmtauth* auth,
    char* errbuf)
```

参数：

1. source：以/0终止的字符串，其中包含要打开的源名称。
2. snaplen: 必须保留的包的长度。对于由过滤器接收的每个数据包，只有第一个“snaplen”字节存储在缓冲区中并传递给用户应用程序。
3. flags: 保留捕获数据包可能需要的几个标志。
4. read_timeout: 以毫秒为单位读取超时。
5. auth: 指向“结构pcap_rmtauth”的指针，它保留在远程机器上验证用户所需的信息。如果这不是远程捕获，则该指针可以设置为NULL。
6. errbuf: 指向用户分配的缓冲区的指针，该缓冲区将在该函数失败的情况下包含错误。

三、捕获数据包：pcap_next_ex()

函数定义如下：

```
int pcap_next_ex(
    pcap_t* p,
    struct pcap_pkthdr** pkt_header,
    const u_char** pkt_data)
```

参数：

1. p: 已打开的捕捉实例的描述符
2. pkt_header : 报文头
3. pkt_data : 报文内容

返回值：

1 : 成功

0 : 获取报文超时

- 1 : 发生错误

- 2 : 获取到离线记录文件的最后一个报文

四、发送数据包：pcap_sendpacket()

函数定义如下：

```
int pcap_sendpacket(
    pcap_t* p,
    const u_char* buf,
    int size);
```

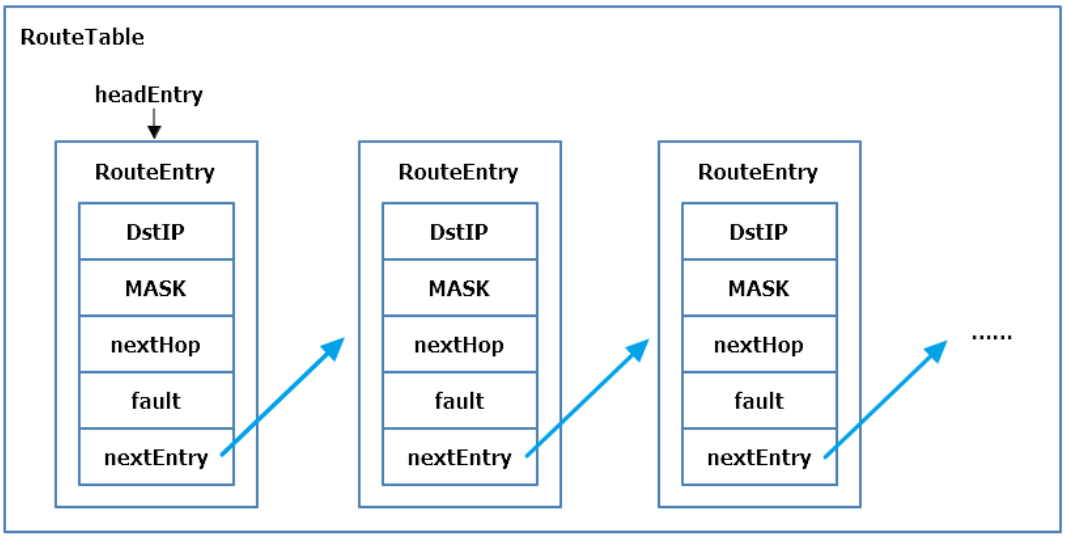
参数:

1. 一个装有要发送数据的缓冲区;
2. 要发送的长度;
3. 一个适配器;

注意：缓冲区中的数据将不被内核协议处理，只是作为最原始的数据流被发送，所以我们必须填充好正确的协议头以便正确的将数据发送

(三) 设计路由表

本次实验的路由表采用链式结构，其结构大致如下图所示:

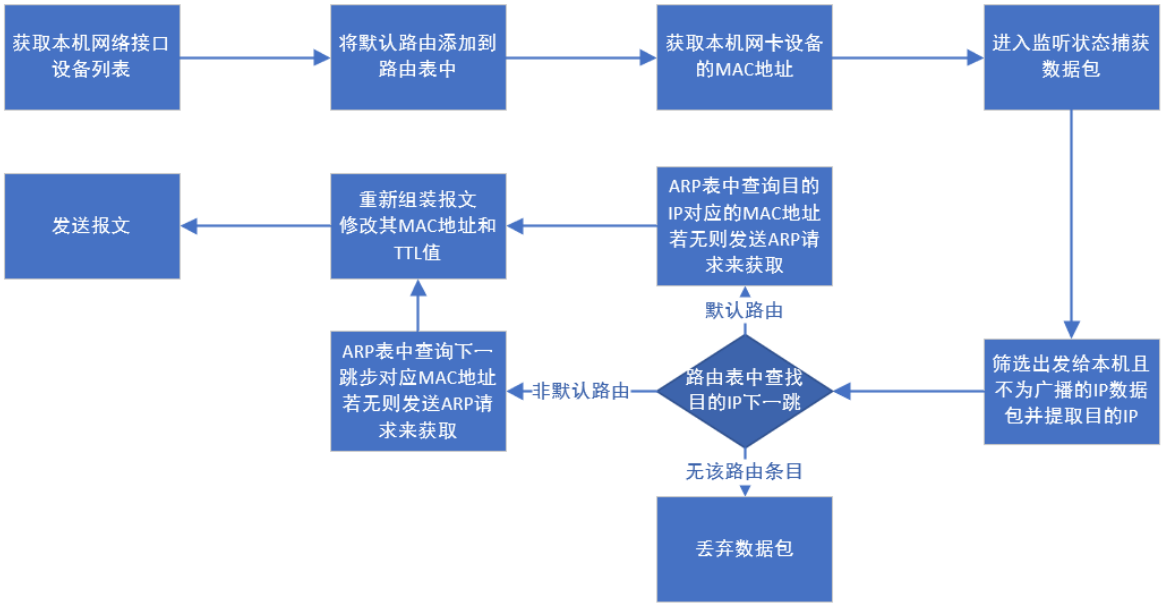


使用链式结构来存储路由表，对于内存空间的管理更为灵活便捷，路由条目的数量不会被结构所限制，能够更好的利用内存，可以随时添加和删除路由条目；且使用链式结构存储可以更加方便的维护一个有序列表，更容易实现最长匹配原则。

第3节 实验过程

(一) 项目设计思路

该程序实现路由转发的大致流程如图所示：



程序执行后首先会进入命令输入模式，可以进行路由条目的添加和删除、以及路由表和ARP表的打印等操作，之后输入命令route start启动路由：

1. 首先程序会获取本机网卡的信息，将其进行打印并存储IP地址和掩码
2. 之后程序对路由表进行初始化，并添加默认路由条目
3. 初始化ARP表，获取本机网卡的MAC地址并添加到ARP表中
4. 完成初始化工作后路由程序进入监听状态捕获数据包并对其进行筛选
5. 路由程序会筛选出目的MAC地址为本机MAC且不为广播的IP数据包进行解析，获取其中的目的IP地址
6. 之后程序会到路由表中查询目的IP地址的下一跳步：
    1. 若无该路由条目，丢弃数据包
    2. 若为默认路由则直接投递，到ARP表中查询目的IP对应的MAC地址
    3. 若不为默认路由，获取下一跳步的IP地址，到ARP表中查询其对应的MAC地址
7. 若ARP表中未查找到对应的MAC地址，路由程序会发送ARP报文获取对应IP的MAC地址
8. 获取到对应的MAC地址后路由程序会重新组装报文，修改其中的目的MAC地址与TTL值
9. 最后路由程序转发报文，完成路由

(二) 报文格式

```
#pragma pack (1)//进入字节对齐方式
//以太网帧 14字节
typedef struct FrameHeader_t {
    BYTE DesMAC[6];// 目的地址
    BYTE SrcMAC[6];//源地址
    WORD FrameType;//帧类型
}FrameHeader_t;
//ARP帧 28字节
```

```
typedef struct ARPFrame_t {
    FrameHeader_t FrameHeader;//以太网帧头
    WORD HardwareType;//硬件类型
    WORD ProtocolType;//协议类型
    BYTE HLen;//硬件地址长度
    BYTE PLen;//协议地址长度
    WORD Operation;
    BYTE SendHa[6]; //发送端以太网地址
    DWORD SendIP;   //发送端IP地址
    BYTE RecvHa[6]; //目的以太网地址
    DWORD RecvIP;   //目的IP地址
} ARPFrame_t;
//IP首部
typedef struct IPHeader_t {
    BYTE Ver_HLen;
    BYTE TOS;
    WORD TotalLen;
    WORD ID;
    WORD Flag_Segment;
    BYTE TTL;//生命周期
    BYTE Protocol;
    WORD Checksum;//校验和
    ULONG SrcIP;//源IP
    ULONG DstIP;//目的IP
}IPHeader_t;
//包含帧首部和IP首部的数据包
typedef struct Data_t {
    FrameHeader_t FrameHeader;//帧首部
    IPHeader_t IPHeader;//IP首部
}Data_t;
//包含帧首部和IP首部的数据包
typedef struct ICMP {
    FrameHeader_t FrameHeader;
    IPHeader_t IPHeader;
    char buf[40];
}ICMP_t;
#pragma pack ()
```

(三) 路由表的设计与实现

路由条目RouteEntry类

本次简单路由程序的设计中路由表是采用链式存储的方式实现的，路由条目RouteEntry类的结构较为简单，除了包含路由条目所需的四项内容：目的地址、子网掩码、下一跳步、默认路由标记之外，还包括一个RouteEntry类指针变量来指向下一条路由条目。

考虑到路由表可能需要频繁的删除和插入操作， 以及我们需要维护一个有序的序列来实现最长匹配原则查找下一跳步，因此使用链式结构进行存储有一定优势所在。

```
class RouteEntry
{
public:
    DWORD destIP;   //目的地址
    DWORD mask;     //子网掩码
    DWORD nextHop;  //下一跳
```

```
    bool fault;     //是否为默认路由
    RouteEntry* nextEntry;  //链式存储
    RouteEntry(){
        memset(this, 0, sizeof(*this)); //初始化
        nextEntry = NULL;
    }
    void printEntry(); //打印表项内容
};
```

路由表RouteTable类

路由表RouteTable类用于管理路由条目，采用的是链式存储的方式，成员变量包括一个路由条目头结点指针来指向链表开头以及一个变量来记录路由条目数量。

其中：

- 对路由表进行初始化时，会将默认路由表项(即本机IP所在网段)添加到路由表中
- 添加新的路由条目时会根据子网掩码的长度进行插入，保证子网掩码长度为降序排列
- 删除路由表时会检验要删除的路由条目是否为默认路由，只有非默认路由才可以被删除
- 查询路由表时若未查找到则返回 -1，若查找到的路由条目为默认路由则返回0，若非默认路由则返回下一跳步
- 其中默认路由会被维护在链表的起始

```
//路由表
class RouteTable
{
public:
    RouteEntry* head;
    int routeNum;
    //初始化路由表 添加默认路由
    void initRouteTable() {
        head = NULL;
        routeNum = 0;
        for (int i = 0; i < 2; i++) {
            RouteEntry* newEntry = new RouteEntry();
            newEntry->destIP = (inet_addr(ipList[i])) &
(inet_addr(maskList[i]));
            newEntry->mask = inet_addr(maskList[i]);
            newEntry->fault = 1;//1为默认路由
            this->addEntry(newEntry);//添加表项
        }
    }
    //添加路由表项
    void addEntry(RouteEntry* newEntry) {
        if (head == NULL) {//头结点处理
            head = newEntry;
            routeNum++;
            return;
        }
        if (newEntry->mask > head->mask) {//头结点处理
            newEntry->nextEntry = head;
            head = newEntry;
            routeNum++;
            return;
        }
```

```
        RouteEntry* cur = head;
        while(cur->falut){//跳过默认路由
            cur=cur->nextEntry;
        }
        while (cur->nextEntry) {//按照掩码长度查找合适位置
            if (newEntry->mask > cur->nextEntry->mask) {
                break;
            }
            cur = cur->nextEntry;
        }
        newEntry->nextEntry = cur->nextEntry;
        cur->nextEntry = newEntry;
        routeNum++;
        return;
    }
    //删除路由表
    bool deleteEntry(DWORD IP) {
        if (IP == head->destIP && !head->fault) {
            delete head;
            head = NULL;
            return true;
        }
        RouteEntry* cur = head;
        while (cur->nextEntry) {
            if (cur->nextEntry->destIP == IP) {//查找对应路由条目进行删除
                RouteEntry* temp = cur->nextEntry;
                if (temp->fault) {//默认路由无法删除
                    return false;
                }
                cur->nextEntry = temp->nextEntry;
                delete temp;
                return true;
            }
            cur = cur->nextEntry;
        }
        return false;
    }
    //查找最长前缀,返回下一跳步
    DWORD lookup(DWORD ip) {
        RouteEntry* cur = head;
        while (cur != NULL) {
            if ((cur->mask & ip) == (cur->mask & cur->destIP)) {
                if (cur->fault) {//默认路由返回0
                    return 0;
                }
                return cur->nextHop;//查找成功返回下一跳步
            }
            cur = cur->nextEntry;
        }
        return -1;//查找失败返回-1
    }
    //路由表的打印 mask net next type
    void printTable();
};
```

(四) ARP表设计

ARP条目ArpEntry类

ARP条目ArpEntry类较为简单，只需包括IP地址和MAC地址以及一个打印功能函数即可：

```
//arp条目
class ArpEntry
{
public:
    DWORD IP;//IP地址
    BYTE MAC[6];//MAC地址
    void printEntry();
};
```

ARP表ArpTable类

ARP表ArpTable类用于管理ARP条目，该部分实际上可以选择直接使用Hash表的结构来实现，每次查询只需O(1)的复杂度就能完成操作，但由于Hash表的实现过于复杂，且本次实验环境中又无法使用STL库，因此选择直接使用数组来实现该ARP表。

实际ARP表中的条目应设置生命周期，避免出现IP地址与MAC地址的映射过期的情况，但本次实验较为简单并未涉及。此外由于ARP协议为自动获取，实际上我们无需手动操作ARP表，因此也未专门设计ARP表的删除接口。

而是设计在插入条目时考虑IP地址是否为表中已存在的IP地址，若已存在则不插入新条目，而是对旧条目进行更新，以此实现插入与更新操作，实际上这种设计会提高插入操作的复杂度，但考虑到本次实验较为简单，这种影响其实可以忽略不计。程序在查找ARP条目时会直接将查找到的MAC地址直接存入到参数中，如查找到则返回索引，否则返回-1。

```
//arp表
class ArpTable
{
public:
    ArpEntry arp_table[50];
    ArpTable() {
        arpNum = 0;
    };
    int arpNum = 0;
    //插入or更新ARP表
    void insert(DWORD ip, BYTE mac[6]) {
        for (int i = 0; i < arpNum; i++) {
            if (arp_table[i].ip == ip) {
                CopyMAC(mac,arp_table[i].mac);
                return;
            }
        }
        arp_table[arpNum].ip = ip;
        CopyMAC(mac, arp_table[arpNum].mac);
        arpNum++;
    }
    //查找ARP表
    int lookup(DWORD ip,BYTE mac[6]) {
        for (int i = 0; i < arpNum; i++) {
```

```
            pIP = (unsigned char*)&arp_table[i].ip;
            if (ip == arp_table[i].ip) {
                CopyMAC(arp_table[i].mac, mac);
                return i;
            }
        }
        return -1;
    }
    void printTable();
};

```

(五) 获取MAC地址

伪造ARP包

MAC地址的获取需要通过ARP协议，因此我们需要先组装一个ARP报文，其中内容如下注释：

```
//伪造ARP包
ARPFrame_t MakeARP() {
    ARPFrame_t ARPFrame;
    for (int i = 0; i < 6; i++)
        ARPFrame.FrameHeader.DesMAC[i] = 0xff;//表示广播
    for (int i = 0; i < 6; i++)
        ARPFrame.FrameHeader.SrcMAC[i] = 0x0f;//最初需要编造MAC地址 后续会更改
    ARPFrame.FrameHeader.FrameType = htons(0x806);//帧类型为ARP
    ARPFrame.HardwareType = htons(0x0001);//硬件类型为以太网
    ARPFrame.ProtocolType = htons(0x0800);//协议类型为IP
    ARPFrame.HLen = 6;//硬件地址长度为6
    ARPFrame.PLen = 4;//协议地址长为4
    ARPFrame.Operation = htons(0x0001);//操作为ARP请求
    for (int i = 0; i < 6; i++)
        ARPFrame.SendHa[i] = 0x0f;//最初需要编造MAC地址 后续会更改
    for (int i = 0; i < 6; i++)
        ARPFrame.RecvHa[i] = 0;//最初设置为0 表示目的地址未知
    return ARPFrame;
}
```

获取MAC地址

MAC地址的获取流程如下：

1. 伪造ARP请求报文，其中内容如上一节代码及注释所示
    1. 若获取本地MAC地址，则其中的源MAC地址等信息直接伪造即可
    2. 若获取远端MAC地址，则需要将本机网卡的MAC地址与IP地址以及目的IP填入
2. 打开对应网卡并发送伪造的ARP报文
3. 对该网卡进行流量监听，筛选出我们需要的ARP报文：
    1. 类型为0x806
    2. 源MAC地址不为发出ARP包的MAC地址，确定不是本机发出的APR报文
    3. 目的MAC地址为发出ARP包的源MAC地址，确定为响应ARP报文
    4. 目的IP地址我发出ARP包的源IP地址，确定为响应ARP报文
4. 解析捕获的ARP报文，提取MAC地址并更新ARP表

```
//获取远端MAC地址
void getRemoteMAC(pcap_if_t* device, DWORD DstIP) {
    int index = 0;
    for (pcap_addr* a = device->addresses; a != nullptr; a = a->next) {
        if (((struct sockaddr_in*)a->addr)->sin_family == AF_INET && a->addr) {
            DWORD devIP = inet_addr(inet_ntoa(((struct sockaddr_in*)a->addr)->sin_addr));
            if ((devIP & inet_addr(maskList[index])) != (DstIP & inet_addr(maskList[index]))) {//使用同网段的IP来获取远端MAC地址
                continue;
            }
            ARPFrame_t ARPFrame = MakeARP();//在当前网卡上伪造一个ARP包
            CopyMAC(MyMAC, ARPFrame.FrameHeader.SrcMAC);//本机MAC填入伪造的ARP包
            CopyMAC(MyMAC, ARPFrame.SendHa);
            ARPFrame.SendIP = inet_addr(inet_ntoa(((struct sockaddr_in*)a->addr)->sin_addr));
            ARPFrame.RecvIP = DstIP;//设置ARP包中源和目的IP地址
            pcap_t* adhandle = pcap_open(device->name, 655340, PCAP_OPENFLAG_PROMISCUOUS, 1000, 0, 0);  //打开该网卡的网络接口
            if (adhandle == NULL) { return; }
            //发送ARP包
            int res = pcap_sendpacket(adhandle, (u_char*)&ARPFrame, sizeof(ARPFrame_t));
            //捕获数据包
            ARPFrame_t* RecPacket;
            struct pcap_pkthdr* pkt_header;
            const u_char* pkt_data;
            while ((res = pcap_next_ex(adhandle, &pkt_header, &pkt_data)) >= 0) {
                RecPacket = (ARPFrame_t*)pkt_data;
                if (!CompareMAC(RecPacket->FrameHeader.SrcMAC, ARPFrame.FrameHeader.SrcMAC)
                    && CompareMAC(RecPacket->FrameHeader.DesMAC, ARPFrame.FrameHeader.SrcMAC)
                    && RecPacket->SendIP == ARPFrame.RecvIP
                    ) {//过滤成功
                    //插入ARP表中
                    arpTable.insert(DstIP, RecPacket->FrameHeader.SrcMAC);
                    break;
                }
            }
        }
        index++;
    }
}

```

(六) 数据报捕获

启动路由转发功能后，程序会进入循环，监听数据报。对捕获到的数据报进行筛选：

- 报文发往本机，即目的MAC地址为本机MAC地址
- 报文为IP格式的数据报
- 使用校验和进行校验结果正确
- 路由表中有对应条目
- 确定不是广播消息
- 能够获取对应IP的MAC地址

若满足上述条件则进行路由转发，否则将报文丢弃。

其中对于路由表的查询在前文中已叙述过，会对默认路由、非默认路由以及无该路由条目的情况进行相应的区分处理。

而对于MAC地址的获取，若查询ARP表中发现并无该条目后，会自动调用前文所述的getRemoteMAC函数来尝试获取MAC地址并更新ARP表，之后再次查询ARP表，若依旧未查找到，说明获取失败，选择丢弃该报文。

```
//捕获IP数据报
ICMP CapturePacket(pcap_if_t* device) {
    pcap_t* adhandle = pcap_open(device->name, 655340,
PCAP_OPENFLAG_PROMISCUOUS, 1000, 0, 0);
    pcap_pkthdr* pkt_header;
    const u_char* pkt_data;
    while (1) { //循环 监听数据报
        int res = pcap_next_ex(adhandle, &pkt_header, &pkt_data);
        if (res > 0) {
            FrameHeader_t* header = (FrameHeader_t*)pkt_data;
            if (CompareMAC(header->DesMAC, MyMAC) //目的MAC为本机
               && ntohs(header->FrameType) == 0x800 //IP格式数据报
               && checkCheckSum((Data_t*)pkt_data)){ //校验通过
                outputLog(data, 1);//打印日志
                //提取IP数据报中的目的IP并到路由表中查找
                DWORD DstIP = data->IPHeader.DstIP;
                DWORD routeFind = routeTable.lookup(DstIP);
                if (routeFind == -1) { //没有该路由条目 丢弃
                    continue;
                }
                BYTE broadcast[6] = "fffff";//是否广播
                if (!CompareMAC(data->FrameHeader.DesMAC, broadcast)
                    && !CompareMAC(data->FrameHeader.SrcMAC, broadcast)){
                    ICMP_t* sendPacket_t = (ICMP_t*)pkt_data;
                    ICMP_t sendPacket = *sendPacket_t;
                    BYTE mac[6];
                    if (routeFind == 0) { //默认路由表项直接投递 ARP表中查询DstIP
                        if (arpTable.lookup(DstIP, mac) == -1) {
                            //ARP表中无该条目 getRemoteMAC获取目的IP的MAC地址
                            getRemoteMAC(device, DstIP);
                            if (arpTable.lookup(DstIP, mac) == -1) {//失败
                                continue;//丢弃
                            }
                        }
                        routeForward(adhandle, sendPacket, mac);//路由转发
```

```
                    }
                    else {//非默认路由 获取下一跳 ARP表中查询nextHop的MAC地址
                        if (arpTable.lookup(routeFind, mac) == -1) {
                            //ARP表中无该条目 getRemoteMAC获取目的IP的MAC地址
                            getRemoteMAC(device, routeFind);
                            if (arpTable.lookup(routeFind, mac) == -1) {
                                continue;//丢弃
                            }
                        }
                        routeForward(adhandle, sendPacket, mac);//路由转发
                    }
                }
            }
        }
    }
}
```

(七) 路由转发函数

对要进行路由转发的数据报，修改其中的MAC地址，源MAC地址填入为本机的MAC地址，目的MAC地址填入函数参数的MAC地址，更新TTL值并重新设置校验和，之后发送数据报。

```
//路由转发函数
void routeForward(pcap_t* adhandle, ICMP_t data, BYTE DstMAC[]) {
    Data_t* sendPacket = (Data_t*)&data;
    //源MAC为本机MAC
    memcpy(sendPacket->FrameHeader.SrcMAC, sendPacket->FrameHeader.DesMAC,
6);
    //目的MAC为下一跳MAC
    memcpy(sendPacket->FrameHeader.DesMAC, DstMAC, 6);
    sendPacket->IPHeader.TTL -= 1;//TTL-1
    if (sendPacket->IPHeader.TTL < 0) {
        return;//丢弃
    }
    setCheckSum(sendPacket);//重设校验和
    int res = pcap_sendpacket(adhandle, (const u_char*)sendPacket, 74);//发送
数据报
    if (res == 0) {
        outputLog(sendPacket, 0);
    }
}
```

(八) 校验和

```
void setCheckSum(Data_t* dataPacket) {//设置校验和

    dataPacket->IPHeader.Checksum = 0;
    unsigned int sum = 0;
    WORD* t = (WORD*)&dataPacket->IPHeader;//16bit一组
    for (int i = 0; i < sizeof(IPHeader_t) / 2; i++) {
        sum += t[i];
        while (sum >= 0x10000) {//溢出
            int s = sum >> 16;
            sum -= 0x10000;
```

```
            sum += s;
        }
    }
    dataPacket->IPHeader.Checksum = ~sum;//取反
}
bool checkCheckSum(Data_t* dataPacket) {//检验
    unsigned int sum = 0;
    WORD* t = (WORD*)&dataPacket->IPHeader;
    for (int i = 0; i < sizeof(IPHeader_t) / 2; i++) {
        sum += t[i];
        while (sum >= 0x10000){
            int s = sum >> 16;
            sum -= 0x10000;
            sum += s;
        }
    }
    if (sum == 65535)
        return 1;//校验正确
    return 0;
}
```

(八) 日志输出

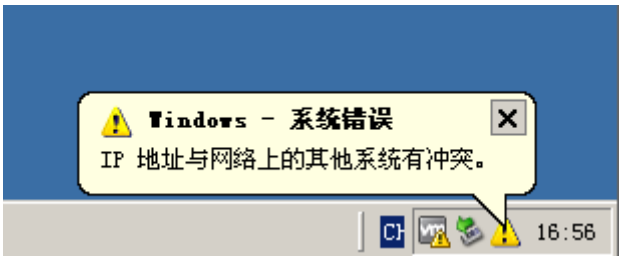
```
//日志输出
void outputLog(Data_t* dataPacket, bool receive) {
    FILE* fp = fopen("myRouter.log", "a");
    if (fp == NULL){
        printf("File opening failed! \n");
        return;
    }
    DWORD nexthop = routeTable.lookup((DWORD)dataPacket->IPHeader.DstIP);
    unsigned char* SrcIP = (unsigned char*)&dataPacket->IPHeader.SrcIP;
    unsigned char* DstIP = (unsigned char*)&dataPacket->IPHeader.DstIP;
    unsigned char* nextHop = (unsigned char*)&nexthop;
    BYTE* SrcMAC = dataPacket->FrameHeader.SrcMAC;
    BYTE* DstMAC = dataPacket->FrameHeader.DesMAC;

    if (receive) {
        fprintf(fp, "[receive IP] SrcIP:%u.%u.%u.%u DesIP:%u.%u.%u.%u
SrcMAC:%02X-%02X-%02X-%02X-%02X-%02X DesMAC:%02X-%02X-%02X-%02X-%02X-
%02X\n",
            *SrcIP, *(SrcIP + 1), *(SrcIP + 2), *(SrcIP + 3),
            *DstIP, *(DstIP + 1), *(DstIP + 2), *(DstIP + 3),
            SrcMAC[0], SrcMAC[1], SrcMAC[2], SrcMAC[3], SrcMAC[4],
SrcMAC[5],
            DstMAC[0], DstMAC[1], DstMAC[2], DstMAC[3], DstMAC[4],
DstMAC[5]);
    }
    else {
        fprintf(fp, "[forward IP] nextHop:%u.%u.%u.%u SrcMAC:%02X-%02X-%02X-
%02X-%02X-%02X DesMAC:%02X-%02X-%02X-%02X-%02X-%02X\n",
            *nextHop, *(nextHop + 1), *(nextHop + 2), *(nextHop + 3),
            SrcMAC[0], SrcMAC[1], SrcMAC[2], SrcMAC[3], SrcMAC[4],
SrcMAC[5],
```

```
            DstMAC[0], DstMAC[1], DstMAC[2], DstMAC[3], DstMAC[4],
DstMAC[5]);
    }
}
```

第4节 特殊情况分析

(一) ARP协议问题

实验中发现虚拟机出现如下图所示警告，且主机A（206.1.1.2）无法Ping通同一网段下的路由主机（206.1.1.1）



上网查阅后得到如下图结果，最初以为是由于程序发送的ARP报文过多导致问题的产生， 且重启后又恢复正常的网络连通性。

之后在调试中发现问题并不在此，查询主机A的ARP表后发现其中存储的路由主机对应的MAC地址为: **0f-0f-0f-0f-0f-0f**。

重新分析之前代码发现原来是由于获取本机MAC地址时，为了更准确的筛选出ARP响应报文，我在伪造的APR报文中填入了本机的IP地址，而根据ARP协议，这个伪造的ARP报文是广播的，主机A收到后会误以为这是正确的ARP报文，并将其中我伪造的源MAC地址当做路由主机真正的MAC地址进行存储，因此主机A中存储的路由主机的MAC地址一直为错误的地址，就无法正常与路由主机进行通信，网络连通性也就出了问题。

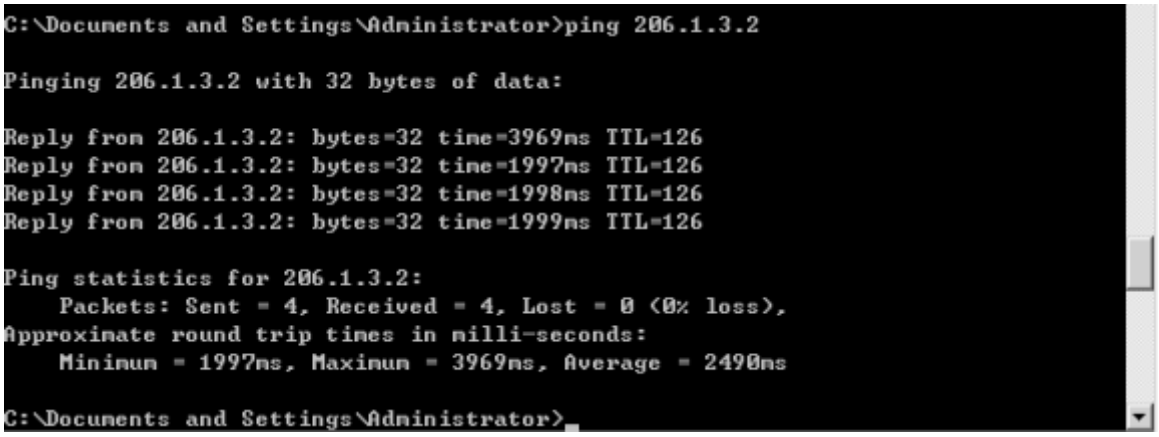
而对于第一张图中的IP地址冲突，可能也是由于前文所示MAC地址不对所产生的，此外也可能是虚拟机配置的问题，由于对该操作系统以及虚拟机的其他具体设置并不了解，因此无法确定该问题的确切原因。

(二)校验和问题

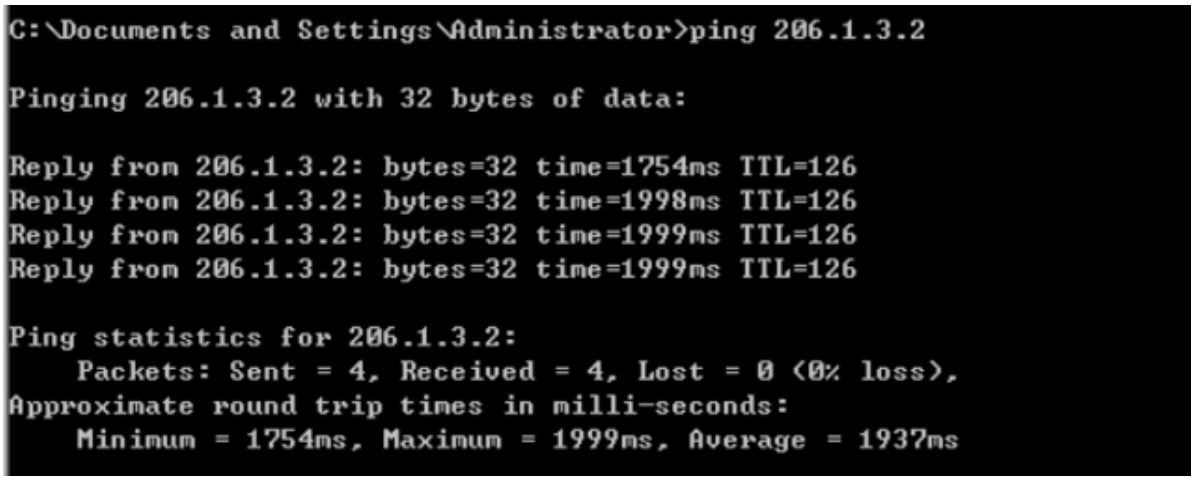
实验中使用wireShark捕获数据报后发现路由程序已经成功将报文重组并转发给下一跳步，且检查其中的目的MAC地址、目的IP地址等信息均正确，但仍旧无法Ping通，经过分析和调试后意识到在对数据包进行重组后其中的校验和已经失效，需重新设置校验和，否则其他主机在收到后无法通过校验，会直接丢弃。

重新设置校验和后成功Ping通。

(三)Ping过程中time相差较大



如上图可见，在主机A上对主机B使用Ping命令时发现耗时较长，且第一次Reply的time额外长，足足等待了将近4s才收到回复，而之后的time则相差不大，分析原因是由于最开始ARP表中无下一跳步的MAC地址，因此要先获取MAC地址，这部分操作耗时较长， 之后再进行转发时由于ARP表中已有MAC地址，无需再重新获取，因此时间缩短。



如上图可见，再次进行Ping命令发现之后的回复所用的time也都相差不大，说明分析正确。