lab5：简单路由器程序的设计实验报告

2112614 刘心源

一、实验内容

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。

二、实验准备

1、npcap库

Npcap是一个用于Windows操作系统的网络数据包捕获库，它是WinPcap的后继者。Npcap提供了多种用于网络数据包捕获和发送的功能。其中Packet.dll为内核级、低层次的包过滤动态连接库；wpcap.dll为高级别系统无关函数库。

2、搭建虚拟实验环境

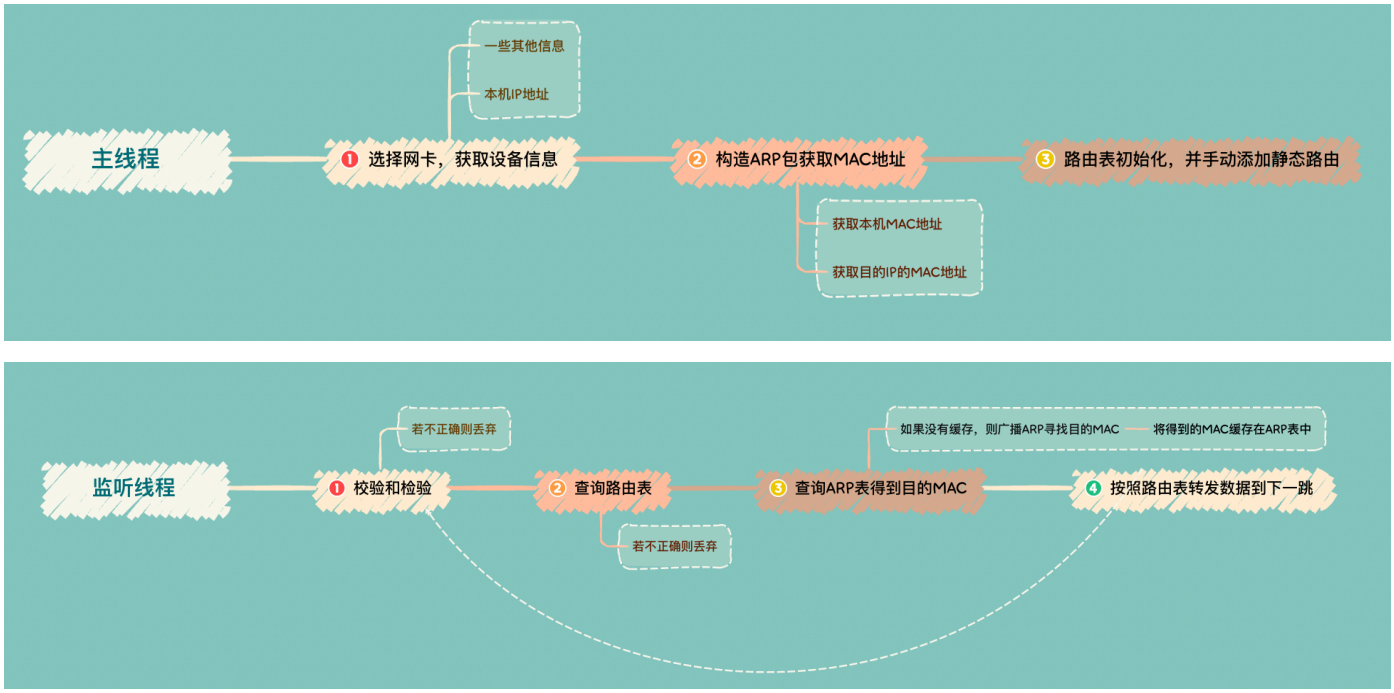
文件夹下共有四台虚拟机的压缩包，分别是1，2，3和4。

其中，1号设备和4号设备为终端设备，2号设备为你需要运行路由程序的设备，3号设备为另一台路由器。所有4台设备的IP地址已经全部分配好，所有4台设备已经安装好x86的VC++运行时环境。其中3号设备的路由功能已经全部开启，但仍需要每次开机后手动添加路由表项 `route ADD 206.1.1.0 MASK 255.255.255.0 206.1.2.1` 。

此外还需要而在自己实现的路由程序中，添加静态路由表项——目的网络：206.1.3.0；子网掩码：255.255.255.0；下一条地址：206.1.2.2。

三、实验过程

（1）整体流程图



（2）构建所需要的结构体

```
// 报文首部
typedef struct Ethernet_Header{
  BYTE ether_dhost[6]; // 目的地址
  BYTE ether_shost[6]; // 源地址
  WORD ether_type;   // 帧类型
} Ethernet_Header;
// ARP报文格式
typedef struct ARP_Header{
  Ethernet_Header FrameHeader;   // 帧首部
  WORD HardwareType;          // 硬件类型
  WORD ProtocolType;          // 协议类型
  BYTE HardwareSize;          // 硬件地址长度
  BYTE ProtocolSize;          // 协议地址
  WORD Operation;           // 操作
  BYTE SenderHardwareAddress[6]; // 发送方MAC
  DWORD SenderProtocolAddress;   // 发送方IP
  BYTE TargetHardwareAddress[6]; // 接收方MAC
  DWORD TargetProtocolAddress;   // 接收方IP
} ARP_Header;
// IP报文首部
typedef struct IP_Header{
  BYTE Ver_HLen;      // 版本+首部长度
  BYTE TOS;       // 服务类型
  WORD TotalLen;      // 总长度
  WORD ID;        // 标识
  WORD Flag_Segment; // 标志+片偏移
  BYTE TTL;       // 生命周期
  BYTE Protocol;      // 上层协议类型
  WORD Checksum;      // 校验和
  ULONG SrcIP;      // 源IP
  ULONG DstIP;      // 目的IP
} IP_Header;
typedef struct Data_t{
  // 包含帧首部和IP首部的数据包
  Ethernet_Header FrameHeader; // 帧首部
  IP_Header IPHeader;       // IP首部
} Data_t;
// ICMP报文首部
typedef struct ICMP_t{
  // 包含帧首部和IP首部的数据包
  Ethernet_Header FrameHeader; // 帧首部
  IP_Header IPHeader;       // IP首部
  // BYTE Type;
  // BYTE Code;
  // WORD Checksum;
  // WORD Identifier;
  // WORD SequenceNumber;
```

```
  char buf[0x80];
} ICMP_t;
// 路由表表项
class RouteItem{
public:
  DWORD Mask;    // 掩码
  DWORD TargetNet; // 目的网络
  DWORD nextIp;  // 下一跳的IP
  BYTE nextMAC[6]; // 下一跳的MAC
  int index;     // 索引
  int type;    // 0为直接连接，1为用户添加，0不可删除
  RouteItem *next_item;
  RouteItem(){
    memset(this, 0, sizeof(*this));
  }
  void PrintItem(); // 打印表项
};

// 使用链表
class RouteTable{
public:
  RouteItem *head, *tail; // 链表头尾
  int num;         // 转发表项数量
  RouteTable();      // 构造函数
  // 路由表的添加，直接投递在最前，前缀长的在前面
  void AddRouteItem(RouteItem *a);
  // 删除，type=0不能删除
  void RemoveRouteItem(int index);
  // 路由表的打印
  void printRouteTable();
  // 查找，最长前缀,返回下一跳的ip
  DWORD SearchNext(DWORD ip);
};

#pragma pack() // 恢复4bytes对齐
class ARP_Table
{
public:
  DWORD ip;                     // IP地址
  BYTE mac[6];                    // MAC地址
  static int num;                   // 表项数量
  static void addARPItem(DWORD ip, BYTE mac[6]);   // 插入表项
  static int SearchARPItem(DWORD ip, BYTE mac[6]); // 查找表项，返回索引
} arp_table[50];
```

Ethernet_Header 结构体

- 定义了以太网帧的首部。
- 包含目的地址 `ether_dhost`（6字节）、源地址 `ether_shost`（6字节）和帧类型 `ether_type`（2字节）。

ARP_Header 结构体

- 定义了ARP（地址解析协议）报文格式。
- 包含了一个 `Ethernet_Header`（表示它是以太网帧的一部分），以及ARP报文特有的字段：硬件类型、协议类型、硬件地址长度、协议地址长度、操作类型、发送方和目标方的MAC及IP地址。

IP_Header 结构体

- 定义了IP（互联网协议）报文的首部。
- 包含了版本和首部长度、服务类型、总长度、标识、标志和片偏移、生存时间、上层协议类型、校验和、源IP地址和目的IP地址。

Data_t 结构体

- 表示一个包含以太网首部和IP首部的数据包。
- 包含一个 `Ethernet_Header` 和一个 `IP_Header`。

ICMP_t 结构体

- 用于ICMP（互联网控制消息协议）报文。
- 类似于 `Data_t`，但增加了ICMP特有的字段（注释掉了）和一个大小为0x80字节的缓冲区 `buf`。

RouteItem 类

- 表示路由表中的一项。
- 包含掩码、目的网络、下一跳IP、下一跳MAC、索引和类型（直接连接、用户添加、不可删除）。
- 提供了打印路由表项的方法。

RouteTable 类

- 管理路由表，使用**链表**对于路由表项进行维护。
- 包含路由项的链表头尾和数量。
- 提供了添加、删除路由项和打印路由表的方法以及查找下一跳IP的功能。
    - 添加函数：`AddRouteItem(RouteItem *routeitem`

```
void RouteTable::AddRouteItem(RouteItem *routeitem){
  RouteItem *current;
  // 找到合适的地方
  if (!routeitem->type){//  0 直接投递
    routeitem->next_item = head->next_item;
    head->next_item = routeitem;
    routeitem->type = 0;
  }
  // 其它，按照掩码由长至短找到合适的位置
  else{
```

```
    for (current = head->next_item; current != tail && current->next_item !=
tail; current = current->next_item){// head有内容，tail没有
      if (routeitem->Mask < current->Mask && routeitem->Mask >= current-
>next_item->Mask || current->next_item == tail)
        break;
    }
    // 插入到合适位置
    routeitem->next_item = current->next_item;
    current->next_item = routeitem;
  }
  RouteItem *p = head->next_item;
  for (int i = 0; p != tail; p = p->next_item, i++) p->index = i;
  this->num++;
}
```

- 删除函数：`RemoveRouteItem`

```
void RouteTable::RemoveRouteItem(int index){
  for (RouteItem *item = head; item->next_item != tail; item = item->next_item)
{
    if (item->next_item->index == index){
      // 直接投递的路由表项不可删除
      if (item->next_item->type == 0){// 直接投递的路由表项不可删除
        cout << "该项为直接投递的路由表项，不可删除" << endl;
        return;
      }
      else{
        item->next_item = item->next_item->next_item;
        return;
      }
    }
  }
  cout << "该表项不存在" << endl;
}
```

- 查找函数:

```
DWORD RouteTable::SearchNext(DWORD ip){
  for (RouteItem *item = head->next_item; item != tail; item = item->next_item)
{
    if ((item->Mask & ip) == item->TargetNet) return item->nextIp;
  }
  return -1;
}
```

ARP_Table 类

- 管理ARP表，使用**数组**进行维护。
- 包含IP地址、MAC地址、表项数量。
- 提供了添加和搜索ARP表项的静态方法。
  - 添加函数：`addARPItem(DWORD ip, BYTE mac[6])`

```
void ARP_Table::addARPItem(DWORD ip, BYTE mac[6]){
  cout << "INFO: 开始添加ARP表项QQQ" << endl;
  arp_table[num].ip = ip;
  GetOtherMAC(ip, arp_table[num].mac);
  memcpy(mac, arp_table[num].mac, 6);
  num++; // 表项个数++
  cout << "SUCCESS:添加ARP表项成功QAQ" << endl;
  cout << "num: " << num << endl;
}
```

  - 搜索函数：`SearchARPItem(DWORD ip, BYTE mac[6])`

```
int ARP_Table::SearchARPItem(DWORD ip, BYTE mac[6]){
  cout << "INFO: 查找ARP表项ing" << endl;
  memset(mac, 0, 6);
  cout << "num: " << num << endl;
  for (int i = 0; i < num; i++){
    if (ip == arp_table[i].ip){
      cout << "SUCCESS: 找到对应的MAC地址QWWQ" << endl;
      memcpy(mac, arp_table[i].mac, 6);
      return 1;
    }
  }
  return 0;
}
```

同时编写了**校验和**的相关函数如下：

```
void setChecksum(IP_Header *response){
  response->Checksum = 0;
  uint32_t checkSum = 0;
  uint16_t *sec = (uint16_t *)response; // 每16位为一组
  int size = sizeof(IP_Header);
  while (size > 1){
    checkSum += *(sec++);
    size -= 2U;// 16位相加
  }
  if (size)
```

```
    checkSum += *(uint8_t *)sec;
  checkSum = (checkSum & 0xffff) + (checkSum >> 16);
  checkSum += (checkSum >> 16);
  response->Checksum = (uint16_t)~checkSum;// 取反
}

bool Checksum(IP_Header *response)
{
  uint32_t checkSum = 0;
  uint16_t *sec = (uint16_t *)response; // 每16位为一组
  bool checkOut = true;
  for (int i = 0; i < sizeof(IP_Header) / 2; i++){
    checkSum += sec[i];
    while (checkSum >= 0x10000){
      int c = checkSum >> 16;
      checkSum -= 0x10000;
      checkSum += c;
    }
  }
  if (sizeof(IP_Header) % 2 != 0){
    checkSum += *(uint8_t *)(sec + (sizeof(IP_Header) - 1));
    while (checkSum >= 0x10000){
      int c = checkSum >> 16;
      checkSum -= 0x10000;
      checkSum += c;
    }
  }
  checkOut = (checkSum == 0xffff) ? true : false;
  return checkOut;
}
```

（3）查找设备并选择网卡

```
void find_alldevs(){
  if (pcap_findalldevs_ex(pcap_src_if_string, NULL, &alldevs, errbuf) == -1)
    printf("%s", "error");
  else{
    int i = 0;
    for (device = alldevs; device != NULL; device = device->next){// 获取该网络接口设备的
ip地址信息
      if (i == index){
        int t = 0;
        for (a = device->addresses; a != nullptr; a = a->next){
          if (((struct sockaddr_in *)a->addr)->sin_family == AF_INET && a->addr){
            printf("输入选择设备的序号:\n");
            printf("%d ", i);
            printf("%s\t", device->name, device->description);
```

```
            printf("%s\t%s\n", "IP地址:", inet_ntoa(((struct sockaddr_in *)a->addr)->sin_addr));
            // 存储对应IP地址与MAC地址
            strcpy(ip[t], inet_ntoa(((struct sockaddr_in *)a->addr)->sin_addr));
            strcpy(mask[t++], inet_ntoa(((struct sockaddr_in *)a->netmask)->sin_addr));
          }
        }
        ahandle = pcap_open(device->name, 65536, PCAP_OPENFLAG_PROMISCUOUS, 100, NULL, errbuf);
      }
      i++;
    }
  }
  pcap_freealldevs(alldevs);
}
```

1. **查找网络设备**:
   - 使用 `pcap_findalldevs_ex` 函数来查找系统中所有的网络接口设备。
   - 如果这个函数调用失败（返回-1），则打印错误消息。
2. **遍历设备列表**:
   - 遍历由 `pcap_findalldevs_ex` 返回的设备列表。
   - `alldevs` 是一个指向网络设备列表的指针。
   - `device` 是一个遍历这个列表的指针。
3. **打印指定设备的IP信息**:
   - 当找到与指定索引 `index` 对应的设备时，遍历该设备的IP地址列表。
   - 对于每个IP地址（只考虑IPv4地址，即 `AF_INET` 类型），打印出设备的名称、描述和IP地址。
   - IP地址使用 `inet_ntoa` 函数转换为可读的字符串格式。
4. **存储IP地址和掩码信息**:
   - 将找到的IP地址和对应的网络掩码存储在数组 `ip` 和 `mask` 中。
5. **打开一个用于捕获的网络接口**:
   - 使用 `pcap_open` 打开找到的网络设备，准备进行数据包捕获。
   - 该设备被设置为混杂模式（`PCAP_OPENFLAG_PROMISCUOUS`），允许捕获经过网络接口的所有数据包。
6. **释放设备列表**:
   - 使用 `pcap_freealldevs` 释放设备列表，以避免内存泄漏。

（4）构造ARP包获取MAC地址

与上一次ARP映射的实验类似，编写代码如下：

获取本机MAC

通过发送一个广播ARP请求并解析其回复来获取本地计算机的MAC地址。

```
void GetSelfMAC(DWORD ip) // 获得本地IP地址以及对应的MAC地址
{
  memset(selfmac, 0, sizeof(selfmac));
  ARP_Header ARPFrame;
  // 将APRFrame.FrameHeader.DesMAC设置为广播地址
  for (int i = 0; i < 6; i++) ARPFrame.FrameHeader.ether_dhost[i] = 0xff;

  ARPFrame.FrameHeader.ether_shost[0] = 0x0f;
  ARPFrame.FrameHeader.ether_shost[1] = 0x0f;
  ARPFrame.FrameHeader.ether_shost[2] = 0x0f;
  ARPFrame.FrameHeader.ether_shost[3] = 0x0f;
  ARPFrame.FrameHeader.ether_shost[4] = 0x0f;
  ARPFrame.FrameHeader.ether_shost[5] = 0x0f;
  ARPFrame.FrameHeader.ether_type = htons(0x0806); // 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.HardwareSize = 6;          // 硬件地址长度为6
  ARPFrame.ProtocolSize = 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.SenderHardwareAddress[i] = 0x0f
  ARPFrame.SenderProtocolAddress = inet_addr("122.122.122.122");
  for (int i = 0; i < 6; i++) ARPFrame.TargetHardwareAddress[i] = 0;
  ARPFrame.TargetProtocolAddress = ip;
  if (ahandle == nullptr) printf("网卡接口打开错误\n");
  else{
    if (pcap_sendpacket(ahandle, (u_char *)&ARPFrame, sizeof(ARP_Header)) != 0)
      printf("senderror\n");
    else{
      cout << "发送数据包成功" << endl;
      while (1){
        struct pcap_pkthdr *pkt_header;
        const u_char *pkt_data;
        int ret = pcap_next_ex(ahandle, &pkt_header, &pkt_data);
        if (ret > 0){
          if (*(uint16_t *)(pkt_data + 12) == htons(0x0806) && *(uint16_t *)(pkt_data +
20) == htons(2) && *(uint32_t *)(pkt_data + 28) == ARPFrame.TargetProtocolAddress){
            cout << "SUCCESS: 成功获得自身主机的MAC地址qqq" << endl;
            cout << "INFO: MAC是: ";
            for (int i = 0; i < 6; i++){
              selfmac[i] = *(uint8_t *)(pkt_data + 22 + i);
              printf("%02x", selfmac[i]);
              if (i != 5) cout << ":";
            }
            cout << endl;
            break;
```

```
            }
        }
    }
}
}
```

函数定义了一个名为 `GetSelfMAC` 的函数，其目的是获取与给定本地IP地址相对应的MAC地址。函数使用了 `pcap` 库进行网络包捕获和发送。下面是对这个函数的详细解释：

1. **构建ARP请求帧**:
    - 创建 `ARP_Header` 结构体的实例 `ARPFrame`。
    - 设置目的MAC地址为广播地址（`0xff`），这意味着ARP请求会被网络上的所有设备接收。
    - 设置源MAC地址为 `0x0f:0x0f:0x0f:0x0f:0x0f:0x0f`（这个MAC地址似乎是随意设置的，通常应该使用实际的MAC地址）。
    - 设置帧类型为ARP（0x0806）。
    - 设置硬件类型为以太网（0x0001）和协议类型为IP（0x0800）。
    - 设置硬件地址长度为6（MAC地址）和协议地址长度为4（IP地址）。
    - 设置操作类型为ARP请求（0x0001）。
    - 设置发送方的硬件地址（与源MAC地址相同）和协议地址（这里使用了一个固定的IP地址`122.122.122.122`，通常应该使用实际的IP地址）。
    - 设置目标硬件地址为空（用于ARP请求）和目标协议地址为函数参数提供的IP地址。
2. **发送ARP请求**:
    - 检查网络接口句柄 `ahandle` 是否有效。
    - 使用 `pcap_sendpacket` 发送构建的ARP请求。
3. **捕获响应并解析MAC地址**:
    - 循环调用 `pcap_next_ex` 来捕获网络上的响应。
    - 检查捕获的数据包是否是对ARP请求的回复（帧类型为ARP，操作为应答，目标IP地址与发送的ARP请求中的目标IP相同）。
    - 如果找到匹配的响应，则提取回复中的发送方MAC地址，并存储在 `selfmac` 数组中。
    - 打印出获得的MAC地址。

获取目的MAC

过程与上面类似，就是将源MAC地址变为上面得到的MAC地址，不赘述。

```
void GetOtherMAC(DWORD dest_ip, BYTE mac[]){
  memset(mac, 0, sizeof(mac));
  struct pcap_pkthdr *pkt_header;
  const u_char *pkt_data;
  ARP_Header ARPFrame;
  int flag;
  int ret = 1;
  ARPFrame.FrameHeader.ether_type = htons(0x0806);
  memset(ARPFrame.FrameHeader.ether_dhost, 0xff, 6);
```

```
  memset(ARPFrame.TargetHardwareAddress, 0x00, 6);
  for (int i = 0; i < 6; i++)
    ARPFrame.FrameHeader.ether_shost[i] = ARPFrame.SenderHardwareAddress[i] =
selfmac[i];
  // 将APRFrame.FrameHeader.SrcMAC设置为本机网卡的MAC地址
  ARPFrame.HardwareType = htons(0x0001); // 硬件类型为以太网
  ARPFrame.ProtocolType = htons(0x0800); // 协议类型为IP
  ARPFrame.HardwareSize = 6;          // 硬件地址长度为6
  ARPFrame.ProtocolSize = 4;          // 协议地址长为4
  ARPFrame.Operation = htons(0x0001);     // 操作为ARP请求
  ARPFrame.SenderProtocolAddress = inet_addr(ip[0]);
  ARPFrame.TargetProtocolAddress = dest_ip;
  if (ahandle == nullptr)
    cout << "ERROR!: 网卡接口打开错误T_T~"<< endl;
  else{
    if (pcap_sendpacket(ahandle, (u_char *)&ARPFrame, sizeof(ARP_Header)) != 0)
      cout << "ERROR: send ERROR!" << endl;
    else{ // 发送成功
      cout << "SUCCESS: 成功发送ARP数据包!!!!QAQ" << endl;
      while ((flag = pcap_next_ex(ahandle, &pkt_header, &pkt_data) > 0)){
        if (*(uint16_t *)(pkt_data + 12) == htons(0x0806) // arp 以太帧的上层协议类型
          && *(uint16_t *)(pkt_data + 20) == htons(2)   // 响应 arp操作类型
          && *(uint32_t *)(pkt_data + 28) == ARPFrame.TargetProtocolAddress){
          // ip正确
          cout << "SUCCESS: 目标MAC获取成功!!!!QWQ" << endl;
          cout << "INFO: 目标MAC为: " << endl;
          for (int i = 0; i < 6; i++){
            mac[i] = *(uint8_t *)(pkt_data + 22 + i);
            printf("%02x", mac[i]);
            if (i != 5) cout << ":";
          }
          cout << endl;
          break;
        }
      }
    }
  }
}
```

（5）路由器实现转发功能

```
void PacketResend(ICMP_t icmpdata, BYTE destinationMAC[]){
  cout << "INFO: 进入转发函数!!" << endl;
  ICMP_t *icmp = (ICMP_t *)&icmpdata;
  Data_t *changed_icmp = (Data_t *)&icmpdata;
  memcpy(icmp->FrameHeader.ether_shost, icmp->FrameHeader.ether_dhost, 6); // 源MAC为本
机MAC
```

```
  memcpy(icmp->FrameHeader.ether_dhost, destinationMAC, 6); // 目的MAC为下一跳MAC
  icmp->IPHeader.TTL -= 1;
  cout << "INFO: TTL = " << icmp->IPHeader.TTL << endl;
  //  如果TTL小于0，则丢弃
  if (icmp->IPHeader.TTL < 0)
    return;
  setChecksum(&(icmp->IPHeader));
  cout << "SUCCESS: 设置校验和成功!" << endl;                    // 重新设置校验和
  int ret = pcap_sendpacket(ahandle, (const u_char *)icmp, sizeof(ICMP_t)); // 发送数据报
  if (ret == 0)
    cout << "INFO: 转发" << changed_icmp << endl;
}
```

这个函数修改ICMP数据包的首部，并将其转发到指定的下一跳地址。

1. **处理ICMP数据包**:
   - 创建一个 `ICMP_t` 类型指针 `icmp`，指向传入的 `icmpdata`。
2. **修改MAC地址**:
   - 将源MAC地址（`ether_shost`）设置为原目的MAC地址（`ether_dhost`），也就是本机的MAC地址。
   - 将目的MAC地址（`ether_dhost`）设置为 `destinationMAC`，也就是下一跳地址。
3. **处理TTL（生存时间）**:
   - 将IP首部中的TTL减1。
   - 打印新的TTL值。
   - 如果TTL小于0，函数返回，不再转发该数据包。这是为了防止数据包在网络中无限循环。
4. **重新计算校验和**:
   - 调用 `setChecksum` 函数来重新计算IP首部的校验和。
   - 打印成功设置校验和的信息。
5. **发送数据包**:
   - 使用 `pcap_sendpacket` 函数发送修改后的数据包。
   - 如果发送成功，打印转发信息。

（6）监听线程的实现

```
DWORD WINAPI handlePacket(LPVOID lparam){// 接收和处理线程函数
  RouteTable routetable = *(RouteTable *)(LPVOID)lparam; // 将路由表传入
  while (1){
    pcap_pkthdr *pkt_header;
    const u_char *pkt_data;
    // 一直接受packet
    while (1){
      int ret = pcap_next_ex(ahandle, &pkt_header, &pkt_data);
      if (ret) break;
    }
    Ethernet_Header *header = (Ethernet_Header *)pkt_data;
```

```
    // 如果目的mac是自己的mac
    if (compare(header->ether_dhost, selfmac)){
      if (ntohs(header->ether_type) == 0x800) { // 是IP数据包
        Data_t *data = (Data_t *)pkt_data; // 只提取首部
        ICMP_t *icmp = (ICMP_t *)pkt_data; // 提取首部和数据
        DWORD ip1_ = data->IPHeader.DstIP;
        DWORD ip2 = routetable.SearchNext(ip1_); // 查找是否有对应表项
        if (ip2 == -1) continue;   // 如果没有则直接丢弃或直接递交至上层
        if (Checksum(&(icmp->IPHeader))) { // 如果校验和不正确，则直接丢弃不进行处理
          cout << "SUCCESS: 校验和正确!!!" << endl;
          if (data->IPHeader.DstIP != inet_addr(ip[0]) && data->IPHeader.DstIP !=
inet_addr(ip[1])) { // 将目的IP与预设的两个IP比对，需要路由
            cout << "INFO: 需要路由" << endl;
            int t1 = compare(data->FrameHeader.ether_dhost, broadcast);
            int t2 = compare(data->FrameHeader.ether_shost, broadcast);
            if (!t1 && !t2){
              cout << "INFO: t1,t2都不是广播地址" << endl;
              // ICMP报文包含IP数据包报头和其它内容
              ICMP_t *temp_ = (ICMP_t *)pkt_data;
              ICMP_t temp = *temp_;
              BYTE mac[6];
              if (ip2 == 0){
                //  如果ARP表中没有所需内容，则需要获取ARP
                if (!ARP_Table::SearchARPItem(ip1_, mac))
                  ARP_Table::addARPItem(ip1_, mac);
                PacketResend(temp, mac);
                cout << "SUCCESS: 数据包转发成功!!" << endl;
              }
              if (ip2 != -1) { // 非直接投递，查找下一条IP的MAC
                if (!ARP_Table::SearchARPItem(ip2, mac))
                  ARP_Table::addARPItem(ip2, mac);
                PacketResend(temp, mac);
                cout << "SUCCESS: 数据包转发成功!!" << endl;
              }
            }
          }
        }
      }
    }
  }
}
```

四、实验结果

（1）ping

主机1和主机4可以成功ping通。

使用wireshark抓包可以发现得到来自206.1.3.2的ICMP响应。

（2）路由器日志输出

1. 启动时获取网卡信息
2. 手动添加静态路由
3. 转发数据包的日志信息

路由器成功处理主机1(206.1.3.2)发出的数据包，该数据包目标IP是206.1.1.2。经过路由表的检索，发现下一跳是206.1.2.2，这个地址在路由表中直接连接。

五、实验总结

路由表项的数据结构优化

在之前的某次小组讨论题中其实已经有了一个很简陋的优化方案：

> 为了查询效率，可以考虑使用前缀树（字典树）这个数据结构，其特点如下：
>
> - 前缀树中每个节点代表一个字符或一部分字符串。从根节点到任一节点的路径上的字符连接起来，形成该节点对应的字符串。
> - 如果多个字符串有共同的前缀，它们会共享前缀树中的相同路径。这意味着每个唯一前缀只被存储一次。
> - 每个节点通常包含其子节点的链接
> - 在最佳情况下，查找操作的时间复杂度与树的深度成线性关系。
>
> **树节点结构：**

```
struct TreeNode{
  char particalNum;
  tableItem *p;
  treeNode * children[16]; // 储存16进制的数，有16个孩子，索引位i的指针指向下一层particalNum值为i的节点
  uint_8 floor; //表示节点所在层数
  bool isEnd;
  bool isEnd(){
    uint_32 bit=p->getNetBit();
    if ceil((bit/(double)4)) return true;
    return false;
}
```

用一个字符表示十六进制网络号的一位，p指针指向一个路由表项结构体对象

路由表项的结构：

```
struct TableItem{
  //新增
  getNetBit(); // 获取这个路由表项对应网络号位数（简单解析mask就行）
}
```

树结构以及查找和插入算法的伪代码：

```
struct Trie {
  TrieNode* root; // 根节点指针
  Trie() {
    root = new TreeNode();
  }
  void insert(TableItem item){
    currentNode = root
      for each hexDigit in item.destNet (十六进制表示供8位):
    if currentNode.children[hexDigit] is null:
    create a new treeNode
      currentNode.children[hexDigit] = new treeNode
      currentNode = currentNode.children[hexDigit]
      currentNode.p = tableItem
  };
  void search(uint_32 destNet,uint_32 mask){
    currentNode = root
      for each hexDigit in destNet(十六进制表示供8位):
    if currentNode.children[hexDigit] is null:
    return null
      currentNode = currentNode.children[hexDigit]
      return currentNode.p
  }
};
```

> 网络号32位，用8个十六进制数表示xx:yy:zz:ww，树节点中的`particalNum`是一个x/y/z/w，所以用一个8层的树就可以维护32位的目的网络号，空间是2的32字节也就是4G内存。但实际上，**空间复杂度会小得多**，因为
>
> - 不可能一个路由表要储存世界上绝大多数网络的路由信息。
> - 网路有子网掩码，通常16位/24位就可以表示一个网络，证明字典树不是满树，有的叶节点在第4层或第6层。
> - 路由器管理一个局域网，路由表现大多具有公共前缀。

已经有一个非常简陋的demo在了……

~~尝试实现完整的Trie算法，~~

等到熬过这堆ddl再进行更新✅

~~编译作业太多绷不住半点~~

六、Github链接

一个悲伤的链接