简单路由器程序的设计

王泽舜
学号：2310655

December 24, 2025

1 实验要求

本实验要求设计并实现一个简单的软件路由器程序，使其能与现有路由器产品协同工作。主要功能包括：

1. 实现 IP 数据报的获取、选路和投递等基本功能
2. 支持路由表的手工插入和删除
3. 维护 ARP 缓存表用于 IP-MAC 地址映射
4. 记录数据报接收和转发过程的工作日志
5. 验证 IP 首部校验和，转发前重新计算校验和

2 宏观设计框架

路由器程序采用三层递进式设计：**设备信息初始化 → 表项维护实现 → 数据转发核心流程**。这种架构确保每一层都有明确的职责边界。

2.1 整体工作流程

初始化阶段	运行阶段	转发阶段
枚举网络设备	路由表维护	接收数据包
用户选择网卡	ARP 缓存维护	校验和验证
获取本机 MAC	创建转发线程	路由表查询
初始化路由表	用户交互循环	MAC 地址查询
		修改转发数据包

3 初始化设计：建立运行环境

3.1 设备信息获取

程序启动时通过 Npcap 库枚举所有可用网络接口，并存储 IP 地址和子网掩码。

Listing 1: 网络设备枚举

```
if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL,
                        &alldevs, errbuf) == -1) {
    printf("Error getting local devices\n");
    return 0;
}

for (d = alldevs; d != NULL; d = d->next) {
    for (a = d->addresses; a != NULL; a = a->next) {
        if (a->addr->sa_family == AF_INET) {
            strcpy(ip[t], inet_ntoa(...));
            strcpy(mask[t], inet_ntoa(...));
        }
    }
}
```

用户从枚举列表中选择网卡后，程序使用 pcap_open() 打开设备。采用混杂模式（PCAP_OPENFLAG_PROMISCUOUS）

3.2 本机 MAC 地址获取

由于 Windows 环境下无法直接获取网卡 MAC 地址，我们用本机网卡发送一个伪造的 ARP 请求，本机驱动程序会自动回复，从而获得真实的 MAC 地址。

Listing 2: 伪造 ARP 请求获取本机 MAC

```
ARPFrame_t ARPFrame;
ARPFrame.FrameHeader.DesMAC[i] = 0xff;          // 广播地址
ARPFrame.FrameHeader.FrameType = htons(0x0806);// ARP帧
ARPFrame.SendIP = inet_addr("1.2.3.4");         // 虚构IP
ARPFrame.RecvIP = inet_addr(ip[0]);             // 本机实际IP

pcap_sendpacket(ThisHandle, (u_char*)&ARPFrame,
                sizeof(ARPFrame_t));

// 等待并捕获ARP应答
while (1) {
    pcap_next_ex(ThisHandle, &pkt_header, &pkt_data);
    IPPacket = (ARPFrame_t*)pkt_data;
    if (ntohs(IPPacket->FrameHeader.FrameType) == 0x0806 &&
        ntohs(IPPacket->Operation) == 0x0002) {
        for(int i = 0; i < 6; i++)
            SrcMac[i] = IPPacket->FrameHeader.SrcMAC[i];
        break;
    }
}
```

这个设计巧妙地解决了”先有鸡还是先有蛋”的循环依赖问题：网卡驱动程序会自动识别发往本机 IP 的 ARP 请求，并在应用程序之外生成应答，因此无需显式处理。

3.3 初始化路由表

路由表在创建时自动添加两个直接连接的网络条目。这些条目对应本机的两个网卡接口，类型标记为 0（不可删除）。

Listing 3: 路由表初始化

```
RouterTable::RouterTable() {
    head = new RouterItem;
    tail = new RouterItem;
    head->nextitem = tail;
    num = 0;

    for (int i = 0; i < 2; i++) {
        RouterItem* temp = new RouterItem;
        temp->net = inet_addr(ip[i]) & inet_addr(mask[i]);
        temp->mask = inet_addr(mask[i]);
        temp->type = 0;  // 直接连接，不可删除
        this->RouterAdd(temp);
    }
}
```

4 路由表与 ARP 缓存维护

4.1 路由表结构与最长前缀匹配

路由表采用带哨兵的链表存储，按子网掩码长度从长到短排序。这种设计直接实现了最长前缀匹配：从表头开始遍历，第一个匹配的条目即为最佳匹配。

Listing 4: 最长前缀匹配

```
DWORD RouterTable::RouterFind(DWORD ip) {
    for (RouterItem* t = head->nextitem; t != tail;
         t = t->nextitem) {
        if ((t->mask & ip) == t->net) {
            return t->nextip;  // 返回下一跳IP
        }
    }
    return -1;  // 无匹配路由
}
```

用户可以通过程序的交互菜单手动添加路由表项。新添加的条目按掩码长度插入相应位置，保持有序性。

Listing 5: 添加路由表项

```
else if(op == 2) {
    RouterItem ri;
    char temp[30];
    printf("Enter destination network: ");
    scanf("%s", &temp);
    ri.net = inet_addr(temp);
```

```
    printf("Enter netmask: ");
    scanf("%s", &temp);
    ri.mask = inet_addr(temp);

    printf("Enter next hop address: ");
    scanf("%s", &temp);
    ri.nextip = inet_addr(temp);
    ri.type = 1;  // 用户添加
    RT.RouterAdd(&ri);
}
```

4.2 ARP 缓存机制

ARP 缓存是一个全局数组，存储 IP 地址到 MAC 地址的映射。转发过程中，如果缓存中没有所需的 MAC 地址，程序会发送 ARP 请求并将结果缓存，避免重复查询。

Listing 6: ARP 缓存查询与插入

```
if (!ArpTable::FindArp(nextip, mac)) {
    // 缓存未命中：发送ARP请求
    ArpTable::InsertArp(nextip, mac);
}

static void InsertArp(DWORD ip, BYTE mac[6]) {
static void InsertArp(DWORD ip, BYTE mac[6]) {
    arptable[num].ip = ip;
    SendARP(ip, arptable[num].mac);  // 发ARP请求
    memcpy(mac, arptable[num].mac, 6);
    num++;
}

static int FindArp(DWORD ip, BYTE mac[6]) {
    for (int i = 0; i < num; i++) {
        if (ip == arptable[i].ip) {
            memcpy(mac, arptable[i].mac, 6);
            return 1;  // 找到
        }
    }
    return 0;  // 未找到
}
```

5 转发线程：数据包处理核心

5.1 线程函数架构

转发线程在后台持续监听网络接口，对每个接收到的报文执行五步处理：

Listing 7: 转发线程五步处理

```
DWORD WINAPI Thread(LPVOID lparam) {
    RouterTable RT = *(RouterTable*)(LPVOID)lparam;
```

```
    while (1) {
        // 步骤1: 阻塞接收报文
        pcap_pkthdr* pkt_header;
        const u_char* pkt_data;
        while (pcap_next_ex(ThisHandle, &pkt_header,
                            &pkt_data) == 0);

        // 步骤2: MAC目的地址过滤
        FrameHeader_t* header = (FrameHeader_t*)pkt_data;
        if (!Compare(header->DesMAC, SrcMac))
            continue;  // 不是发给我的

        // 步骤3: 帧类型检查与路由查询
        if (ntohs(header->FrameType) == 0x0800) {
            Data_t* data = (Data_t*)pkt_data;
            DWORD dstip = data->IPHeader.DstIP;
            DWORD IFip = RT.RouterFind(dstip);

            if (IFip == -1) continue;  // 无路由

            // 步骤4: 校验和验证
            if (!CheckSum(data)) continue;

            // 步骤5: ARP查询与转发
            BYTE mac[6];
            if (!ArpTable::FindArp(IFip, mac))
                ArpTable::InsertArp(IFip, mac);
            resend(data, mac);
        }
    }
}
```

5.2 数据包修改与发送

resend() 函数负责修改报文的 MAC 地址字段、递减 TTL，以及重新计算校验和。

Listing 8: 数据包转发与修改

```
void resend(ICMP_t data, BYTE desmac[]) {
    Data_t* temp = (Data_t*)&data;

    // 交换源目MAC地址
    memcpy(temp->FrameHeader.SrcMAC,
           temp->FrameHeader.DesMAC, 6);
    memcpy(temp->FrameHeader.DesMAC, desmac, 6);

    // TTL递减与检查
    temp->IPHeader.TTL -= 1;
    if (temp->IPHeader.TTL < 0) return;

    // 重新计算校验和
```

```
    SetCheckSum(temp);

    // 发送报文
    pcap_sendpacket(ThisHandle, (const u_char*)temp, 74);
    LT.WritelogIP("转发", temp);
}
```

5.3 IP 首部校验和处理

IP 首部校验和采用 16 位反码求和算法。发送端计算并填充校验和字段，接收端验证；转发时必须重新计算。

Listing 9: 校验和计算与验证

```
bool CheckSum(Data_t* temp) {
    unsigned int sum = 0;
    WORD* t = (WORD*)&temp->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;
        }
    }
    return (sum == 65535);
}

void SetCheckSum(Data_t* temp) {
    temp->IPHeader.Checksum = 0;
    unsigned int sum = 0;
    WORD* t = (WORD*)&temp->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;
        }
    }
    temp->IPHeader.Checksum = ~sum;
}
```

验证原理：正确的 IP 数据包中，所有 16 位字（包括校验和）相加的结果应为 0xFFFF。

6 测试结果

6.1 网络拓扑

本实验在虚拟环境中进行。网络拓扑配置如下：

- **终端 1**：206.1.1.2/24
- **路由器 2（本程序）**：206.1.1.1 和 206.1.2.1
- **路由器 3**：206.1.2.2 和 206.1.3.1
- **终端 4**：206.1.3.2/24

网络拓扑结构 - 终端 1 和终端 4 分别连接到路由器 2 和路由器 3，两台路由器通过 206.1.2.0/24 网段相连

6.2 程序运行界面

程序启动后枚举所有可用网络接口，显示设备名、描述、IP 地址和子网掩码。用户选择要使用的网卡后，程序自动获取本机 MAC 地址并初始化路由表。

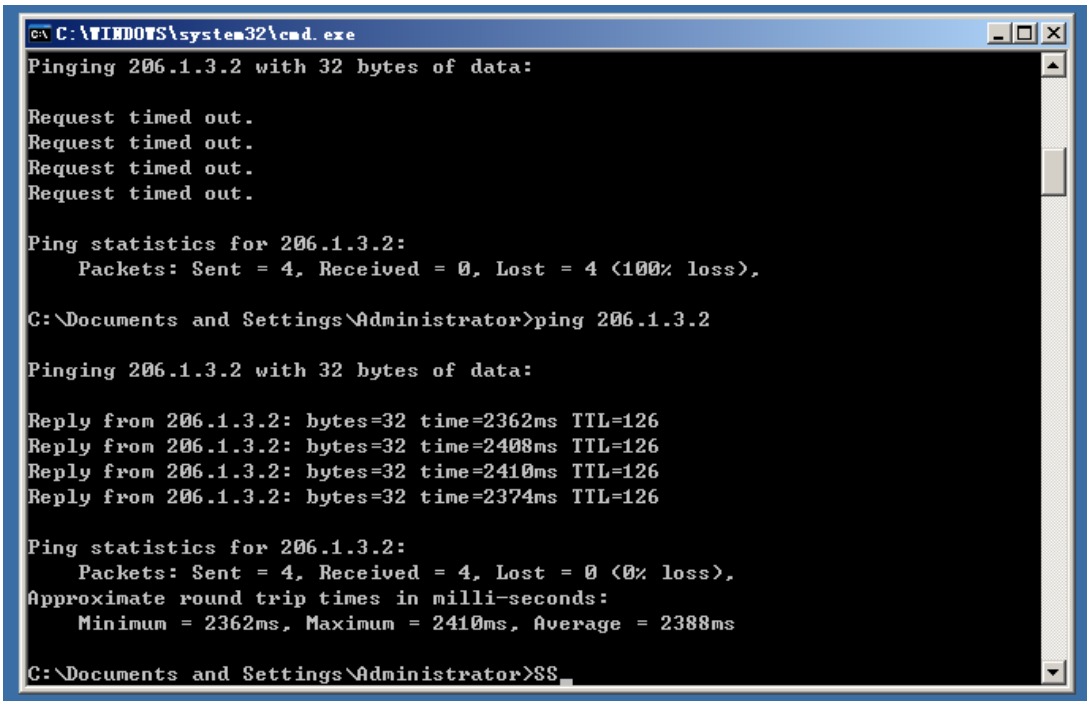

Figure 1: 终端 1 ping 终端 4 测试

6.3 路由表维护

初始路由表包含两个直接连接的网络。通过菜单选项 2 可以添加静态路由，例如添加到 206.1.3.0/24 网段的路由。

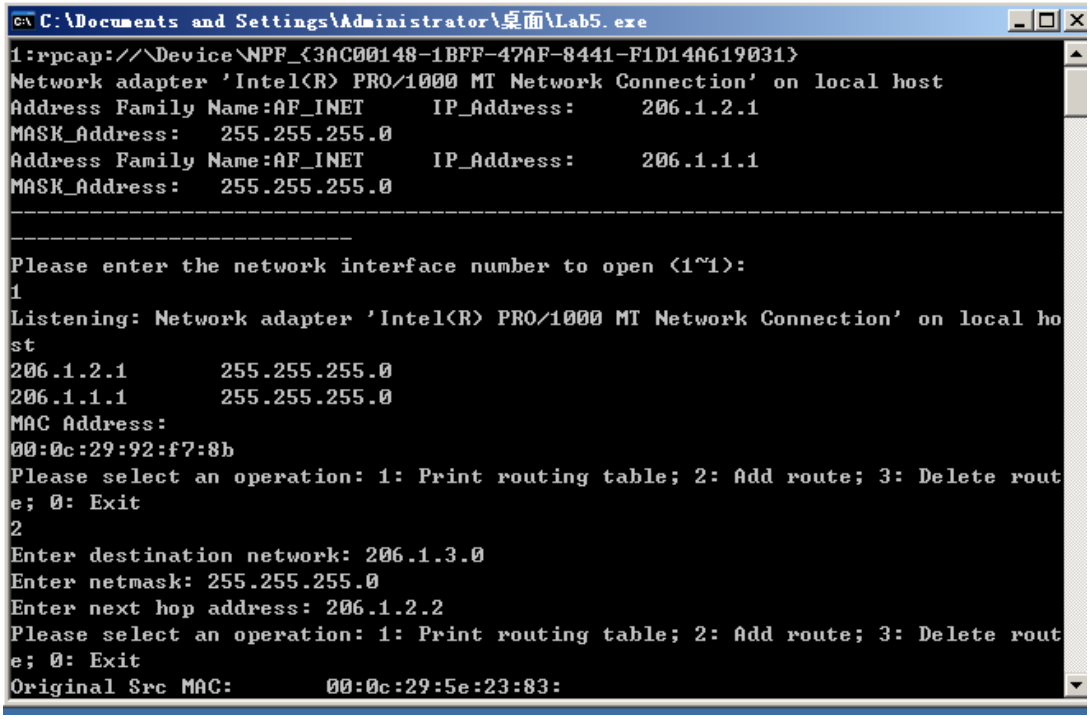

Figure 2: 在路由器 2 上添加静态路由表项

6.4 数据转发测试

6.4.1 终端 1 ping 终端 4

从终端 1 向终端 4 发送 ICMP 请求时，本路由器需要进行转发。转发过程如下：

1. 终端 1 向 206.1.3.2 发送 ICMP 请求
2. 本路由器在 206.1.1.1 网卡接收报文
3. 查路由表：206.1.3.0/24 → 下一跳 206.1.2.2
4. 查 ARP 缓存获取 206.1.2.2 的 MAC 地址
5. 修改源 MAC 为 206.1.2.1 的 MAC，目的 MAC 为 206.1.2.2 的 MAC
6. 递减 TTL，重新计算校验和
7. 从 206.1.2.1 网卡转发出去

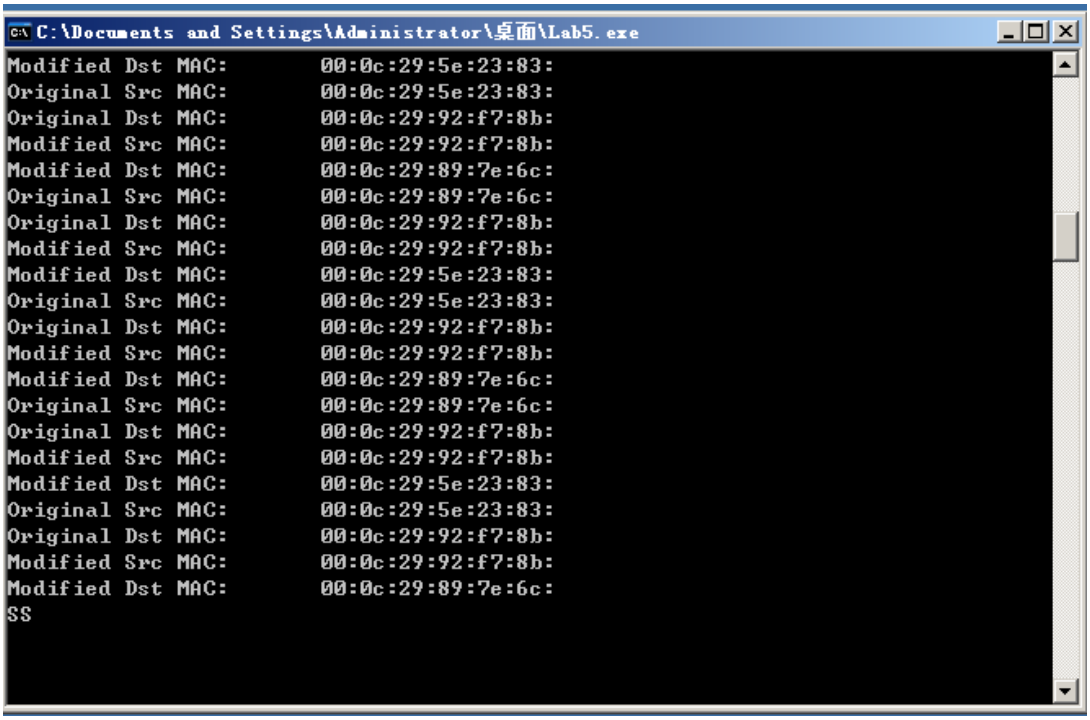

Figure 3: 路由器 2 转发终端 1 ping 终端 4 的流量

6.4.2 终端 4 ping 终端 1

反向转发过程中，终端 4 发往终端 1 的报文同样需要经过本路由器转发。

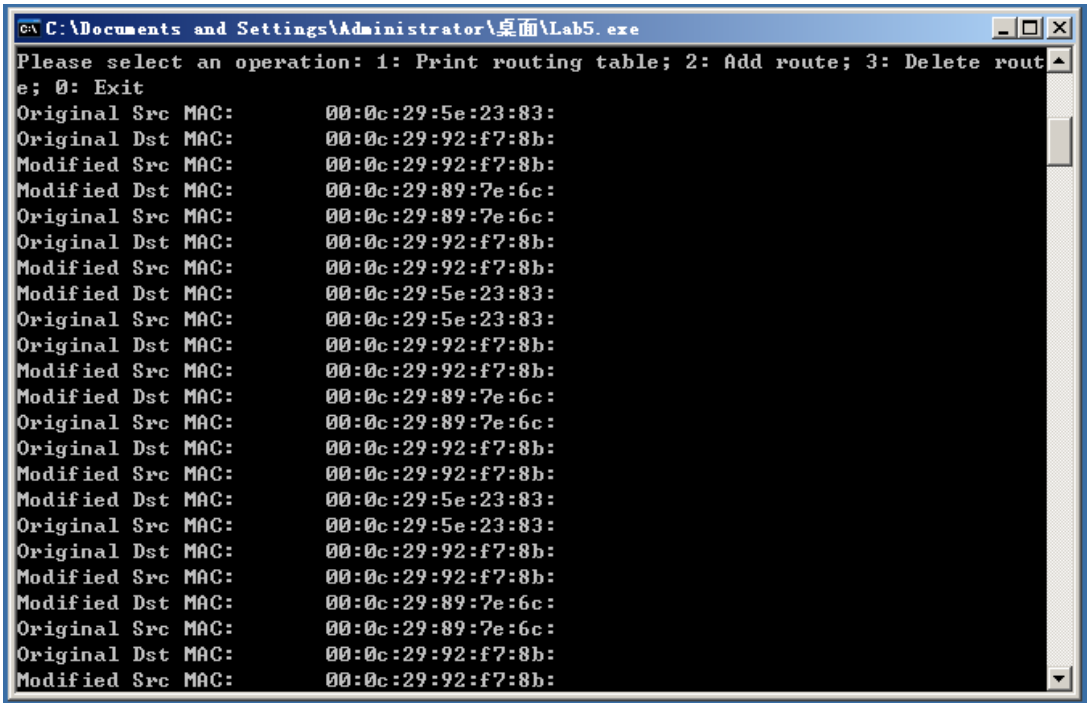

Figure 4: 路由器 2 转发终端 4 ping 终端 1 的流量

6.4.3 反向验证

从终端 4 的角度验证回程流量的正确接收：

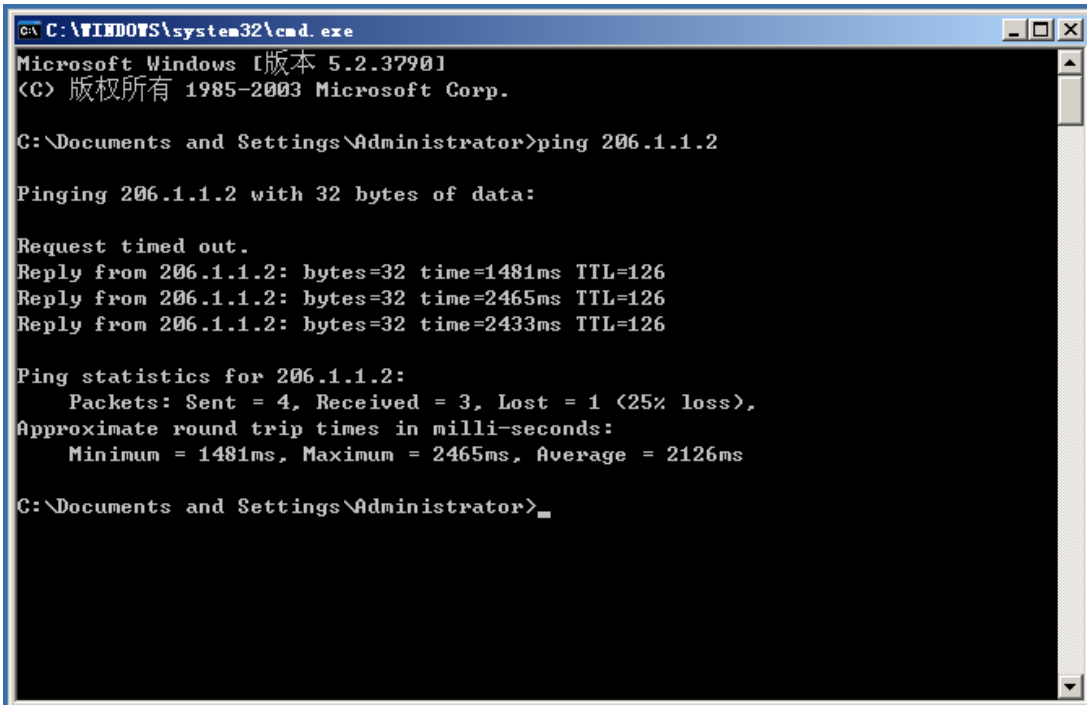

Figure 5: 终端 4 接收来自终端 1 的 ICMP 应答

7 总结

本实验成功设计并实现了一个简单的软件路由器，该程序能与真实路由器协同工作，通过验证可以正确转发跨域数据包。在实际应用中，还可以加入超时机制、优先级队列、多线程同步等特性进一步完善。