南開大學

网络技术与应用课程实验报告

实验五：简单路由器程序的设计

学　　院　密码与网络空间安全学院

专　　业　信息安全、法学双学位班

学　　号　2313815

姓　　名　段俊宇

班　　级　信息安全、法学双学位班

一、实验目的

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、实验原理

1. 数据包发送方法

NPcap 提供了 pcap_sendpacket()函数来发送数据包，函数参数如下所示：

```
PCAP_AVAILABLE_0_8
PCAP_API int pcap_sendpacket(pcap_t *, const u_char *, int);
```

第一个参数是已经打开的句柄；第二个参数是要发送的数据包缓冲区；第三个参数是数据包大小。在使用这一函数时，需要获取设备列表，然后打开指定的网卡设备，之后要构造好数据包才可以调用该函数进行发送。数据包发送成功后便可以像上次实验一样捕获然后进行分析，最后就可以得到目标 MAC 地址。

2. ARP 协议

ARP（Address Resolution Protocol，地址解析协议）是一种用于在局域网中将 IP 地址解析为 MAC 地址的网络协议。当设备需要与同一局域网内的另一台设备通信时，它首先通过广播发送 ARP 请求包，询问"谁拥有这个 IP 地址"；所有设备都会收到请求，但只有目标设备会回复 ARP 响应包，告知自己的 MAC 地址。这样，源设备就能建立 IP 地址到 MAC 地址的映射并存入 ARP 缓存表，实现数据链路层的直接通信。ARP 协议

工作在 OSI 模型的网络层与数据链路层之间，是 TCP/IP 协议栈中实现局域网通信的基础协议。

ARP 报文头部结构如下：

字段名称	硬件类型	上层协议类型	MAC 地址长度	IP 地址长度	操作码	源 MAC 地址	源 IP 地址	目的 MAC 地址	目的 IP 地址
大小（字节）	2	2	1	1	2	6	4	6	4

在寻找目标 MAC 时，设备首先将 ARP 包中的目的 MAC 地址设置为 FF-FF-FF-FF-FF-FF，这样就涵盖了所有的设备，也就达到了广播的效果；而硬件类型应当为以太网，也就是 0x0001,；操作码应当为 0x0001，表示请求；协议类型应当为 0x0800，表示 IPv4 协议。

3. **路由器**

路由器是现代网络的核心枢纽，其核心功能是连接并智能管理不同网络之间的数据通信。具体表现为：通过 WAN 口接入互联网，通过 LAN 口和 Wi-Fi 创建本地局域网；内置 DHCP 服务器自动为接入设备分配 IP 地址；最关键的是，它能分析数据包的目的地址，并根据网络状况从可用路径中选择最优路径进行转发，从而高效、准确地引导数据在复杂网络环境中到达目的地。

三、实验过程

首先对实验环境进行配置，本次实验我使用助教提供的虚拟机实验环境。但是 Visual Studio2022 编译的程序无法在虚拟机 2 上使用，因此我使用自己的电脑代替了虚拟机 2，实现路由功能。因此我自定义了 vmnet2 网卡，设置为仅主机模式，并配置 IP 为 206.1.2.1 和 206.1.1.1，最后将另外三台虚拟机的网络适配器设置为 vmnet2 网卡即可，这样就实现了虚拟机 2 的功能。

下面开始正式讲解我的代码。在编程的过程中我定义了以太网帧结构体、ARP 报文结构体和一些函数，用来构造数据包并对捕获的数据包进行处理等。接下来我将使用注释的形式进行解释。

1. 以太网帧结构体

```
// 以太网帧头部
struct EthernetHeader
{
    byte dstMAC[6];          // 目标 MAC 地址
    byte srcMAC[6];          // 源 MAC 地址
    WORD ethertype;          // 协议类型
};
```

2. ARP 结构体

```
/ ARP 报文
struct ARP
{
    WORD hardware;           // 硬件类型
    WORD protocol;           // 协议类型
    byte MAC_length;         // MAC 地址长度，单位是字节
    byte IPaddress_length;   // 协议长度
    WORD opcode;             // 操作码
    byte srcMAC[6];          // 源 MAC 地址
    byte srcIP[4];           // 源 IP
    byte dstMAC[6];          // 目标 MAC 地址
    byte dstIP[4];           // 目标 IP
};
```

3. IP 数据包头部

```
// IP 报文头部
struct IPv4Header
{
    byte ver_ihl;      // 版本类型+头部长度
    byte TOS;          // 服务类型
    WORD Total_len;        // 总长度
    WORD ID;           // 标识
    WORD Flag_fragment; // 标志+片偏移
    byte TTL;          // 生存时间
    byte protocol;         // 协议类型
    WORD checksum;         // 头部校验和
    DWORD srcIP;       // 源 IP 地址
    DWORD dstIP;       // 目标 IP 地址
};
```

4. 路由表项

```
// 路由表项
struct RouteEntry
{
    DWORD network;      // 目的网络
    DWORD netmask;      // 子网掩码
    DWORD nextHop;      // 下一跳
};
```

5. 用于转发的数据包

```
// 用于转发的数据包
struct PacketForForward
{
    vector<byte> data;
    int length;
};
// 需要转发的数据包列表以及日志列表
queue<PacketForForward> g_pkt_queue;
queue<string> g_log_queue;
```

6. 全局变量

```
// 全局变量
pcap_t* handle;         // 网络设备句柄
byte localMAC[6];       // 网络设备 MAC
vector<RouteEntry> routing_table;   // 路由表
vector<DWORD> localIPs;             // 网络设备的 IP
```

```
unordered_map<DWORD, pair<array<byte, 6>,
chrono::steady_clock::time_point>> g_arp_cache;   // arp 缓存
const int ARP_CACHE_TIMEOUT_SEC = 300;  // ARP 缓存超时时间
mutex g_route_mutex, localIPs_mutex, arp_mutex, g_queue_mutex, g_log_mutex;
// 和锁有关的互斥量
condition_variable g_queue_cv;      // 条件变量用于同步线程
bool running = true;    // 路由器是否在运行
```

7. 将 DWORD 类型的 IP 地址转为字符串

```
string ip_to_str(DWORD ip_host)
{
    struct in_addr a;
    a.s_addr = htonl(ip_host);
    char buf[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &a, buf, sizeof(buf));
    return string(buf);
}
```

8. 日志记录和输出函数

```
// 日志记录
void safe_log(const string& msg)
{
    lock_guard<mutex> lk(g_log_mutex);
    g_log_queue.push(msg);
}
// 输出日志
void flush_logs()
{
    lock_guard<mutex> lk(g_log_mutex);
    while (!g_log_queue.empty())
    {
        cout << g_log_queue.front();
        g_log_queue.pop();
    }
    cout.flush();
}
```

由于我的代码中使用了多线程来实现路由器，因此需要设置互斥锁来确保写入正确。这里先将日志存入队列中，在使用 log 命令查看时再通过 flush_logs()函数统一输出，避免出现添加路由时输出日志信息的情况。

9. 打印 MAC 地址

```
// 打印 MAC 地址
void print_MAC(const byte MAC[6])
{
    for (int i = 0; i < 6; i++)
    {
        printf("%02X", MAC[i]);
        if (i < 5) cout << "-";
    }
    cout << endl;
}
```

10. 计算校验和

```
// 计算校验和
WORD ip_checksum(const void* vdata, size_t length)
{
    const uint8_t* data = (const uint8_t*)vdata;
    DWORD acc = 0;
    for (size_t i = 0; i + 1 < length; i += 2)
    {
        WORD word = (data[i] << 8) | data[i + 1];
        acc += word;
    }
    if (length & 1)
        acc += data[length - 1] << 8;
    while (acc >> 16)
        acc = (acc & 0xFFFF) + (acc >> 16);
    return (WORD)(~acc);
}
```

11. 寻找路由

```
// 寻找路由
RouteEntry* find_route(DWORD dst_host)
{
    // 设置线程锁，防止工作线程和捕获线程发生混乱
    lock_guard<mutex> lk(g_route_mutex);
    for (auto& r : routing_table)
        if ((dst_host & r.netmask) == (r.network & r.netmask))
            return &r;
    return nullptr;
}
```

12. 展示路由表

```
// 展示路由表
void display_routing_table()
{
    lock_guard<mutex> lk(g_route_mutex);
    cout << "\n====================================\n";
    cout << "Routing Table:\n";
    cout << "====================================\n";
    cout << "Index  Destination Network    Netmask            Next Hop\n";
    cout << "-----------------------------------------------------\n";
    for (size_t i = 0; i < routing_table.size(); ++i)
    {
        const auto& r = routing_table[i];
        string nextHopStr = (r.nextHop == 0) ? "direct" :
ip_to_str(r.nextHop);
        printf("%-5zu  %-20s  %-18s  %s\n",
               i,
               ip_to_str(r.network).c_str(),
               ip_to_str(r.netmask).c_str(),
               nextHopStr.c_str());
    }
    cout << "====================================\n\n";
}
```

13. 添加路由

```
// 添加路由
bool add_route(const string& net, const string& mask, const string& nh)
{
    RouteEntry e;
    e.network = ntohl(inet_addr(net.c_str()));
    e.netmask = ntohl(inet_addr(mask.c_str()));
    e.nextHop = (nh == "0" || nh == "0.0.0.0") ? 0 :
ntohl(inet_addr(nh.c_str()));
    // 设置线程锁防止发生混乱
    lock_guard<mutex> lk(g_route_mutex);
    routing_table.push_back(e);
    cout << "Route added: " << net << " / " << mask
         << " -> " << (e.nextHop == 0 ? "direct" : nh) << "\n";
    return true;
}
```

添加路由时，先设置目标网络、子网掩码和下一跳地址，如果下一跳地址是 0 或者 0.0.0.0，那么就代表直连，之后输出路由添加记录并将路由项放入路由表，完成了路由添加的操作。

14. 删除特定的路由表项

```
// 删除特定的路由表项
bool delete_route_by_index(size_t index)
{
    lock_guard<mutex> lk(g_route_mutex);
    if (index >= routing_table.size())
    {
        cout << "Invalid route index!\n";
        return false;
    }

    routing_table.erase(routing_table.begin() + index);
    return true;
}
```

该函数和下面的函数都是用于删除路由表项，但是这个函数可以用来删除特定的路由表项，下面函数直接将路由表清空。

15. 删除所有路由表项

```
// 删除所有路由表项
bool delete_route(const string& net, const string& mask)
{
    DWORD network = ntohl(inet_addr(net.c_str()));
    DWORD netmask = ntohl(inet_addr(mask.c_str()));

    lock_guard<mutex> lk(g_route_mutex);
    for (size_t i = 0; i < routing_table.size(); ++i)
    {
        if (routing_table[i].network == network && routing_table[i].netmask
== netmask)
        {
            routing_table.erase(routing_table.begin() + i);
            return true;
        }
    }
    return false;
}
```

16. 构造 ARP 数据包并发送请求

```
// 构造 ARP 数据包并发送请求
bool send_arp_request(DWORD target_ip_host)
{
    if (!handle) return false;
    byte pkt[42];
    EthernetHeader* eth = (EthernetHeader*)pkt;
    memset(eth->dstMAC, 0xFF, 6);
    memcpy(eth->srcMAC, localMAC, 6);
    eth->ethertype = htons(0x0806);
    ARP* arp = (ARP*)(pkt + sizeof(EthernetHeader));
    arp->hardware = htons(1);
    arp->protocol = htons(0x0800);
    arp->MAC_length = 6;
    arp->IPaddress_length = 4;
    arp->opcode = htons(1);
    memcpy(arp->srcMAC, localMAC, 6);
    arp->srcIP[0] = arp->srcIP[1] = arp->srcIP[2] = arp->srcIP[3] = 0;
    memset(arp->dstMAC, 0, 6);
    DWORD t = htonl(target_ip_host);
    memcpy(arp->dstIP, &t, 4);
    return pcap_sendpacket(handle, pkt, sizeof(pkt)) == 0;
}
```

这部分实现了构造并发送 ARP 包，用来获取目标 IP 的 MAC 地址，由于和之前实验的代码基本相同，此处就不再赘述。

17. 清理过期的 ARP 缓存

```
// 清理过期的 ARP 缓存条目
void cleanup_expired_arp_cache()
{
    auto now = chrono::steady_clock::now();
    auto timeout = chrono::seconds(ARP_CACHE_TIMEOUT_SEC);
    for (auto it = g_arp_cache.begin(); it != g_arp_cache.end();)
    {
        if (now - it->second.second > timeout)
            it = g_arp_cache.erase(it);
        else
            ++it;
    }
}
```

这部分实现了清除 ARP 缓存，缓存时长为 5 秒，确保了 MAC 地址的正确性，防止恶意 ARP 响应或者错误 MAC 地址造成的影响。

18. 将 IP 地址解析为 MAC 地址

```
// 将 IP 地址解析为 MAC 地址
bool resolve_mac(DWORD ip_host, array<byte, 6>& out_mac, int timeout_ms = 1000)
{
     // 在 ARP 缓存中寻找是否有当前 IP 地址的 MAC 地址
    {
        lock_guard<mutex> lk(arp_mutex);
        cleanup_expired_arp_cache();  // 清理过期条目
        auto it = g_arp_cache.find(ip_host);
        if (it != g_arp_cache.end())
        {
            out_mac = it->second.first;
            return true;
        }
    }
    // 没有则发送 ARP 请求
    if (!send_arp_request(ip_host))
        return false;  // ARP 请求发送失败，直接返回
    // 等待 ARP 响应
    for (int waited = 0; waited < timeout_ms; waited += 50)
    {
        // 每 50ms 检查一次缓存，capture_thread 线程在后台工作，更新缓存
        this_thread::sleep_for(chrono::milliseconds(50));
        lock_guard<mutex> lk(arp_mutex);
        auto it = g_arp_cache.find(ip_host);
        // 超时则返回 false
        if (it != g_arp_cache.end())
        {
            out_mac = it->second.first;
            return true;
        }
    }
    return false;
}
```

这部分实现了通过 IP 地址寻找对应的 MAC 地址。首先在 ARP 缓存中查找是否有对应的 MAC 地址，如果没有，就发送 ARP 数据包并等待响应。该函数的原理是：当转发数据包线程收到需要转发的数据包时，调用该函数解析下一跳的 MAC 地址。若未查询到缓存记录则发送 ARP 请求，后

台运行的捕获数据包线程会将 ARP 响应存入 ARP 缓存中，这样只要定时查看缓存即可。

19. 捕获数据包线程

```
// 捕获数据包线程
void capture_thread()
{
    struct pcap_pkthdr* header;
    const u_char* pkt_data;
    int res;
    // 工作主循环
    while (running)
    {
        // 捕获数据包
        res = pcap_next_ex(handle, &header, &pkt_data);
        if (res == 0)
            continue; // timeout
        if (res < 0)
            break;
        // 解析数据包的以太网帧头部
        const EthernetHeader* eth = (const EthernetHeader*)pkt_data;
        WORD etype = ntohs(eth->ethertype);
        // 属于 ARP 协议
        if (etype == 0x0806)
        {
            // 解析 ARP 数据包
            const ARP* arp = (const ARP*)(pkt_data + sizeof(EthernetHeader));
            if (ntohs(arp->opcode) == 2)
            {
                // 将 MAC 地址存入 ARP 缓存
                DWORD sip_host = ntohl(*(DWORD*)arp->srcIP);
                array<byte, 6> mac;
                memcpy(mac.data(), arp->srcMAC, 6);
                lock_guard<mutex> lk(arp_mutex);
                cleanup_expired_arp_cache();  // 添加新条目时清理过期条目
                g_arp_cache[sip_host] = make_pair(mac,
chrono::steady_clock::now());
            }
        }
        // 如果是 IPv4 数据包
        else if (etype == 0x0800)
        {
```

```
            if (header->caplen < (int)(sizeof(EthernetHeader) +
sizeof(IPv4Header))) continue;
            // 解析 IP 头部
            const IPv4Header* ip = (const IPv4Header*)(pkt_data +
sizeof(EthernetHeader));
            DWORD dst_host = ntohl(ip->dstIP);
            DWORD src_host = ntohl(ip->srcIP);
            // 检查目的 MAC 是否为已打开设备的 MAC 地址，不是则跳过，不需要
转发
            bool match = true;
            for (int i = 0; i < 6; i++)
            {
                if (localMAC[i] != eth->dstMAC[i])
                {
                    match = false;
                    break;
                }
            }
            if (!match) continue;
            // 检查目的 IP 是否为已打开设备的 IP 地址，如果是则跳过，不需要
转发
            {
                lock_guard<mutex> lk(localIPs_mutex);
                bool is_local = false;
                for (DWORD localIP : localIPs)
                {
                    if (dst_host == localIP)
                    {
                        is_local = true;
                        break;
                    }
                }
                if (is_local) continue;
            }
            // 构造转发数据包
            PacketForForward p;
            p.length = header->caplen;
            p.data.resize(p.length);
            memcpy(p.data.data(), pkt_data, p.length);
            // 放入待转发队列并通知 worker 线程
            {
                lock_guard<mutex> lk(g_queue_mutex);
                g_pkt_queue.push(move(p));
```

```
            }
            g_queue_cv.notify_one();
        }
    }
}
```

这部分是代码的核心之一，实现了捕获数据包并进行处理。首先捕获数据包并解析以太网帧的头部信息，根据以太网帧类型判断数据包属于 ARP 还是 IPv4。

如果属于 ARP 数据包，那么就使用 ARP 结构体指针解析，将其中的 MAC 地址存入 ARP 缓存中，便于后续查询

如果属于 IPv4 数据包，那么需要进行一系列检查，确保是路由器要转发的包。先解析 IPv4 数据包头，检查目的 MAC 地址是否为该路由设备的 MAC 地址，如果不是则不需要转发；接着检查目的 IP 是否为该设备的 IP，如果是就说明已经到达，不需要再转发。这里需要使用块语句将其包裹，防止互斥锁出现重定义的情况。

经过上面排查后剩下的包就是需要进行转发的，这些数据包的目的 MAC 是该路由设备，但是目的 IP 不是，因此需要经过转发才能到达目的地。先构造转发数据包，然后放入待转发队列并通知转发数据包线程。这一线程运行在后台，不断捕获 ARP 响应和 IPv4 的数据包并进行处理，为转发数据包线程提供帮助。

20. 转发数据包线程

```
// 转发数据包线程
void worker_thread()
{
    // 工作主循环
    while (running)
    {
        PacketForForward p;
        // 等待队列中的数据包
        {
            unique_lock<mutex> lk(g_queue_mutex);
            g_queue_cv.wait(lk, [] { return !g_pkt_queue.empty()
|| !running; });
            if (!running && g_pkt_queue.empty()) break;
            p = move(g_pkt_queue.front());
```

```
            g_pkt_queue.pop();
        }
        if (p.length < (int)(sizeof(EthernetHeader) + sizeof(IPv4Header)))
continue;
        // 解析数据包头部
        EthernetHeader* eth = (EthernetHeader*)p.data.data();
        IPv4Header* ip = (IPv4Header*)(p.data.data() +
sizeof(EthernetHeader));
        DWORD dst_host = ntohl(ip->dstIP);
        DWORD src_host = ntohl(ip->srcIP);
        // 寻找路由
        RouteEntry* route = find_route(dst_host);
        // 找不到路由信息，输出错误
        if (!route)
        {
            stringstream ss;
            ss << "[DROP] No route found for " << ip_to_str(dst_host) << "\n";
            safe_log(ss.str());
            continue;
        }
        // 确定下一跳 IP 地址（直接连接则用目标 IP，否则用路由表中的下一跳）
        DWORD nextHopHost = (route->nextHop == 0) ? dst_host : route->nextHop;
        array<byte, 6> nexthostmac;
        // 解析下一跳的 MAC 地址
        if (!resolve_mac(nextHopHost, nexthostmac, 1000))
        {
            stringstream ss;
            ss << "[DROP] Failed to resolve MAC for next hop " <<
ip_to_str(nextHopHost) << "\n";
            safe_log(ss.str());
            continue;
        }
        // 修改数据包头部
        vector<byte> outbuf = move(p.data);
        EthernetHeader* oeth = (EthernetHeader*)outbuf.data();
        memcpy(oeth->srcMAC, localMAC, 6);             // 源 MAC 改为路由器
MAC
        memcpy(oeth->dstMAC, nexthostmac.data(), 6);  // 目的 MAC 改为下一跳
MAC

        // 检查并更新 TTL
        byte original_ttl = ip->TTL;
        if (ip->TTL <= 1)
```

```
        {
            stringstream ss;
            ss << "[DROP] TTL expired for packet " << ip_to_str(src_host)
               << " -> " << ip_to_str(dst_host) << " (TTL=" << (int)ip->TTL
<< ")\n";
            safe_log(ss.str());
            continue;
        }
        // TTL 减 1
        ip->TTL--;
        // 重新计算 IP 头部校验和
        ip->checksum = 0;
        size_t ip_hdr_len = (ip->ver_ihl & 0x0F) * 4;
        ip->checksum = htons(ip_checksum((const void*)ip, ip_hdr_len));

        // 发送数据包
        int r = pcap_sendpacket(handle, outbuf.data(), (int)outbuf.size());
        if (r != 0)
        {
            stringstream ss;
            ss << "[SEND ERROR] pcap_sendpacket failed for " <<
ip_to_str(src_host)
               << " -> " << ip_to_str(dst_host) << "\n";
            safe_log(ss.str());
        }
        else
        {
            string nextHopStr = (route->nextHop == 0) ? "direct" :
ip_to_str(nextHopHost);
            stringstream ss;
            ss << "[FORWARD] " << ip_to_str(src_host) << " -> " <<
ip_to_str(dst_host)
               << " via " << nextHopStr << " (TTL: " << (int)original_ttl
<< " -> " << (int)ip->TTL << ")\n";
            safe_log(ss.str());
        }
    }
}
```

这部分实现了路由转发功能，是路由程序的核心。首先等待队列中的数据包，当后台的捕获数据包线程将需要转发的数据包放入待转发队列后，开始工作。首先解析数据包头部并在路由表中查找路由，如果找不到则无法转发；如果找到了，则修改数据包头部，将源 MAC 地址改为

路由器的 MAC 地址，目的 MAC 改为下一跳的 MAC 地址，将 TTL 值减一。之后重新计算校验和并组装数据包，然后发送数据包，这样就实现了转发功能。每经过一个路由器，TTL 就要减一，因此在本次实验中，收到的数据包 TTL 值都应当为 126。

21. 让用户选择需要打开的网卡设备

```
// 让用户选择需要打开的网卡设备
static pcap_if_t* select_device(pcap_if_t* alldevices, int count)
{
    pcap_if_t* d;               // 遍历指针
    int op;                         // 设备编号
    cout << "Please enter the number of the Network interface Device you want
to open:";
    cin >> op;

LOOP:
    if (op < 1 || op > count)           // 输入的数字无效
    {
        cout << "Invalid number!Out of range!" << endl;
        cout << "continue or quit ([c/q]):";// 选择继续或者退出
        char op;
        cin >> op;
        if (op == 'c' || op == 'C')
            goto LOOP;
        else
        {
            pcap_freealldevs(alldevices);   // 释放设备列表
            exit(0);
        }
    }
    // 寻找指定的设备
    int i;
    for (d = alldevices, i = 0;i < op - 1 && d != nullptr;d = d->next, i++);

    return d;
}
```

22. 获取打开的网卡设备的 MAC

```
// 获取打开的网卡设备的 MAC
static void Get_local_MAC(const string device_name, byte mac[6])
{
    IP_ADAPTER_INFO adapterinfo[16];        // 最多 16 个适配器
```

```
    DWORD dwBufLen = sizeof(adapterinfo);    // 缓冲区大小
    // 调用系统 API 获取网络适配器信息
    DWORD dwStatus = GetAdaptersInfo(adapterinfo, &dwBufLen);
    if (dwStatus != ERROR_SUCCESS)
    {
        std::cerr << "GetAdaptersInfo failed!" << std::endl;
        return;
    }
    // 提取设备名称中的 GUID，也就是大括号包裹的部分
    string guid;
    size_t pos = device_name.find("{");
    if (pos != string::npos)
        guid = device_name.substr(pos);
    // 根据 GUID 匹配并找到 MAC 地址
    for (PIP_ADAPTER_INFO p = adapterinfo;p != nullptr;p = p->Next)
    {
        if (guid.find(p->AdapterName) != string::npos)
        {
            memcpy(mac, p->Address, 6);
            return;
        }
    }
}
```

23. main 函数

```
int main()
{
    pcap_if_t* alldevs;             // 设备列表的存储链表
    pcap_if_t* device;              // 遍历指针
    char errbuf[PCAP_ERRBUF_SIZE]; // 存储错误信息，256 字节
    cout << "================================Devices
List================================" << endl;
    // 从本地设备中获取可用设备列表
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) ==
-1)
    {
        // 查找失败，则输出错误信息
        cerr << "pcap_findalldevs_ex failed: " << errbuf << "\n";
        return -1;
    }
    // 显示当前存在的设备信息
    int idx = 0;
    for (pcap_if_t* d = alldevs; d; d = d->next, idx++)
```

```
        cout << (idx+1) << ". " << (d->name ? d->name : "") << " - " <<
(d->description ? d->description : "") << "\n";
    cout << "Total devices:" << idx << endl;
    cout << endl;
    // 选择要打开的设备
    device = select_device(alldevs, idx);
    if (!device)
        return -1;
    // 打开设备
    handle = pcap_open(device->name, 65536, PCAP_OPENFLAG_PROMISCUOUS, 100,
NULL, errbuf);
    if (!handle)
    {
        cerr << "pcap_open failed: " << errbuf << "\n";
        pcap_freealldevs(alldevs);
        return -1;
    }
    cout << "Successfully open network device:" << (device->description ?
device->description : device->name) << "\n";
    // 获取设备 MAC 地址
    Get_local_MAC(device->name, localMAC);
    cout << "Local MAC: ";
    print_MAC(localMAC);
    // 获取网络设备 IP
    {
        lock_guard<mutex> lk(localIPs_mutex);
        for (pcap_addr_t* a = device->addresses; a; a = a->next)
        {
            if (a->addr && a->addr->sa_family == AF_INET)
            {
                DWORD ip_host =
ntohl(((sockaddr_in*)a->addr)->sin_addr.s_addr);
                localIPs.push_back(ip_host);
                cout << "Local IP: " << ip_to_str(ip_host) << "\n";
            }
        }
    }
    // 路由器开始工作
    cout << "\n=== Router Initialization ===\n";
    cout << "Type 'help' for commands.\n\n";
    thread capt(capture_thread);
    thread worker(worker_thread);
    cout << "Router running. Type 'help' for commands.\n";
```

```
    string line;
    cin.ignore();  // 清除之前 cin>>留下的换行符
    while (running)
    {
        cout << "router> ";
        cout.flush();
        getline(cin, line);

        if (line.empty()) continue;

        stringstream ss(line);
        string cmd;
        ss >> cmd;
        // 根据命令进入不同分支，help 显示当前可用命令，add 添加路由表项，
del 删除全部路由表项
        // deli 删除指定路由表项，show 显示路由表，log 显示当前日志，quit/exit
退出并停止路由器
        if (cmd == "help" || cmd == "h")
        {
            cout << "\nAvailable commands:\n";
            cout << "  add <network> <netmask> <next_hop>  - Add a route (use
0 or 0.0.0.0 for direct)\n";
            cout << "  del <network> <netmask>             - Delete a route
by network and mask\n";
            cout << "  deli <index>                        - Delete a route
by index (see 'show')\n";
            cout << "  show                                - Display
routing table\n";
            cout << "  log                                 - Display
pending logs\n";
            cout << "  quit / exit                         - Stop router and
exit\n";
            cout << "  help                                - Show this help
message\n\n";
        }
        // 添加路由项
        else if (cmd == "add" || cmd == "a")
        {
            string net, mask, nh;
            if (ss >> net >> mask >> nh)
                add_route(net, mask, nh);
            else
                cout << "Usage: add <network> <netmask> <next_hop>\n";
        }
```

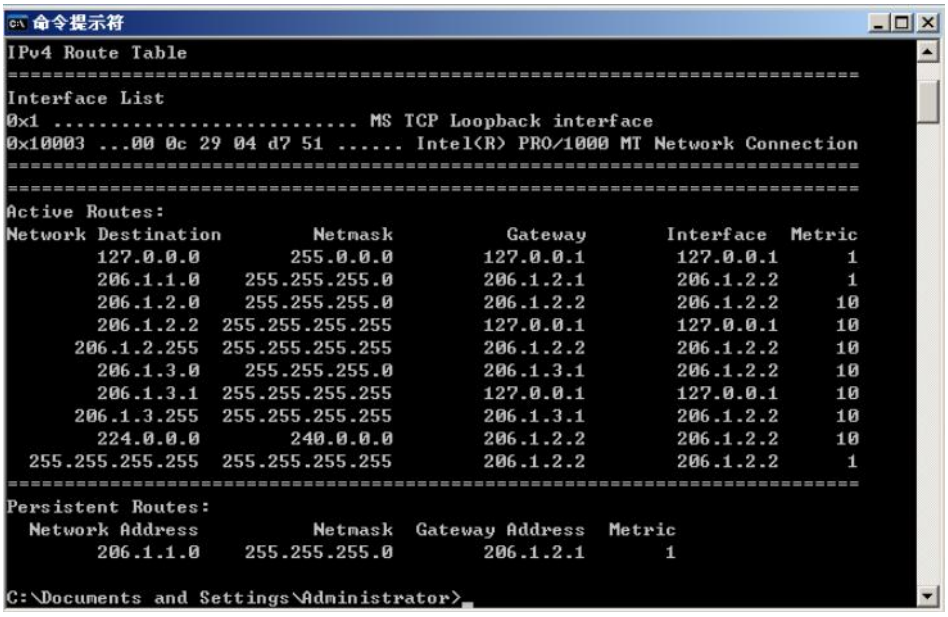
```
        // 删除全部路由表项
        else if (cmd == "del" || cmd == "d")
        {
            string net, mask;
            if (ss >> net >> mask)
                delete_route(net, mask);
            else
                cout << "Usage: del <network> <netmask>\n";
        }
        // 删除指定路由表项
        else if (cmd == "deli")
        {
            size_t index;
            if (ss >> index)
                delete_route_by_index(index);
            else
                cout << "Usage: deli <index>\n";
        }
        // 显示路由表
        else if (cmd == "show" || cmd == "s")
            display_routing_table();
        // 显示日志记录
        else if (cmd == "log" || cmd == "logs" || cmd == "l")
            flush_logs();
        // 退出
        else if (cmd == "quit" || cmd == "exit" || cmd == "q")
        {
            cout << "Stopping router...\n";
            running = false;
            g_queue_cv.notify_all();
            break;
        }
        else
            cout << "Unknown command: " << cmd << ". Type 'help' for available
commands.\n";
    }
    // 结束运行，释放设备并清理资源
    pcap_breakloop(handle);
    capt.join();
    worker.join();
    pcap_close(handle);
    pcap_freealldevs(alldevs);
    cout << "Stopped.\n";
    return 0;
}
```

上面是 main 函数的代码实现，其中包含了许多提示性文字以及各类命令的分支处理，因此显得有些冗长。

在 main 函数中，首先获取可用设备列表，让用户自己选择想要打开的设备并尝试打开，之后自动获得该设备的 MAC 地址和 IP 地址，这样就完成了初始化工作。

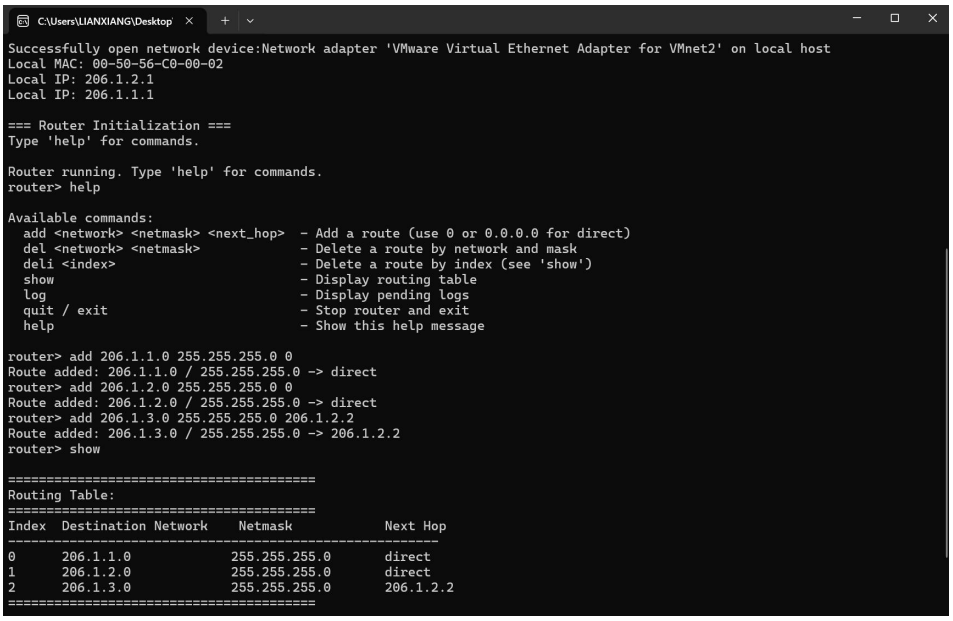
接下来路由器开始工作，捕获数据包线程和转发数据包线程被创建。用户可以输入命令执行添加路由、删除路由、显示路由表、显示日志等操作，当输入 quit/exit 命令时，退出工作主循环，路由器停止工作。最后中断捕获函数，两个线程在执行完毕后结束，再释放设备列表、清理网络资源，这样可以有效防止线程尚未执行完毕而资源被清理的情况出现。

以上就是对所有代码的讲解和注释。在完成路由器程序后，我打开虚拟机 1,3,4，它们都连接 vmnet2 网卡。按照实验手册配置了虚拟机 3 的路由，如下所示：

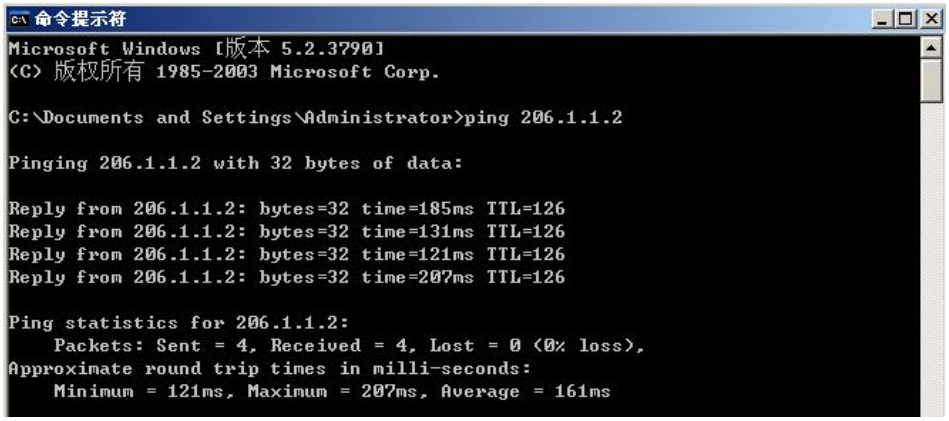


启动路由器程序，首先添加路由，206.1.1.0 和 206.1.2.0 这两个目的网络都应该设置为直达（下一跳为 0.0.0.0），因为 206.1.1.1 是主机 1 的网关，在 vmnet2 网卡中已经配置，这是连接主机 1 的左接口；而 206.1.2.1 是连接右边路由器的右接口，和虚拟机 3 的 206.1.2.2 在同

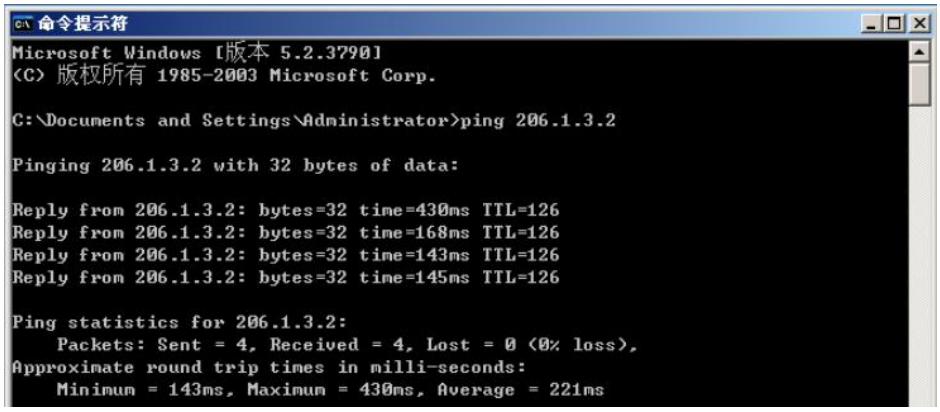
一网络中，因此无需经过转发。然后添加到达 206.1.3.0 网络的路由，下一跳设置为 206.1.2.2。配置如下所示：



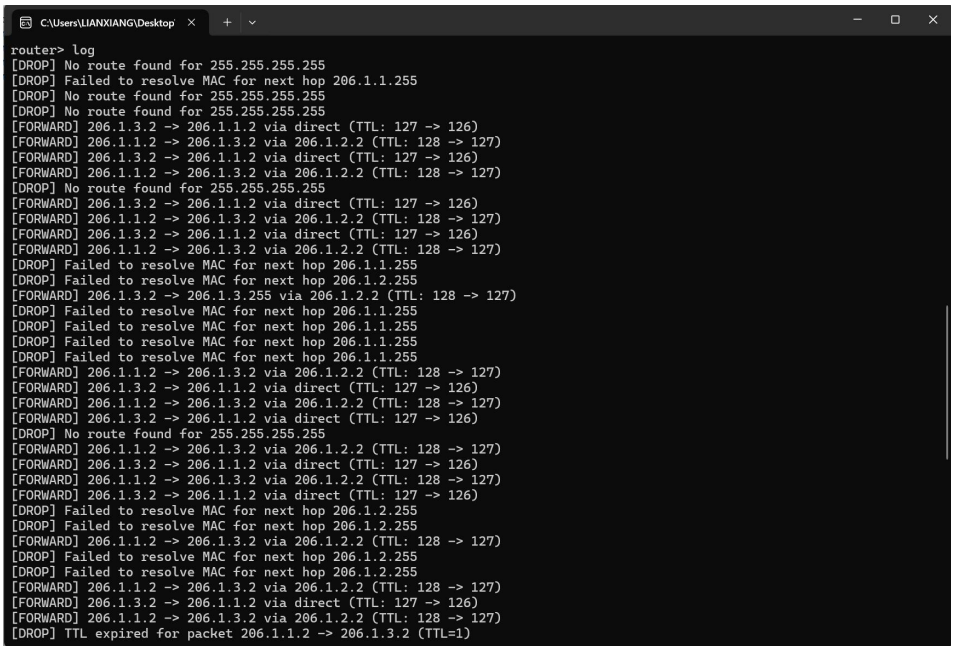
下面在虚拟机上进行测试。在主机 4 上 ping 主机 1 能够连通，并且 TTL 为 126，说明经过了两跳，如下所示：



主机 1ping 主机 4 也能够连通，TTL 为 126，说明程序没有问题！



程序运行时的日志记录如下所示：



由于我首先使用主机 4ping 主机 1，因此第一个 FORWARD 是从 206.1.3.2 到 206.1.1.2 的，并且这个数据包经历了虚拟机 3 的转发，所以 TTL 是 127，再经过路由程序的转发，变成了 126，而主机 1ping 主机 4 的日志记录也是类似的，这就验证了程序的正确性。

在实验过程中出现了主机 1ping 主机收到的数据包 TTL 为 127，而主机 4ping 主机 1 收到的数据包 TTL 为 126 的情况，经过排查后发现是在捕获 IPv4 数据包时未设置过滤器，导致路由程序对所有的数据包都进行了转发，这是错误的。因此在增加过滤器后，就解决了这个问题。

四、实验结论及心得体会

本次实验我实现了一个简单的 IPv4 路由器，包含添加路由、删除路由、转发路由、日志记录等功能。在实验中我出现了相同路径互相 ping 两边 TTL 不对称的问题，经过排查后解决了该问题。这次实验让我对路由转发机制有了更加深刻的理解，对 TTL 的作用也有了进一步的认知，巩固了课上的知识。