重庆邮电大学

学生实验实习报告册

学年学期：2025-2026 学年（1）学期

课程名称：通信软件开发与应用

学生学院：通信与信息工程学院

专业班级：01042301

学生学号：2023211281

学生姓名：丁同勖

联系电话：18019582857

重庆邮电大学教务处制

课程名称	通信软件开发与应用			**课程编号**	A2012230		
实验地点	YF315			**实验时间**	12 周 9-11 节		
校外指导教师	无			**校内指导教师**	梁燕		
实验名称	路由追踪程序设计与实现						
评阅人签字		**操作成绩 80%**		**报告成绩 20%**		**总评成绩**	

一、实验目的

1、掌握路由追踪的基本原理
2、掌握协议数据包构造方法
3、掌握协议数据包解析方法
4、掌握 ICMP 协议结构
5、掌握指针强制类型转换在协议构造与解析中的应用方法

二、实验思路

该程序的逻辑完全遵循标准的 Traceroute 原理，主要分为四个阶段：

第一阶段：初始化与准备

1. **参数校验**：程序启动时检查命令行参数。如果未提供目标 IP 或域名，打印使用说明并退出。
2. **Winsock 初始化**：加载 Windows Socket 库 (WSAStartup)。
3. **地址解析**：
   - 尝试将输入识别为 IP 地址。
   - 如果不是 IP，则使用 DNS (gethostbyname) 将域名解析为 IP 地址。如果解析失败，程序终止。
4. **创建套接字**：创建一个 **原始套接字**，协议类型为 ICMP。

第二阶段：主循环 (Traceroute 核心)

程序进入一个循环，TTL（生存时间）从 1 开始递增，最大值为 30 (MAX_HOPS)。

1. **设置 TTL**：调用 setsockopt 修改 IP 头部的 TTL 字段。
   - TTL=1 时，第一个路由器会丢包并返回 ICMP 超时。
   - TTL=2 时，第二个路由器会丢包，依此类推。
2. **构造数据包**：填充 ICMP 头部（类型为 ECHO_REQUEST），填入序列号和进程 ID（作为标识符），并计算校验和。
3. **发送请求**：记录发送时间，使用 sendto 发送数据包。
4. **等待响应**：
   - 使用 select 模型等待 Socket 可读，设置超时时间为 3 秒。
   - 如果超时，输出 * 表示该跳无响应。

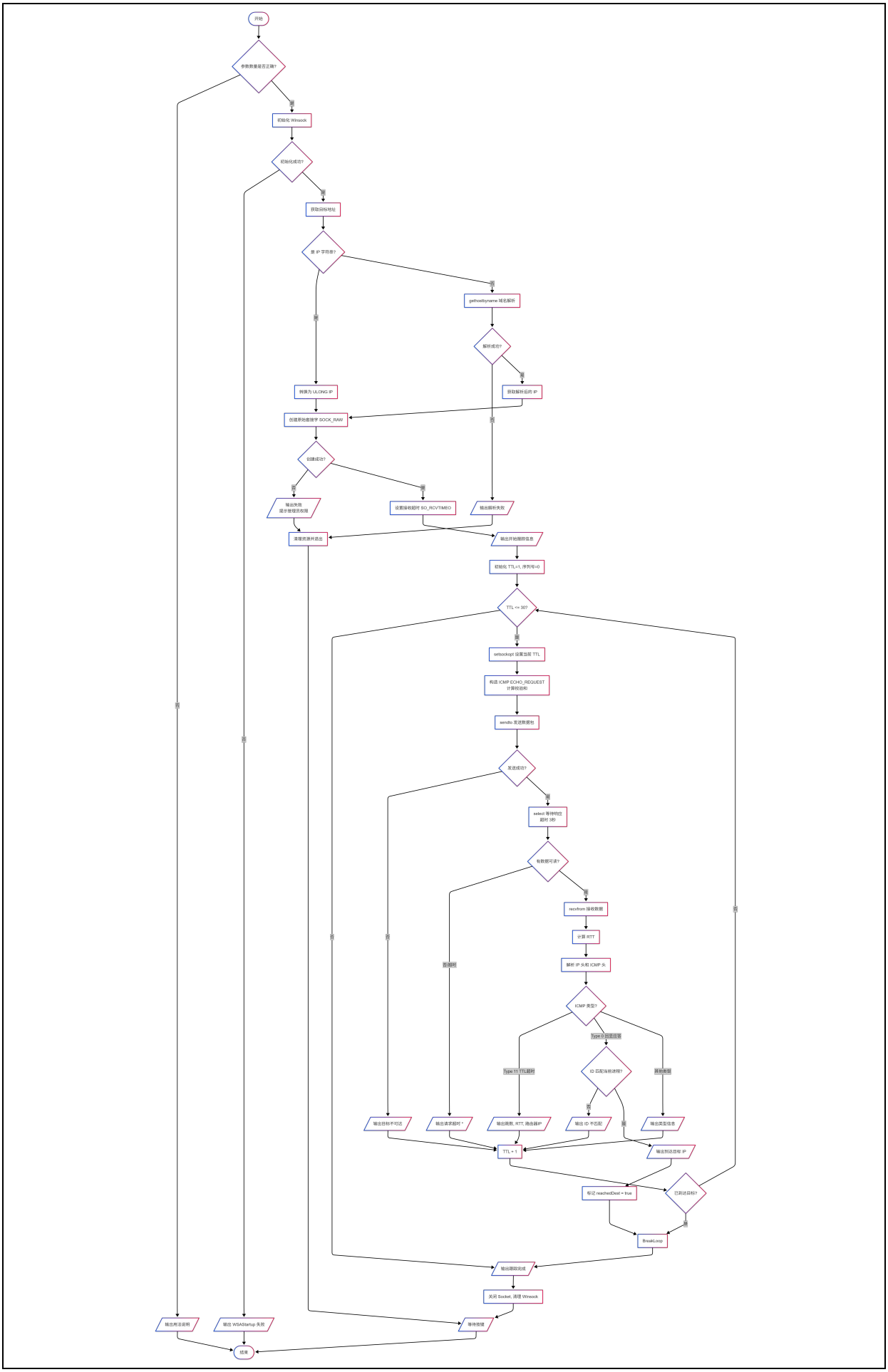
第三阶段：数据包解析与逻辑判断

如果收到了数据包（recvfrom），程序会解析 IP 头和 ICMP 头：

1. **计算往返时间 (RTT)**：接收时间减去发送时间。
2. **判断 ICMP 类型**：
   - **Type 11 (Time Exceeded / 超时)**：表示数据包在传输途中 TTL 耗尽。这代表了路径上的一个中间节点（路由器）。程序输出该节点的 IP。
   - **Type 0 (Echo Reply / 回显应答)**：表示数据包成功到达了目标主机并返回。
     - 程序会检查 ICMP 头部中的 ID 是否与本进程 ID 一致。
     - 如果一致，说明成功到达终点，输出信息并将 reachedDest 标记为真。
   - **其他类型**：如 Type 3 (目标不可达)，程序会照常输出 IP 和类型代码。

第四阶段：循环结束与清理

1. **循环终止条件：**
    - TTL 达到最大值 (30)。
    - 或者变量 reachedDest 为真（已到达目的地）。
2. **资源清理：**打印结束信息，关闭套接字 (closesocket)，清理 Winsock (WSACleanup)。
3. **退出：**等待用户按键后退出程序。


开始
参数数量是否正确?
否
是
初始化 Winsock
初始化成功?
否
是
获取目标地址
是 IP 字符串?
是
否
gethostbyname 域名解析
解析成功?
否
是
获取解析后的 IP
转换为 ULONG IP
创建原始套接字 SOCK_RAW
创建成功?
否
是
输出失败
提示管理员权限
设置接收超时 SO_RCVTIMEO
输出解析失败
清理资源并退出
输出开始跟踪信息
初始化 TTL=1, 序列号=0
TTL <= 30?
是
否
setsockopt 设置当前 TTL
构造 ICMP ECHO_REQUEST
计算校验和
sendto 发送数据包
发送成功?
是
否
select 等待响应
超时 3秒
有数据可读?
是
否
recvfrom 接收数据
计算 RTT
解析 IP 头和 ICMP 头
ICMP 类型?
Type 0 回显应答
Type 11 TTL超时
ID 匹配当前进程?
其他类型
是
否
输出目标不可达
输出请求超时 *
输出跳数、RTT、路由器IP
输出 ID 不匹配
输出类型信息
TTL + 1
输出到达目标 IP
标记 reachedDest = true
已到达目标?
是
否
BreakLoop
输出跟踪完成
关闭 Socket, 清理 Winsock
等待按键
输出用法说明
输出 WSAStartup 失败
结束

三、源代码

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS // 允许使用 inet_addr, gethostbyname 等旧函数

#include <iostream>
#include <winsock2.h>
#include <ws2tcpip.h>
//#include <iomanip>

// 链接 Winsock 库
#pragma comment(lib, "ws2_32.lib")

using namespace std;

// ========================================================================
// 常量定义
// ========================================================================
#define ICMP_ECHO_REQUEST   8
#define ICMP_ECHO_REPLY     0
#define ICMP_TIMEOUT        11
#define DEF_ICMP_DATA_SIZE  32
#define MAX_HOPS            30
#define DEF_TIMEOUT         3000 // 默认超时时间 3000ms (3秒)

// ========================================================================
// 数据结构定义
// ========================================================================

// IP 报头结构
typedef struct {
    unsigned char  h_len : 4;        // 首部长度
    unsigned char  version : 4;      // 版本
    unsigned char  tos;              // 服务类型
    unsigned short total_len;        // 总长度
    unsigned short ident;            // 标识
    unsigned short frag_and_flags;   // 标志与片偏移
    unsigned char  ttl;              // 生存时间
    unsigned char  proto;            // 协议
    unsigned short checksum;         // 校验和
    unsigned int   sourceIP;         // 源IP地址
    unsigned int   destIP;           // 目的IP地址
} IpHeader;

// ICMP 报头结构
typedef struct {
    unsigned char  i_type;           // 类型
    unsigned char  i_code;           // 代码
```

```
    unsigned short i_cksum;          // 校验和
    unsigned short i_id;             // 标识符
    unsigned short i_seq;            // 序列号
    unsigned int   timestamp;        // 数据部分：简单的时间戳
} IcmpHeader;

// ============================================================================
// 辅助函数
// ============================================================================

/**
 * 计算校验和
 * @param buffer 数据缓冲区
 * @param size 数据大小
 * @return 计算出的校验和
 */
unsigned short checksum(unsigned short* buffer, int size) {
    unsigned long cksum = 0;
    while (size > 1) {
        cksum += *buffer++;
        size -= sizeof(unsigned short);
    }
    if (size) {
        cksum += *(unsigned char*)buffer;
    }
    // 将32位累加和折叠成16位
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >> 16);
    return (unsigned short)(~cksum);
}

// ============================================================================
// 主程序
// ============================================================================

int main(int argc, char* argv[]) {
    // 0. 参数校验
    if (argc != 2) {
        cout << "Usage: itracert.exe ip_or_hostname" << endl;
        return 1;
    }

    // 1. 初始化 Winsock
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cerr << "WSAStartup failed." << endl;
```

```
        return 1;
    }

    // 2. 解析目的地址
    char* destStr = argv[1];
    unsigned long destIp = inet_addr(destStr);
    struct hostent* remoteHost;

    // 如果输入的是域名而非 IP，则进行域名解析
    if (destIp == INADDR_NONE) {
        remoteHost = gethostbyname(destStr);
        if (remoteHost == NULL) {
            cerr << "无法解析主机名: " << destStr << endl;
            WSACleanup();
            return 1;
        }
        destIp = *(unsigned long*)remoteHost->h_addr_list[0];
    }

    // 转换 IP 为字符串形式以便显示
    struct in_addr destAddrStruct;
    destAddrStruct.s_addr = destIp;
    char* destIpStr = inet_ntoa(destAddrStruct);

    // 3. 创建原始套接字 (Raw Socket)
    // 注意：创建原始套接字通常需要管理员权限
    SOCKET sockRaw = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockRaw == INVALID_SOCKET) {
        cerr << "无法创建套接字。请确认是否以【管理员权限】运行程序。" << endl;
        cerr << "Error Code: " << WSAGetLastError() << endl;
        WSACleanup();
        return 1;
    }

    // 设置 Socket 接收超时
    int timeout = DEF_TIMEOUT;
    setsockopt(sockRaw, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout, sizeof(timeout));

    // 4. 准备目的地址结构
    sockaddr_in destSockAddr;
    memset(&destSockAddr, 0, sizeof(destSockAddr));
    destSockAddr.sin_family = AF_INET;
    destSockAddr.sin_addr.s_addr = destIp;

    // 5. 输出头部信息
    cout << "==== 开始跟踪“" << destStr << "”（最大 " << MAX_HOPS << " 跳）====" << endl;
```

```
    cout << "跳数 \t往返时间 (ms) \t节点IP地址" << endl;

    // 6. 主循环：TTL 从 1 到 MAX_HOPS
    bool reachedDest = false;
    USHORT seq_no = 0;
    int processId = GetCurrentProcessId(); // 使用进程ID作为 ICMP ID

    for (int ttl = 1; ttl <= MAX_HOPS; ++ttl) {
        // 6.1 设置 IP 头部的 TTL 字段
        if (setsockopt(sockRaw, IPPROTO_IP, IP_TTL, (const char*)&ttl, sizeof(ttl)) ==
SOCKET_ERROR) {
            cerr << "Set TTL failed." << endl;
            break;
        }

        // 6.2 构造 ICMP 报文
        char icmp_data[sizeof(IcmpHeader) + DEF_ICMP_DATA_SIZE];
        IcmpHeader* icmp_hdr = (IcmpHeader*)icmp_data;
        memset(icmp_data, 0, sizeof(icmp_data));

        icmp_hdr->i_type = ICMP_ECHO_REQUEST;
        icmp_hdr->i_code = 0;
        icmp_hdr->i_id = (unsigned short)processId;
        icmp_hdr->i_seq = ++seq_no;
        icmp_hdr->i_cksum = 0;
        // 计算校验和必须在填充完所有数据后进行
        icmp_hdr->i_cksum = checksum((unsigned short*)icmp_data, sizeof(icmp_data));

        // 记录发送时间
        DWORD startTime = GetTickCount();

        // 6.3 发送 ICMP 请求
        int iResult = sendto(sockRaw, icmp_data, sizeof(icmp_data), 0, (sockaddr*)&destSockAddr,
sizeof(destSockAddr));
        if (iResult == SOCKET_ERROR) {
            cout << ttl << "\t*\t\t目标不可达(Send Fail)" << endl;
            continue;
        }

        // 6.4 接收响应
        sockaddr_in fromAddr;
        int fromLen = sizeof(fromAddr);
        char recvBuf[1024];

        // 使用 select 模型处理超时（比 setsockopt 更灵活准确）
        fd_set readfds;
```

```
        FD_ZERO(&readfds);
        FD_SET(sockRaw, &readfds);

        struct timeval timeVal;
        timeVal.tv_sec = 3; // 3秒超时
        timeVal.tv_usec = 0;

        int selectResult = select(0, &readfds, NULL, NULL, &timeVal);

        if (selectResult > 0) {
            // 有数据可读
            int recvLen = recvfrom(sockRaw, recvBuf, sizeof(recvBuf), 0, (sockaddr*)&fromAddr,
&fromLen);
            if (recvLen == SOCKET_ERROR) {
                cout << ttl << "\t*\t\t目标不可达 (Recv Error)" << endl;
            }
            else {
                // 收到数据，解析 IP 头和 ICMP 头
                DWORD endTime = GetTickCount();
                DWORD rtt = endTime - startTime;

                // 解析接收到的 IP 头
                IpHeader* recvIpHdr = (IpHeader*)recvBuf;
                int ipHdrLen = recvIpHdr->h_len * 4; // IP头长度单位是4字节

                // 确保接收到的长度足够包含 IP头 + ICMP头
                if (recvLen >= ipHdrLen + (int)sizeof(IcmpHeader)) {
                    // 定位到 ICMP 头部
                    IcmpHeader* recvIcmpHdr = (IcmpHeader*)(recvBuf + ipHdrLen);

                    // 处理不同类型的 ICMP 响应
                    if (recvIcmpHdr->i_type == ICMP_TIMEOUT) {
                        // 类型 11: TTL 超时（这是路径上的中间路由器）
                        cout << ttl << "\t";
                        if (rtt < 1) cout << "<1ms";
                        else cout << rtt << "ms";
                        cout << "\t\t" << inet_ntoa(fromAddr.sin_addr) << endl;

                    }
                    else if (recvIcmpHdr->i_type == ICMP_ECHO_REPLY) {
                        // 类型 0: 回显应答（到达目的地）
                        // 检查 ID 是否匹配（确认是本进程发的包）
                        if (recvIcmpHdr->i_id == (unsigned short)processId) {
                            cout << ttl << "\t";
                            if (rtt < 1) cout << "<1ms";
                            else cout << rtt << "ms";
```

```
                        cout << "\t\t" << inet_ntoa(fromAddr.sin_addr) << endl;
                        reachedDest = true;
                    }
                    else {
                        // ID 不匹配，可能是其他程序的包或旧包
                        cout << ttl << "\t*\t\t目标不可达 (ID mismatch)" << endl;
                    }

                }
                else {
                    // 其他类型，如类型 3 (目标不可达)
                    cout << ttl << "\t";
                    if (rtt < 1) cout << "<1ms";
                    else cout << rtt << "ms";
                    cout << "\t\t" << inet_ntoa(fromAddr.sin_addr)
                        << " (Type " << (int)recvIcmpHdr->i_type << ")" << endl;
                }
            }
        }
    }
    else {
        // select 返回 0 (超时) 或 < 0 (错误)
        cout << ttl << "\t*\t\t请求超时" << endl;
    }

    if (reachedDest) break;
}

// 7. 结束程序
cout << "本次跟踪完成，请按任意键结束程序" << endl;

closesocket(sockRaw);
WSACleanup();

cin.get(); // 等待用户按键
return 0;
}
```

四、实验结果及分析

五、实验中问题及分析

在使用原始套接字进行 Traceroute 实验时，最常遇到的首要障碍是权限不足导致程序无法启动。当你运行程序时，可能会立即收到“无法创建套接字”的错误提示，错误码通常为 10013。这是因为代码中使用了 SOCK_RAW，这种类型的套接字赋予了程序直接构建 IP 数据包头部的能力，拥有极高的网络控制权，操作系统出于安全考虑，默认禁止普通用户使用。解决这个问题的唯一办法是提升权限，以“管理员身份”启动编译器或命令提示符，然后再执行编译好的程序，这样才能获得操作系统内核的许可来创建原始套接字。

另一个是防火墙拦截导致的全程超时，即程序运行后每一跳都显示星号（*），看起来像是网络断开了。这通常不是代码错误，而是因为 Windows 防火墙或杀毒软件默认会拦截入站的 ICMP 差错报文（如“超时”或“目标不可达”），导致程序发得出去但收不到回音。此外，如果目标主机（如某些高防服务器）配置了禁止 Ping，也会导致最后一跳无法响应。为了解决这个问题，需要在实验期间暂时关闭本地防火墙，或者在防火墙的高级设置中允许 ICMP 回显请求和应答。同时，建议测试时选择像 8.8.8.8 或 114.114.114.114 这样公认允许 ICMP 的公共 DNS 地址作为目标，以排除目标主机配置的原因。

在数据处理逻辑上，容易出现接收到无关数据包（串扰）的问题。由于原始套接字的特性，它会接收到达本机的所有 ICMP 协议数据包。如果在运行 Traceroute 程序的同一台机器上，后台还开着 ping 命令或者浏览器正在访问网络，程序 recvfrom 可能会意外收到这些不属于当前进程的 ICMP 回应，导致输出错误的 IP 或逻辑混乱。解决这个问题的关键在于严格的“身份验证”，代码中利用 GetCurrentProcessId() 获取当前进程 ID 并填入 ICMP 头部的 i_id 字段是至关重要的。在

接收数据时，必须检查收到的 ICMP 包头部 ID 是否与本进程 ID 一致，如果不一致则说明这是其他程序的包，应当直接忽略并继续等待，而不是将其视为有效的响应。

此外，可能会发现部分中间节点始终显示请求超时，但后续节点和最终目标却能正常响应。我开始以为是程序在某一跳出了 Bug。实际上，这是互联网路由设备的正常行为，许多骨干网路由器为了减轻 CPU 负担或出于隐藏网络拓扑的安全策略，被配置为不回复 TTL 超时的 ICMP 报文，直接丢弃数据包。针对这种情况，代码层面无法强制路由器回复，但可以在程序中增加重试机制，例如每一跳发送三次数据包，只有当三次全部超时才认定该节点不可达，这样也能有效减少因网络抖动造成的偶然丢包。

六、心得体会

(一) 重新认识 TTL 机制

通过亲手编写 Traceroute，我最大的收获是将书本上枯燥的“TTL”概念转化为了直观的认知。过去我只知道 TTL 是为了防止数据包在网络中无限循环，而本次实验让我看到，Traceroute 正是巧妙利用了这一“错误处理机制”——故意发送 TTL 逐渐递增的数据包，迫使沿途的路由器丢包并返回 ICMP Time Exceeded 消息，从而“骗取”到了路径信息。这种“利用协议特性反向探测”的设计思路，让我对 TCP/IP 协议设计的灵活性和健壮性有了更深的理解。

(二) Raw Socket 的威力

在使用原始套接字的过程中，我体会到了与普通 TCP/UDP 编程截然不同的自由度与复杂度。平时编程我们只需关心应用层数据，而这次我需要手动去填充 IP 头和 ICMP 头的每一个字节——计算校验和、处理字节序转换（网络字节序与主机字节序）、设置协议字段等。这不仅让我明白了操作系统在幕后默默完成了多少工作，也让我理解了为什么这类操作需要管理员权限：因为拥有了构建底层包的能力，同时也意味着拥有了伪造数据包（如 IP 欺骗）的风险，安全与能力永远是并存的。

(三) 直面真实网络的复杂性

实验过程中遇到的最大困难并非代码本身，而是真实网络环境的“不确定性”。在理论模型中，网络是畅通的；但在实际测试中，我遭遇了防火墙拦截、路由器配置静默丢包（显示为星号）、以及网络抖动导致的 RTT 剧烈波动。这迫使我在代码中引入了超时重试机制、身份标识（ID）过滤机制等防御性编程手段。我深刻体会到，网络编程的核心难点往往不在于发送数据，而在于如何在一个不可靠的传输媒介上，通过逻辑判断和错误处理来构建可靠的业务逻辑。

(四) 内存管理与位运算

通过将字符数组强制转换为结构体指针（如 (IpHeader*)buffer）来解析数据，我必须对内存对齐和结构体布局了如指掌。此外，校验和计算中的二进制反码求和逻辑，也让我复习了底层的位运算知识。这些在高级语言开发中容易被忽视的基本功，在底层网络编程中却是决定程序能否正常运行的基石。