哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2025 秋季

课程名称：计算机网络

实验名称：协议栈设计与实现

学生班级：未公开的情报

学生学号：未公开的情报

学生姓名：Hanatani Takahiro-Github

评阅教师：

报告成绩：

实验与创新实践教育中心制

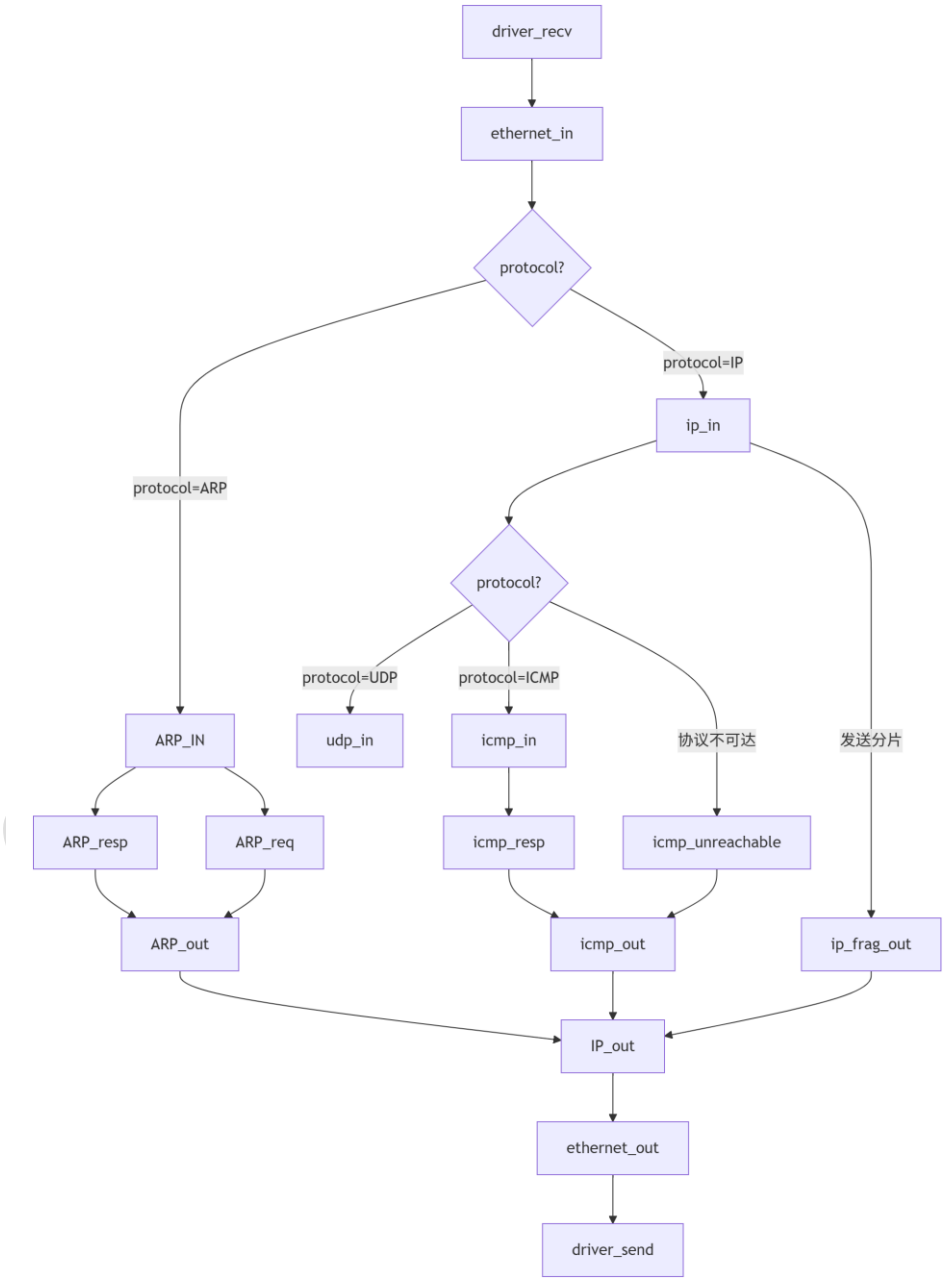
2025 年 10 月

一、 协议实现详述

（注意不要完全照搬实验指导书上的内容，请根据你自己的设计方案来填写图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。）

1. 请给出协议栈实验的整体流程图

（绘制协议栈实验的整体流程图，涵盖协议栈接收和发送主要步骤，包括 Eth 接收 / 发送、ARP 处理、IP 接收 / 发送、ICMP 处理、UDP 接收 / 发送、TCP 接收 / 发送以及 web 服务器请求处理等步骤，并标注主要函数调用关系。）



2. Eth 协议详细设计

（描述以太网（Eth）协议的数据封装与解封装过程等。）

整体设计思路

以太网（Ethernet）协议位于 TCP/IP 协议栈的数据链路层，其主要功能是完成数据帧的封装与解封装。在发送数据时，为上层协议的数据包添加以太网包头；在接收数据时，解析并移除以太网包头，然后将数据传递给上层协议处理。

以太网帧格式如下：

MAC 目标地址 (6 字节)	MAC 源地址 (6 字节)	长度/类型 (2 字节)	数据 (46-1500 字节)	FCS (4 字节)

本实验的主要任务是实现以太网数据帧的发送和接收处理函数。

（1）以太网数据帧发送处理流程（ethernet_out 函数）的设计与实现

流程设计：

1. **数据长度检查与填充**：检查数据长度是否小于最小传输单元（46 字节），如果不足则填充 0
2. **添加以太网包头**：调用 **buf_add_header()**函数为数据包添加以太网头部空间
3. **填写目的 MAC 地址**：将参数中的目标 MAC 地址复制到包头
4. **填写源 MAC 地址**：将本机 MAC 地址复制到包头
5. **填写协议类型**：将上层协议类型转换为网络字节序后写入包头
6. **发送数据帧**：调用 **driver_send()**函数发送数据

```
void ethernet_out(buf_t *buf, const uint8_t *mac, net_protocol_t protocol)
{
    // Step1: 数据长度检查与填充
    if(buf->len <ETHERNET_MIN_TRANSPORT_UNIT) {
        int padding_len = ETHERNET_MIN_TRANSPORT_UNIT - buf->len;
        buf_add_padding(buf, padding_len);
    }
    //step2:添加包头
    buf_add_header(buf, sizeof(ether_hdr_t));
    ether_hdr_t *hdr = (ether_hdr_t *)buf->data;
    //step3:添加目的 mac
    for(int i = 0; i < NET_MAC_LEN; i++) {
        hdr->dst[i] = mac[i];
    }
    //step4:添加源 mac
    uint8_t src_mac[] = NET_IF_MAC;
    memcpy(hdr->src, src_mac, NET_MAC_LEN);
    //step5:填写协议类型（转化成大端序）
    hdr->protocol16 = swap16(protocol);
    //step6:发送数据帧
    driver_send(buf);
}
```

（2）以太网数据帧接收处理流程（ethernet_in 函数）的设计与实现

流程设计：

1. **数据长度检查**：检查数据包长度是否小于以太网头部长度，如果小于则丢弃
2. **解析以太网包头**：解析包头获取协议类型和源 MAC 地址
3. **移除以太网包头**：调用 buf_remove_header()函数移除包头
4. **向上层传递数据**：调用 net_in()函数将数据传递给上层协议

```
void ethernet_in(buf_t *buf) {
    //step1:数据长度检查
    if(buf->len < 14) {
        return;
    }
    //step2:移除包头
    ether_hdr_t *hdr = (ether_hdr_t *)buf->data;
    buf_remove_header(buf, 14);
    //step3:向上传递数据包
    // 注意：协议类型需要转换为大端序，所以使用 swap16
    net_in(buf, swap16(hdr->protocol16), hdr->src);
}
```


3. ARP 协议详细设计

（描述 ARP 请求/响应处理逻辑、ARP 表项的更新机制等。）

整体设计思路

ARP 协议用于将网络层的 IP 地址解析为数据链路层的 MAC 地址。主要功能包括：

- 发送 ARP 请求查询目标 IP 对应的 MAC 地址
- 响应 ARP 请求，提供本机的 MAC 地址

- 维护 ARP 缓存表，提高解析效率

（1）ARP 请求函数（arp_req）的设计与实现

流程设计：

1. 初始化缓冲区，长度为 ARP 报文大小
2. 填写 ARP 报头，使用预定义的初始 ARP 包
3. 修改操作类型为 ARP_REQUEST，设置目标 IP 地址
4. 通过以太网层发送 ARP 请求（使用广播 MAC 地址）

```
5. void arp_req(uint8_t *target_ip) {
6.     // Step1: 初始化缓冲区
7.     buf_init(&txbuf, sizeof(arp_pkt_t));
8.
9.     // Step2: 填写 ARP 报头
10.    arp_pkt_t *pkt = (arp_pkt_t *)txbuf.data;
11.    memcpy(pkt, &arp_init_pkt, sizeof(arp_pkt_t));
12.
13.    // Step3: 设置操作类型和目标 IP 地址
14.    // 注意：操作类型需要从主机字节序转换为网络字节序
15.    pkt->opcode16 = swap16(ARP_REQUEST);
16.    memcpy(pkt->target_ip, target_ip, NET_IP_LEN);
17.
18.    // Step4: 发送 ARP 请求（使用广播地址）
19.    // ARP 请求是广播报文，目标 MAC 地址为全 FF
20.    ethernet_out(&txbuf, ether_broadcast_mac, NET_PROTOCOL_ARP);
21.}
```

（2）ARP 发送处理函数（arp_out）的设计与实现

流程设计：

1. 根据目标 IP 地址查询 ARP 缓存表
2. 如果找到对应 MAC 地址，直接发送数据包
3. 如果未找到，检查是否已有等待该 IP 响应的数据包
4. 如果没有，缓存当前数据包并发送 ARP 请求

```
5. void arp_out(buf_t *buf, uint8_t *ip) {
6.     // Step1: 查找 ARP 表
7.     uint8_t *mac = (uint8_t *)map_get(&arp_table, ip);
8.
9.     if (mac != NULL) {
10.        // Step2: 找到对应 MAC 地址，直接发送
11.        ethernet_out(buf, mac, NET_PROTOCOL_IP);
12.    } else {
13.        // Step3: 未找到对应 MAC 地址
14.        // 检查是否已经有等待该 IP 的缓存包
15.        buf_t *cached_buf = (buf_t *)map_get(&arp_buf, ip);
16.
```

```
        if (cached_buf != NULL) {
            // 已经有缓存包，说明正在等待 ARP 响应，不再发送 ARP 请求
            return;
        } else {
            // 没有缓存包，缓存当前数据包并发送 ARP 请求
            map_set(&arp_buf, ip, buf);
            arp_req(ip);
        }
    }
}

```

（3）ARP 接收处理函数（arp_in）的设计与实现

流程设计：

1. 检查数据包长度，确保完整性
2. 验证 ARP 报头各字段是否符合协议规定
3. 更新 ARP 缓存表
4. 处理缓存的等待数据包或响应 ARP 请求

```
 */
void arp_in(buf_t *buf, uint8_t *src_mac) {
    // Step1: 检查数据包长度
    if (buf->len < sizeof(arp_pkt_t)) {
        return; // 数据包不完整，丢弃
    }

    // Step2: 报头检查
    arp_pkt_t *arp_pkt = (arp_pkt_t *)buf->data;

    // 检查硬件类型（以太网）
    if (swap16(arp_pkt->hw_type16) != ARP_HW_ETHER) {
        return;
    }

    // 检查上层协议类型（IP）
    if (swap16(arp_pkt->pro_type16) != NET_PROTOCOL_IP) {
        return;
    }

    // 检查硬件地址长度
    if (arp_pkt->hw_len != NET_MAC_LEN) {
        return;
    }

```

```
    // 检查协议地址长度
    if (arp_pkt->pro_len != NET_IP_LEN) {
        return;
    }

    // 检查操作类型（ARP_REQUEST 或 ARP_REPLY）
    uint16_t opcode = swap16(arp_pkt->opcode16);
    if (opcode != ARP_REQUEST && opcode != ARP_REPLY) {
        return;
    }

    // Step3: 更新 ARP 表项
    // 将发送方的 IP-MAC 映射存入 ARP 表
    map_set(&arp_table, arp_pkt->sender_ip, arp_pkt->sender_mac);

    // Step4: 查看缓存情况
    // 检查是否有等待该 IP 地址 MAC 地址的数据包
    buf_t *cached_buf = (buf_t *)map_get(&arp_buf,
arp_pkt->sender_ip);

    if (cached_buf != NULL) {
        // 有缓存的数据包，说明之前因为不知道 MAC 地址而无法发送
        // 现在知道了 MAC 地址，可以发送缓存的数据包了
        ethernet_out(cached_buf, arp_pkt->sender_mac,
NET_PROTOCOL_IP);

        // 发送后删除缓存
        map_delete(&arp_buf, arp_pkt->sender_ip);
    } else {
        // 没有缓存，检查是否是 ARP 请求且目标 IP 是本机
        if (opcode == ARP_REQUEST) {
            // 检查目标 IP 是否为本机 IP
            if (memcmp(arp_pkt->target_ip, net_if_ip, NET_IP_LEN) ==
0) {
                // 是本机，发送 ARP 响应
                arp_resp(arp_pkt->sender_ip, arp_pkt->sender_mac);
            }
        }
    }
}

```

（4）ARP 响应函数（arp_resp）的设计与实现

流程设计：

1. 初始化缓冲区
2. 填写 ARP 报头
3. 设置操作类型为 ARP_REPLY，填写目标 IP 和 MAC 地址
4. 通过以太网层发送 ARP 响应

```
void arp_resp(uint8_t *target_ip, uint8_t *target_mac) {
    // Step1: 初始化缓冲区
    buf_init(&txbuf, sizeof(arp_pkt_t));

    // Step2: 填写 ARP 报头
    arp_pkt_t *pkt = (arp_pkt_t *)txbuf.data;
    memcpy(pkt, &arp_init_pkt, sizeof(arp_pkt_t));

    // Step3: 设置操作类型、目标 IP 地址和目标 MAC 地址
    // 注意：操作类型需要从主机字节序转换为网络字节序
    pkt->opcode16 = swap16(ARP_REPLY);
    memcpy(pkt->target_ip, target_ip, NET_IP_LEN);
    memcpy(pkt->target_mac, target_mac, NET_MAC_LEN);

    // Step4: 发送 ARP 响应（直接发送给请求者）
    ethernet_out(&txbuf, target_mac, NET_PROTOCOL_ARP);
}
```


4. IP 协议详细设计

（描述 IP 数据包的封装与解封装、IP 数据包的分片、校验和计算等。）

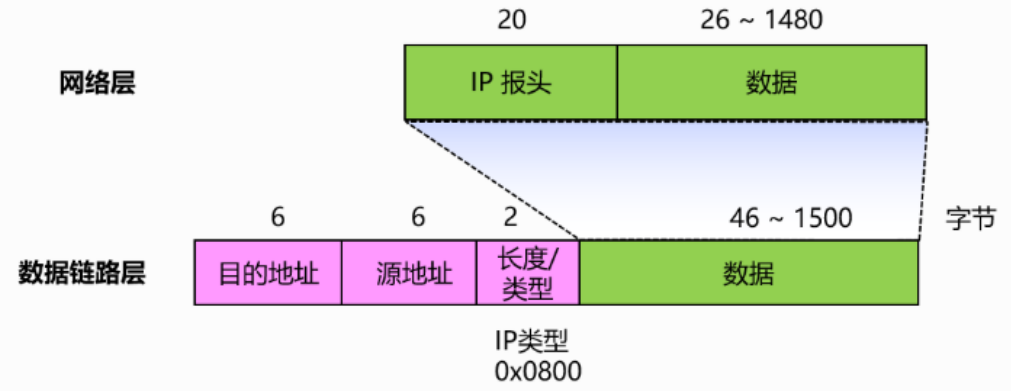
IP 协议位于网络层，提供无连接、尽力而为的数据包传输服务。本次实验实现 IP 数据报的收发、头部解析、校验和计算与分片发送功能，路由转发暂未涉及。

此次实验的重点：收到数据包时对 IP 头部进行解析，获得相关信息，然后再去除 IP 头部；发送数据包时，对 IP 头部的字段进行填充并对 IP 数据包进行分片操作（如有必要），然后进行发送。

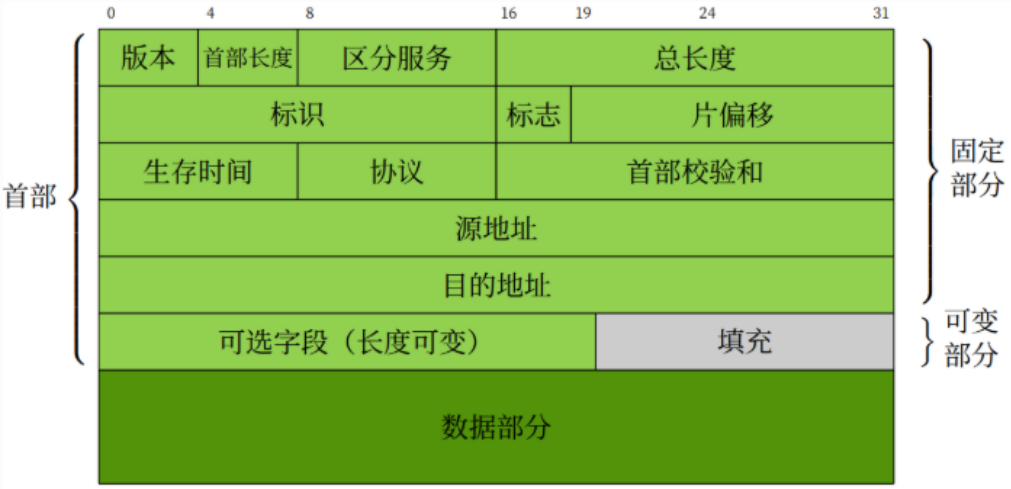
由于 IP 头部还含有首部校验和字段，故本小节实验还需要再完成校验和计算函数的设

计 。

网络层位于数据链路层之上，其数据包的结构如下所示：



本实验是基于 IPv4 协议展开的，下面将重点介绍 IPv4 版本的 IP 协议。IPv4 的头部结构长度固定为 20 字节，如果包含可变长的选项部分，最多可达 60 字节。



要考虑的字段主要有：版本号、头部长度、区分服务 TOS、总长度、标识 id、标志、片偏移、生存时间、协议、头部校验和、 源 IP 地址、目的 IP 地址等。

IP 数据包输入处理函数 ip_in

设计流程：

1. **长度检查**：检查数据包长度是否小于 IP 头部最小长度
2. **版本验证**：确保协议版本为 IPv4
3. **长度验证**：检查总长度字段是否合理
4. **校验和计算**：重新计算校验和并与接收到的校验和比较
5. **目的 IP 匹配**：确认数据包是发送给本机的
6. **去除填充**：移除可能存在的填充字段
7. **向上传递**：去除 IP 头部后传递给上层协议

```
void ip_in(buf_t *buf, uint8_t *src_mac) {
    // TO-DO
    ip_hdr_t *hdr = (ip_hdr_t*)buf->data;
```

```
    uint16_t total_len = swap16(hdr->total_len16);
    uint16_t ogn_checksum = hdr->hdr_checksum16;//将原来的校验和保存起来
    size_t hdr_len = hdr->hdr_len * 4;//计算头部长度（以字节为单位）
    //step1：检查数据包长度
    if (buf->len < sizeof(ip_hdr_t)) {
        return;//小于 ip 头部长度直接丢弃
    }

    //step2:进行报头检测
    //检查版本号
    if (hdr->version != IP_VERSION_4) {
        return;//版本号不为 4 直接丢弃
    }
    //检查总长度
    if (total_len > buf->len) {
        return;//总长度字段大于收到的数据包长度，直接丢弃
    }
    //step3：校验头部校验和
    hdr->hdr_checksum16 = 0;//将校验和字段设为 0
    uint16_t new_checksum = checksum16((uint16_t*)hdr, hdr_len);//计算校
验和
    if (new_checksum != ogn_checksum) {
        return;//如果计算得到的校验和与原来的校验和不相等，那么丢弃
    } else {
        hdr->hdr_checksum16 = ogn_checksum;
    }
    //step4：对比目的 ip 地址
    if (memcmp(net_if_ip, hdr->dst_ip, NET_IP_LEN) != 0) {
        return;//如果目的 ip 不是本机 ip，将数据包丢弃
    }
    //step5:去除填充字段
    if (total_len < buf->len) {
        buf_remove_padding(buf, buf->len - total_len);//去除填充字段
    }
    //step6：去掉 ip 报头
    // ip_hdr_t *ogn_hdr;
    // memcpy(ogn_hdr, hdr, hdr_len);//把 ip 头拷贝下来，恢复时使用
    buf_remove_header(buf, hdr_len);
    //step7：向上传递数据包
    int message = net_in(buf, hdr->protocol, hdr->src_ip);
    if (message == -1) {
        buf_add_header(buf, hdr_len);
        memcpy(buf->data, hdr, hdr_len);//如果返回不可达信息，重新添加 ip 报
头
```

```
        icmp_unreachable(buf, hdr->src_ip, ICMP_CODE_PROTOCOL_UNREACH);
    }
}
```

校验和计算函数 checksum16

设计原理：

IP 校验和采用 16 位反码求和算法，具有字节序无关性。计算过程如下：

1. 将数据按 16 位分组求和
2. 将求和结果的高 16 位与低 16 位不断相加，直到高 16 位为 0
3. 对最终结果取反得到校验和

```
uint16_t checksum16(uint16_t *data, size_t len) {
    // TO-DO
    uint32_t sum = 0;
    size_t i;

    // Step1: 按 16 位分组相加
    for (i = 0; i < len / 2; i++) {
        sum += data[i];
    }

    // Step2: 处理剩余 8 位（如果有）
    if (len % 2 != 0) {
        // 将剩余的 8 位数据作为高 8 位，低 8 位补 0
        uint8_t *byte_data = (uint8_t *)data;
        sum += (uint16_t)byte_data[len - 1];
    }

    // Step3: 循环处理高 16 位
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }

    // Step4: 取反得到校验和
    return (uint16_t)~sum;
}
```

关键注意点：

- 校验和计算不需要考虑大小端转换，因为反码求和的数学特性保证了字节序无关性
- 计算前需要将校验和字段置为 0
- 使用 32 位变量存储求和结果以避免溢出
-

IP 分片发送函数 ip_fragment_out

设计流程：

1. **添加头部空间**：为 IP 头部分配空间
2. **填充头部字段**：包括版本、长度、标识、标志、片偏移等
3. **计算校验和**：先清零后计算

4. **发送数据**：调用 ARP 协议进行发送

```
void ip_fragment_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol, int id, uint16_t offset, int mf) {
    // TO-DO
    //step1：增加头部
    buf_add_header(buf, sizeof(ip_hdr_t));

    //step2：填写头部字段
    ip_hdr_t *hdr = (ip_hdr_t*)buf->data;
    hdr->hdr_len = 5;//填写首部长度
    hdr->version = IP_VERSION_4;//填写版本号
    hdr->tos = 0;//填写服务类型
    hdr->total_len16 = swap16(buf->len);//填写总长度字段
    hdr->id16 = swap16(id);//填写标识字段
    hdr->flags_fragment16 = 0;//填写标志和片偏移字段
    if (mf) {
        //如果 mf 有效，填写 mf 标志
        hdr->flags_fragment16 |= IP_MORE_FRAGMENT;
    }
    uint16_t offset_units = offset / 8;  // 需要转换为 8 字节单位
    hdr->flags_fragment16 |= (offset_units & 0x1FFF);//分片偏移为该字段的后十三位
    hdr->flags_fragment16 = swap16(hdr->flags_fragment16);//别忘了字节序转换
    hdr->ttl = 64;//填写生存时间
    hdr->protocol = protocol;//填写协议
    memcpy(hdr->dst_ip, ip, NET_IP_LEN);//填写目的 IP 地址
    memcpy(hdr->src_ip, net_if_ip, NET_IP_LEN);//填写源 IP

    hdr->hdr_checksum16 = 0;//先将校验和字段清零，再计算校验和
    hdr->hdr_checksum16 = checksum16((uint16_t*)hdr, sizeof(ip_hdr_t));

    arp_out(buf, ip);//发送
}

```

IP 数据包输出函数 ip_out

设计流程：

1. **检查数据包长度**：判断是否需要分片
2. **分片处理**：如果长度超过 MTU，将数据分割成多个分片
3. **发送分片**：每个分片调用 ip_fragment_out 发送
4. **直接发送**：如果不需要分片，直接发送完整数据包

```
static int packet_id = 0;//全局 id，每次发送时递增
```

```
6. void ip_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol) {
7.     // TO-DO
8.     //step1：检查数据报包长是否大于 MTU 减去 ip 首部长度
9.     int id = packet_id++;//递增 id
10.
11.    if (buf->len > 1480) {
12.        //step2：如果大于最大包长，分片处理
13.        size_t fragments;
14.        if (buf->len % 1480 == 0) {
15.            fragments = buf->len / 1480;//计算总片数，分类讨论能不能
    整除
16.        } else {
17.            fragments = buf->len / 1480 + 1;
18.        }
19.
20.        for (int i = 0; i < fragments; i++) {
21.            buf_t *ip_buf = (buf_t*)malloc(sizeof(buf_t));
22.
23.            if (ip_buf == NULL) {
24.                //处理内存分配失败
25.                return;
26.            }
27.
28.            size_t fragments_offset = i * 1480;//计算每个分片的偏移
    量字段
29.            size_t fragments_len;//初始化每个分片的长度字段
30.            int fragments_mf;
31.
32.            if (i == fragments - 1) {
33.                fragments_len = buf->len - fragments_offset;//计算
    最后一个分片的长度
34.                fragments_mf = 0;
35.            } else {
36.                fragments_len = 1480;//其余分片都是最大长度
37.                fragments_mf = 1;
38.            }
39.
40.            buf_init(ip_buf, fragments_len);
41.
42.            //截断原数据包对应片段到 ip_buf 中
43.            memcpy(ip_buf->data, buf->data + fragments_offset,
    fragments_len);
44.            ip_buf->len = fragments_len;
45.
```

```
            //调用分片发送函数
            ip_fragment_out(ip_buf, ip, protocol, id,
    fragments_offset, fragments_mf);
        }
    } else {
        //step3：数据包长度小于最大长度，直接发送
        ip_fragment_out(buf, ip, protocol, id, 0, 0);
    }
}

```

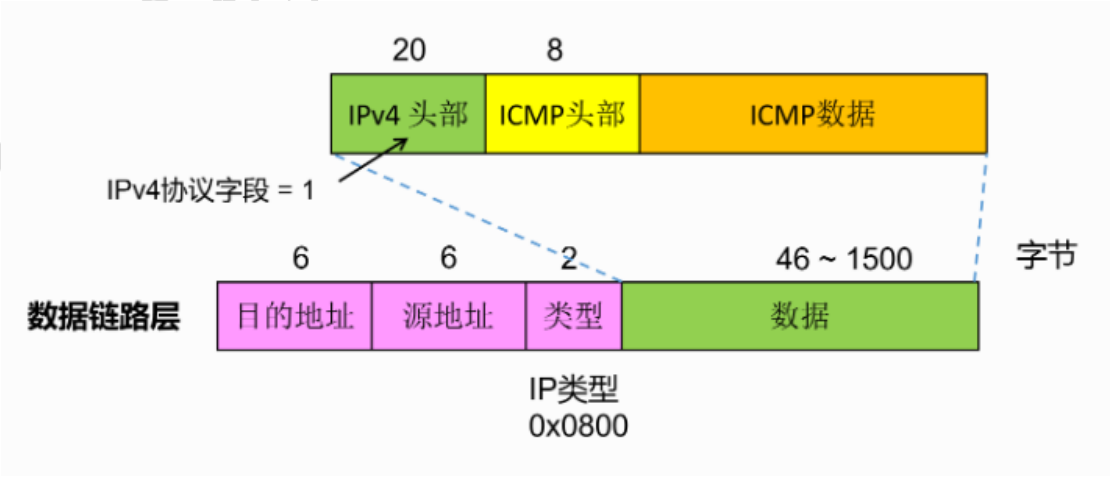
5. ICMP 协议详细设计

（解释如何处理 ICMP 请求和响应，以及如何利用 ICMP 报文进行网络故障诊断等。）

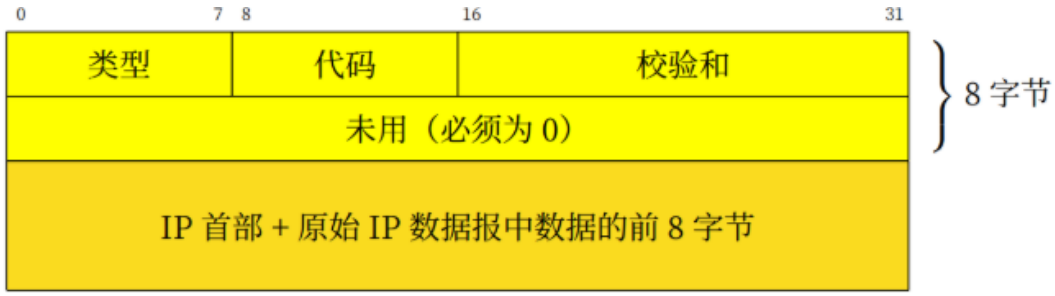
ICMP（Internet Control Message Protocol）协议与 IP 协议协同工作，为网络通信提供差错报告和诊断功能。它主要用于检测网络连通性、报告传输错误以及改善网络配置。ICMP 报文被封装在 IP 数据报中传输，根据功能可分为两类：差错报文和查询报文。本实验主要实现 ICMP 回显请求/应答（ping 功能）以及不可达差错报文。

ICMP 报文的基本结构如下图所示：

本次实验只需处理 ICMP 查询报文中的回显报文，其结构如下图所示：



本次实验只需处理 ICMP 查询报文中的回显报文，其结构如下图所示：



1）ICMP 数据报输入处理函数 icmp_in 的设计与实现

流程设计：

1. **报文头检测**：检查接收到的数据包长度是否小于 ICMP 头部长度，若是则丢弃。
2. **类型判断**：解析 ICMP 头部，判断报文类型是否为回显请求（ICMP_TYPE_ECHO_REQUEST）。
3. **响应处理**：若为回显请求，则调用 icmp_resp 函数发送回显应答。

实现要点：

- ICMP 报文直接使用 IP 层传递的缓冲区，无需额外内存分配。
- 仅处理回显请求报文，其他类型报文在本实验中忽略。

代码实现：

```
void icmp_in(buf_t *buf, uint8_t *src_ip) {
    // TO-DO
    //step1: 报头检测
    if (buf->len < sizeof(icmp_hdr_t)) {
        //若长度小于 icmp 首部长度，丢弃
        return;
    }
    //step2: 检查 ICMP 类型，如果是请求就发送应答
    icmp_hdr_t *hdr = (icmp_hdr_t*)buf->data;
    if (hdr->type == ICMP_TYPE_ECHO_REQUEST) {
        icmp_resp(buf, src_ip);
    }
}
```

（2）ICMP 响应报文发送函数 icmp_resp 的设计与实现

流程设计：

1. **初始化缓冲区**：创建与请求报文相同大小的发送缓冲区。
2. **复制数据**：将请求报文的数据完整复制到发送缓冲区。
3. **修改头部字段**：将类型改为回显应答（ICMP_TYPE_ECHO_REPLY），代码设为 0，保持标识符和序列号不变。
4. **计算校验和**：先将校验和字段置 0，然后调用 checksum16 函数计算。
5. **发送数据报**：调用 ip_out 函数将 ICMP 应答报文发送出去。

实现要点：

- 保持标识符和序列号不变，确保请求与应答的对应关系。
- 重新计算校验和，因为类型字段已改变。

代码实现：

```c
static void icmp_resp(buf_t *req_buf, uint8_t *src_ip) {
    // Step1: 初始化并封装数据
    buf_t txbuf;
    buf_init(&txbuf, req_buf->len);  // 创建与请求相同大小的缓冲区
    memcpy(txbuf.data, req_buf->data, req_buf->len);  // 复制请求数据

    // Step2: 修改头部字段
    icmp_hdr_t *hdr = (icmp_hdr_t *)txbuf.data;
    icmp_hdr_t *req_hdr = (icmp_hdr_t *)req_buf->data;

    hdr->type = ICMP_TYPE_ECHO_REPLY;  // 修改为回显应答
    hdr->code = 0;                     // 代码为 0
    hdr->id16 = req_hdr->id16;         // 保持标识符不变
    hdr->seq16 = req_hdr->seq16;       // 保持序列号不变

    // Step3: 填写校验和
    hdr->checksum16 = 0;  // 先将校验和字段置 0
    hdr->checksum16 = checksum16((uint16_t *)txbuf.data, txbuf.len);

    // Step4: 发送数据报
    ip_out(&txbuf, src_ip, NET_PROTOCOL_ICMP);
}
```

（3）ICMP 差错报文发送函数 icmp_unreachable 的设计与实现

流程设计：

1. **初始化缓冲区**：计算报文总长度（ICMP 头部 + IP 头部 + 前 8 字节数据），创建相应大小的缓冲区。
2. **填充 ICMP 头部**：设置类型为不可达（ICMP_TYPE_UNREACH），代码根据参数设置（协议不可达或端口不可达），标识符和序列号设为 0。
3. **填充数据部分**：包含原始 IP 数据报的头部和前 8 字节数据。
4. **计算校验和**：先将校验和字段置 0，然后计算整个 ICMP 报文的校验和。
5. **发送数据报**：调用 ip_out 函数将 ICMP 不可达报文发送出去。

实现要点：

- 根据 RFC 规定，ICMP 差错报文必须包含原始 IP 数据报的头部和前 8 字节数据。
- 校验和计算需覆盖整个 ICMP 报文，包括头部和数据部分。

代码实现：

```c
void icmp_unreachable(buf_t *recv_buf, uint8_t *src_ip, icmp_code_t code)
{
    // Step1: 初始化并填写报头
    int total_size = sizeof(icmp_hdr_t) + sizeof(ip_hdr_t) + 8;
    buf_t txbuf;
    buf_init(&txbuf, total_size);
```

```
    // 填充 ICMP 头部
    icmp_hdr_t *icmp_hdr = (icmp_hdr_t *)txbuf.data;
    icmp_hdr->type = ICMP_TYPE_UNREACH;  // 类型为不可达
    icmp_hdr->code = code;               // 根据参数设置代码
    icmp_hdr->id16 = 0;                  // 标识符为 0
    icmp_hdr->seq16 = 0;                 // 序列号为 0

    // Step2: 填写数据与校验和
    // 填写 IP 数据报首部
    ip_hdr_t *ip_hdr = (ip_hdr_t *)(icmp_hdr + 1);
    ip_hdr_t *recv_ip_hdr = (ip_hdr_t *)recv_buf->data;
    memcpy(ip_hdr, recv_ip_hdr, sizeof(ip_hdr_t));

    // 填写 IP 数据报的前 8 个字节数据字段
    uint8_t *data = (uint8_t *)(ip_hdr + 1);
    uint8_t *recv_ip_data = (uint8_t *)(recv_ip_hdr + 1);
    memcpy(data, recv_ip_data, 8);

    // 计算校验和
    icmp_hdr->checksum16 = 0;
    icmp_hdr->checksum16 = checksum16((uint16_t *)txbuf.data, total_size);

    // Step3: 发送数据报
    ip_out(&txbuf, src_ip, NET_PROTOCOL_ICMP);
}
```

6. UDP 协议详细设计

（描述 UDP 数据包的封装与解封装、UDP 校验和计算等。）

（1）UDP 数据报输出处理函数 udp_out 的设计与实现

功能：封装 UDP 数据包并发送给 IP 层。

设计思路：

1. 添加 UDP 头部空间
2. 填充 UDP 头部各字段（源端口、目的端口、长度）
3. 计算并填充校验和（使用伪头部）
4. 调用 IP 层发送函数

实现代码：

```
void udp_out(buf_t *buf, uint16_t src_port, uint8_t *dst_ip, uint16_t
dst_port) {
    // Step1: 添加 UDP 头部
    buf_add_header(buf, sizeof(udp_hdr_t));

    // Step2: 填充 UDP 头部字段
    udp_hdr_t *hdr = (udp_hdr_t *)buf->data;
    hdr->src_port16 = swap16(src_port);
    hdr->dst_port16 = swap16(dst_port);
    hdr->total_len16 = swap16(buf->len);

    // Step3: 计算校验和
    hdr->checksum16 = 0;
    hdr->checksum16 = transport_checksum(NET_PROTOCOL_UDP, buf, net_if_ip,
dst_ip);

    // Step4: 调用 IP 层发送
    ip_out(buf, dst_ip, NET_PROTOCOL_UDP);
}
```

关键点分析：

- 使用 buf_add_header 在数据前预留 UDP 头部空间
- 端口号和长度字段需要进行网络字节序转换
- 校验和计算前先将字段置 0，计算完成后再填充

（2）UDP 数据报输入处理函数 udp_in 的设计与实现

功能：处理接收到的 UDP 数据包，验证完整性并分发给相应应用程序。

设计思路：

1. 基础检查：验证数据包长度是否合法
2. 校验和验证：重新计算校验和，确保数据完整性
3. 端口查找：在端口映射表中查找对应处理函数

4. 分发处理：根据查找结果进行相应处理

实现代码：

```
void udp_in(buf_t *buf, uint8_t *src_ip) {
    // Step1: 包检查
    if (buf->len < sizeof(udp_hdr_t))
        return;

    udp_hdr_t *hdr = (udp_hdr_t *)buf->data;
    uint16_t udp_len = swap16(hdr->total_len16);

    if (buf->len < udp_len)
        return;

    // Step2: 重新计算校验和
    uint16_t checksum_backup = hdr->checksum16;
    hdr->checksum16 = 0;
    uint16_t calc_checksum = transport_checksum(NET_PROTOCOL_UDP, buf,
src_ip, net_if_ip);

    if (calc_checksum != checksum_backup)
        return;
    hdr->checksum16 = checksum_backup;

    // Step3: 查询处理函数
    uint16_t dst_port = swap16(hdr->dst_port16);
    udp_handler_t *handler = map_get(&udp_table, &dst_port);

    if (handler == NULL) {
        // Step4: 端口不可达，发送 ICMP 差错报文
        buf_add_header(buf, sizeof(ip_hdr_t));
        icmp_unreachable(buf, src_ip, ICMP_CODE_PORT_UNREACH);
    } else {
        // Step5: 找到处理函数，去除 UDP 头部并调用
        uint16_t src_port = swap16(hdr->src_port16);
        buf_remove_header(buf, sizeof(udp_hdr_t));
        (*handler)(buf->data, buf->len, src_ip, src_port);
    }
}
```

关键点分析：

- 校验和验证是关键环节，确保数据在传输过程中未受损
- 使用 map_get 在哈希表中快速查找端口对应的处理函数
- 端口不可达时，需要添加 IP 头部以发送 ICMP 差错报文

（3）UDP 校验和计算函数 transport_checksum 的设计与实现

功能：计算传输层协议（UDP/TCP）的校验和，包含伪头部。

设计思路：

UDP 校验和不仅包含 UDP 头部和数据，还包含一个 12 字节的伪头部，用于验证 IP 地址、协议类型等关键信息。

实现代码：

```
uint16_t transport_checksum(uint8_t protocol, buf_t *buf, uint8_t *src_ip,
uint8_t *dst_ip) {
    // Step1 增加 UDP 伪头部
    buf_add_header(buf, sizeof(peso_hdr_t));

    // Step2 暂存被覆盖的数据（通常是 IP 头部）
    uint8_t backup_data[sizeof(peso_hdr_t)];
    memcpy(backup_data, buf->data, sizeof(peso_hdr_t));

    // Step3 填写 UDP 伪头部字段
    peso_hdr_t *pseudo_hdr = (peso_hdr_t *)buf->data;
    memcpy(pseudo_hdr->src_ip, src_ip, NET_IP_LEN);
    memcpy(pseudo_hdr->dst_ip, dst_ip, NET_IP_LEN);
    pseudo_hdr->placeholder = 0;
    pseudo_hdr->protocol = protocol;
    pseudo_hdr->total_len16 = swap16(buf->len - sizeof(peso_hdr_t)); // 传
输层数据包长度

    // Step4 计算校验和（包含伪头部 + 传输层数据）
    int padding = 0;
    if (buf->len % 2 == 1) {
        //对奇数长度的数据包做填充
        buf_add_padding(buf, 1);
        padding = 1;
    }
    uint16_t checksum = checksum16((uint16_t *)buf->data, buf->len);

    // Step5 恢复被覆盖的数据
    memcpy(buf->data, backup_data, sizeof(peso_hdr_t));
    if (padding) {
        buf_remove_padding(buf, 1);//如果有填充，去掉填充
    }

    // Step6 去掉 UDP 伪头部
    buf_remove_header(buf, sizeof(peso_hdr_t));

    // Step7 返回校验和值
    return checksum;
}
```

关键点分析：

- 伪头部包含源/目的 IP 地址、协议类型和 UDP 长度，增强了校验的可靠性
- 需要处理奇数长度数据包的填充问题
- 计算完成后需恢复缓冲区原始状态

7. TCP 协议详细设计

（描述 TCP 连接的建立与关闭过程（三次握手、四次挥手）等。解释如何处理 TCP 数据包的确认、连接状态等问题。）

TCP（Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的传输层协议。它通过序列号、确认应答、重传机制、流量控制和拥塞控制等手段，确保数据在不可靠的网络上可靠、有序、无丢失、不重复地传输。本次实验旨在实现一个简化的 TCP 协议栈，重点完成 TCP 收包处理逻辑。

实验提供的代码框架已包含 TCP 协议的基本结构，包括连接管理、报文发送、校验和计算等功能。本次实验的核心是实现 tcp_in 函数，即 **TCP 收包处理逻辑**，具体包括：

1. **TCP 报文头解析**：提取端口号、序列号、确认号、标志位等字段；
2. **状态机处理**：根据当前连接状态（如 TCP_STATE_LISTEN、TCP_STATE_SYN_RECEIVED 等）和接收到的报文标志位，进行状态转换；
3. **数据交付**：将接收到的数据传递给上层应用；
4. **应答报文构造与发送**：根据处理结果构造并发送 ACK、SYN、FIN 等响应报文。

各部分详细设计与实现

1. TCP 报文头结构

TCP 报文头包含以下关键字段：

- 源端口、目的端口（各 2 字节）
- 序列号、确认号（各 4 字节）
- 数据偏移（4 位，表示首部长度）
- 标志位（ACK、SYN、FIN、RST 等）
- 窗口大小（2 字节）

- 校验和（2 字节）
- 紧急指针（2 字节，可选）

2. TCP 连接建立（三次握手）

- **第一次握手**：客户端发送 SYN 报文，进入 SYN_SENT 状态。
- **第二次握手**：服务端收到 SYN，回复 SYN+ACK，进入 SYN_RECEIVED 状态。
- **第三次握手**：客户端回复 ACK，双方进入 ESTABLISHED 状态，连接建立。

3. TCP 连接释放（四次挥手）

- **第一次挥手**：主动方发送 FIN，进入 FIN_WAIT_1 状态。
- **第二次挥手**：被动方回复 ACK，进入 CLOSE_WAIT 状态。
- **第三次挥手**：被动方发送 FIN，进入 LAST_ACK 状态。
- **第四次挥手**：主动方回复 ACK，进入 TIME_WAIT 状态，等待 2MSL 后关闭。

TCP 数据报输出处理 - tcp_out()函数

实现思路：

Step1: 添加 TCP 报头

- **调用 buf_add_header(buf, sizeof(tcp_hdr_t))为数据缓冲区添加 TCP 报头空间**
- **注意：报头长度固定为 20 字节（标准 TCP 首部长度）**

Step2: 填充 TCP 首部字段

- **获取 TCP 首部指针：tcp_hdr_t *hdr = (tcp_hdr_t*)buf->data**
- **填充各字段：**
  - **win 字段：设置为 TCP_MAX_WINDOW_SIZE（需字节序转换）**
  - **uptr 字段：置为 0（紧急指针）**
  - **ack 字段：当前连接的确认号（需字节序转换）**
  - **doff 字段：首部长度，固定为 TCP_HEADER_LEN << 2（20 字节）**
  - **dst_port16 和 src_port16：目标端口和源端口（需字节序转换）**
  - **flags 字段：传入的标志位参数**
  - **seq 字段：当前连接的序列号（需字节序转换）**

Step3: 计算并填充校验和

1. **先将校验和字段填充为 0：hdr->checksum16 = 0**
2. **调用 transport_checksum(NET_PROTOCOL_TCP, buf, net_if_ip, dst_ip)计算校验和**
3. **将计算结果填入校验和字段：hdr->checksum16 = 计算值**

Step4: 发送 TCP 数据报

- **调用 ip_out(buf, dst_ip, NET_PROTOCOL_TCP)将 TCP 数据报交给 IP 层发送**
- **注意：IP 层会添加 IP 报头并进行网络字节序转换**

关键注意事项：

- **所有多字节字段（端口号、序列号、确认号、窗口大小）都需要进行字节序转换**
- **校验和计算前必须先将校验和字段置零**
- **紧急指针字段在本次实验中统一置零**

TCP 数据报输入处理 - tcp_in()函数状态转移部分

Step1: 更新连接状态与回复报文标志

实现思路：

1. **定义 send_flags 变量用于存储回复报文的标志位**

2. 计算数据部分长度：data_sz = buf->len - tcp_hdr_sz
3. 根据当前 TCP 连接状态进行状态转移：

a) TCP_STATE_LISTEN 状态处理

- 检查条件：仅处理 SYN 报文
- 处理逻辑：
    1. 如果收到的不是 SYN 报文，直接返回
    2. 初始化序列号：tcp_conn->seq = tcp_generate_initial_seq()
    3. 设置确认号：tcp_conn->ack = remote_seq + 1
    4. 设置回复标志：send_flags = TCP_FLG_ACK | TCP_FLG_SYN
    5. 状态转移：tcp_conn->state = TCP_STATE_SYN_RECEIVED

b) TCP_STATE_SYN_RECEIVED 状态处理

- 检查条件：仅处理 ACK 报文
- 处理逻辑：
    1. 如果收到的不是 ACK 报文，直接返回
    2. 状态转移：tcp_conn->state = TCP_STATE_ESTABLISHED

c) TCP_STATE_ESTABLISHED 状态处理

- 顺序检查：先检查序列号是否匹配
- 处理逻辑：
    1. 如果 remote_seq != tcp_conn->ack，发送重复 ACK 后返回
    2. 更新确认号：tcp_conn->ack = remote_seq + data_sz
    3. 如果有数据，设置 ACK 标志：if (data_sz != 0) send_flags = TCP_FLG_ACK
    4. 如果收到 FIN 报文：
        - 设置回复标志：send_flags = TCP_FLG_ACK | TCP_FLG_FIN
        - 确认号加 1：tcp_conn->ack += 1
        - 状态转移：tcp_conn->state = TCP_STATE_LAST_ACK

d) TCP_STATE_LAST_ACK 状态处理

- 检查条件：仅处理 ACK 报文
- 处理逻辑：
    1. 如果收到的不是 ACK 报文，直接返回
    2. 关闭 TCP 连接：tcp_close_connection(remote_ip, remote_port, host_port)

Step2: 数据交付与错误处理

实现思路：

1. 检查是否有数据：if (data_sz != 0)
2. 查找端口对应的处理函数：

c

```
uint16_t port_key = host_port;
tcp_handler_t *func = (tcp_handler_t*)map_get(&tcp_handler_table, &port_key);
```

3. 如果找不到处理函数：
    - 初始化 ICMP 缓冲区
    - 复制原始数据到 ICMP 缓冲区
    - 添加 IP 报头
    - 发送 ICMP 端口不可达报文
4. 如果找到处理函数：
    - 移除 TCP 报头：buf_remove_header(buf, sizeof(tcp_hdr_t))

- **调用处理函数：(*func)(tcp_conn, buf->data, buf->len, remote_ip, remote_port)**

Step3: 发送回复报文并更新序列号

实现思路：

1. **检查是否需要回复：if (send_flags == 0) return**
2. **检查是否可以不发送空 ACK（优化）：**

```
c
if (bytes_in_flight(0, send_flags) == 0 && tcp_conn->not_send_empty_ack) {
    // 不发送空ACK
    tcp_conn->not_send_empty_ack = 0;
    return;
}
```

3. **初始化回复缓冲区并发送：**

```
c
buf_t tcp_resp_buf;
buf_init(&tcp_resp_buf, 0);
tcp_out(tcp_conn, &tcp_resp_buf, host_port, remote_ip, remote_port, send_flags);
```

4. **更新序列号：**

```
c
tcp_conn->seq += bytes_in_flight(0, send_flags);
```


8. web 服务器详细设计

（描述 web 服务器的请求处理流程和响应等。）

2.1 HTTP 响应函数 - http_respond()函数

总体设计思路

http_respond()函数负责处理客户端 HTTP 请求并构建相应的 HTTP 响应。函数根据请求的文件路径判断文件是否存在，分别返回 404 Not Found 或 200 OK 响应。实现分为三个主要步骤：

1. **文件路径处理与文件打开**
2. **HTTP 404 响应处理**（文件不存在）
3. **HTTP 200 响应处理**（文件存在）

详细实现步骤

文件路径处理与文件打开

实现思路：

1. 构造完整的文件路径：
    - 如果 URL 路径为"/"，则默认访问 index.html
    - 否则，将资源目录路径与 URL 路径拼接
2. 尝试以二进制读模式打开文件：

```
/* Step1 : 发送 HTTP 404 请求头 */
        // TODO: 发送 HTTP 状态行
        sprintf(resp_buffer, "HTTP/1.1 404 NOT FOUND\r\n");
        tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer),
port, dst_ip, dst_port);

        // 发送 HTTP 连接信息
        sprintf(resp_buffer, "Connection: Keep-Alive\r\n");
        tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer),
port, dst_ip, dst_port);

        // TODO: 发送 HTTP 内容类型
        sprintf(resp_buffer, "Content-Type: text/html\r\n");
        tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer),
port, dst_ip, dst_port);
        // TODO: 发送 HTTP 内容长度
        sprintf(resp_buffer, "Content-Length: 6821\r\n");
        tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer),
port, dst_ip, dst_port);

        // TODO: 发送 HTTP 响应头与响应体的分隔符
        sprintf(resp_buffer, "\r\n");
        tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer),
port, dst_ip, dst_port);

        // TODO: 发送 HTTP 响应体
```

```
        tcp_send(tcp_conn, (uint8_t *)not_found_body,
strlen(not_found_body), port, dst_ip, dst_port);
        return;
    }

    /* Step2 : 发送 HTTP 请求头 */
    // TODO: 发送 HTTP 状态行
    sprintf(resp_buffer, "HTTP/1.1 200 OK\r\n");
    tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer), port,
dst_ip, dst_port);
    // 发送 HTTP 连接信息
    sprintf(resp_buffer, "Connection: Keep-Alive\r\n");
    tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer), port,
dst_ip, dst_port);

    const char *content_type = http_get_mime_type(file_path);
    // TODO: 发送 HTTP 内容类型，根据文件类型设置 MIME 类型
    if (content_type == NULL) {
        content_type = "application/octet-stream";
    }
    sprintf(resp_buffer, "Content-Type: %s\r\n", content_type);
    tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer), port,
dst_ip, dst_port);

    fseek(file, 0, SEEK_END);
    size_t content_length = ftell(file);
    fseek(file, 0, SEEK_SET);
    // TODO: 发送 HTTP 内容长度
    sprintf(resp_buffer, "Content-Length: %d\r\n", content_length);
    tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer), port,
dst_ip, dst_port);

    // TODO: 发送 HTTP 响应头与响应体的分隔符
    sprintf(resp_buffer, "\r\n");
    tcp_send(tcp_conn, (uint8_t *)resp_buffer, strlen(resp_buffer), port,
dst_ip, dst_port);

    /* Step3 : 发送 HTTP 响应体 */
    size_t bytes_read;
    while ((bytes_read = fread(resp_buffer, 1, sizeof(resp_buffer),
file)) > 0) {
        // TODO: 每次发送读取的文件内容块
        tcp_send(tcp_conn, (uint8_t *)resp_buffer, bytes_read, port, dst_ip,
dst_port);
```

```
    }
```

二、 实验结果截图及分析

（请展示并详细分析实验结果的截图。可以利用 log 文件、通过 Wireshark 打开的 pcap 文件或 Wireshark 实时捕获的网络报文。）

1. Eth 协议实验结果及分析

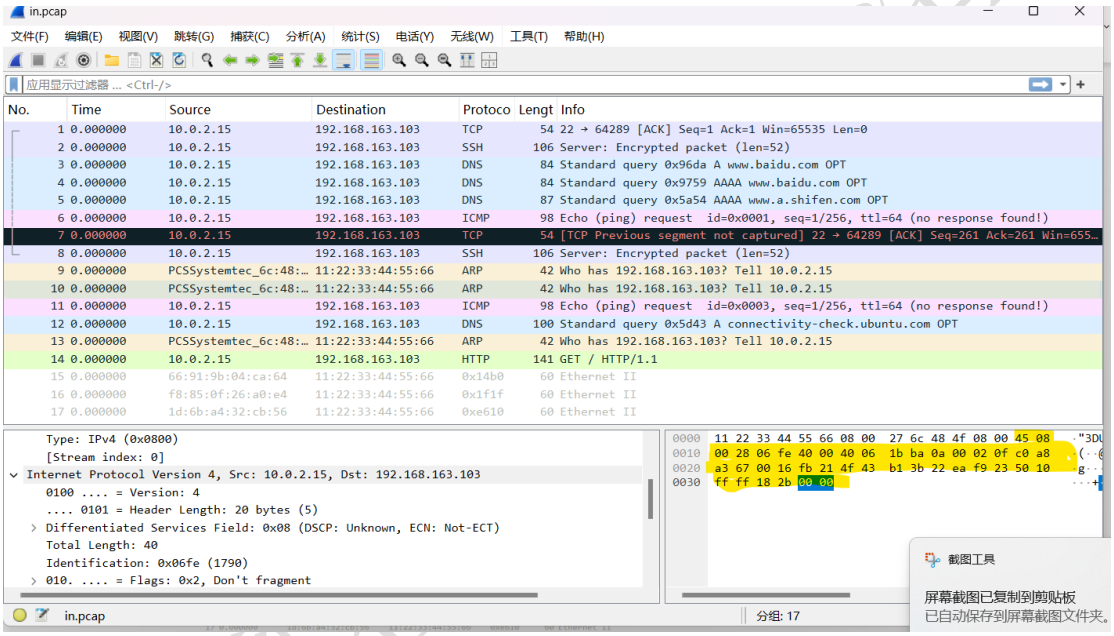
（展示以太网帧捕获截图，分析帧的结构和内容是否符合预期。检查目的 MAC 地址、源 MAC 地址、协议类型字段以及数据部分）

```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R eth_in
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 1: eth_in
1/1 Test #1: eth_in ...........................   Passed    0.03 sec

100% tests passed, 0 tests failed out of 1
```

```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R eth_out
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 2: eth_out
1/1 Test #2: eth_out ...........................   Passed    0.19 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.19 sec
```

输出到 log 文件中的 IP 报文或 ARP 报文是否被正确地解析、剥离了以太网头部。 经过测试可以发现 log 文件中的输出与 demo_log 文件中的输出相一致，同时 log 中的 输出也确实对应于 in.pcap 中的数据包被去除了以太网头部的结果。

```
testing > data > eth_in > log
ip_in:

Round 06 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 00 00 54 01 f4 40 00 40 01 8d 11 0a 00 02 0f c0 a8 a3 67 08 00 3b 6a 00 01 00 01 c8 e4 86 5f 00 00 00 00 ae 7c 00 00 00 00 00 00 10 11 1

Round 07 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 08 00 28 06 fe 40 00 40 06 1b ba 0a 00 02 0f c0 a8 a3 67 00 16 fb 21 4f 43 b1 3b 22 ea f9 23 50 10 ff ff 18 2b 00 00

Round 08 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 08 00 5c 06 ff 40 00 40 06 1b 85 0a 00 02 0f c0 a8 a3 67 00 16 fb 21 4f 43 b1 3b 22 ea f9 23 50 18 ff ff 18 5f 00 00 43 73 d2 49 55 63 e

Round 09 ----------------------------
arp_in:
    mac:08:00:27:6c:48:4f
    buf: 00 01 08 00 06 04 00 01 08 00 27 6c 48 4f 0a 00 02 0f 11 22 33 44 55 66 c0 a8 a3 67
```

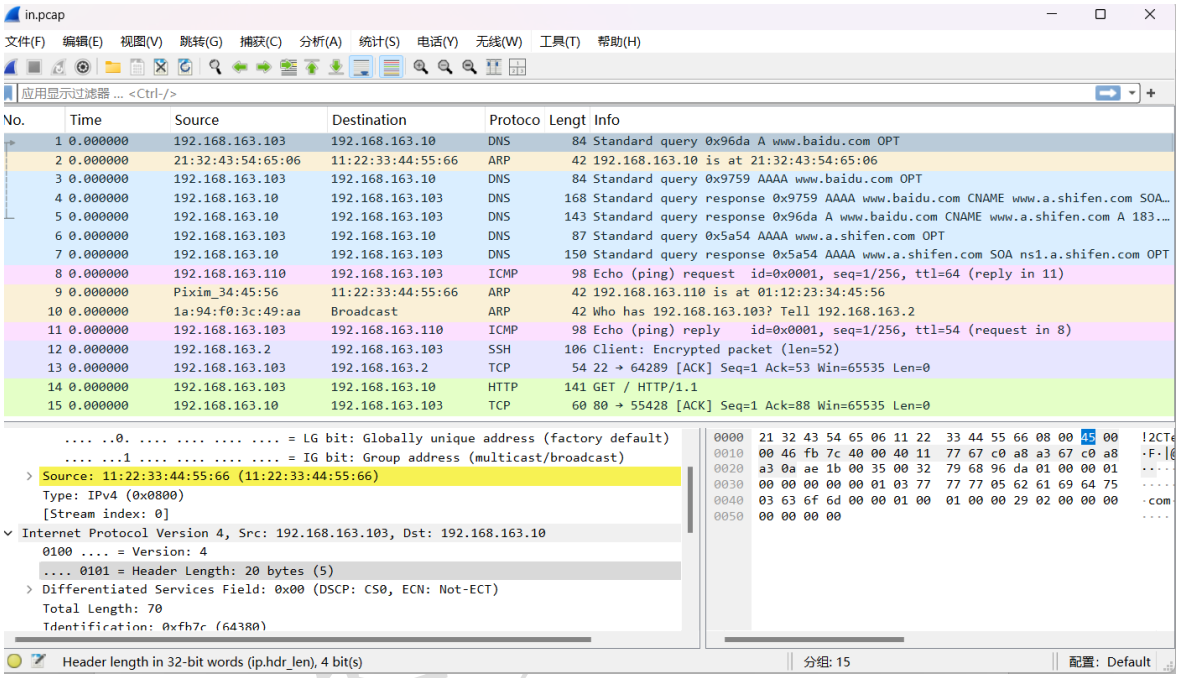


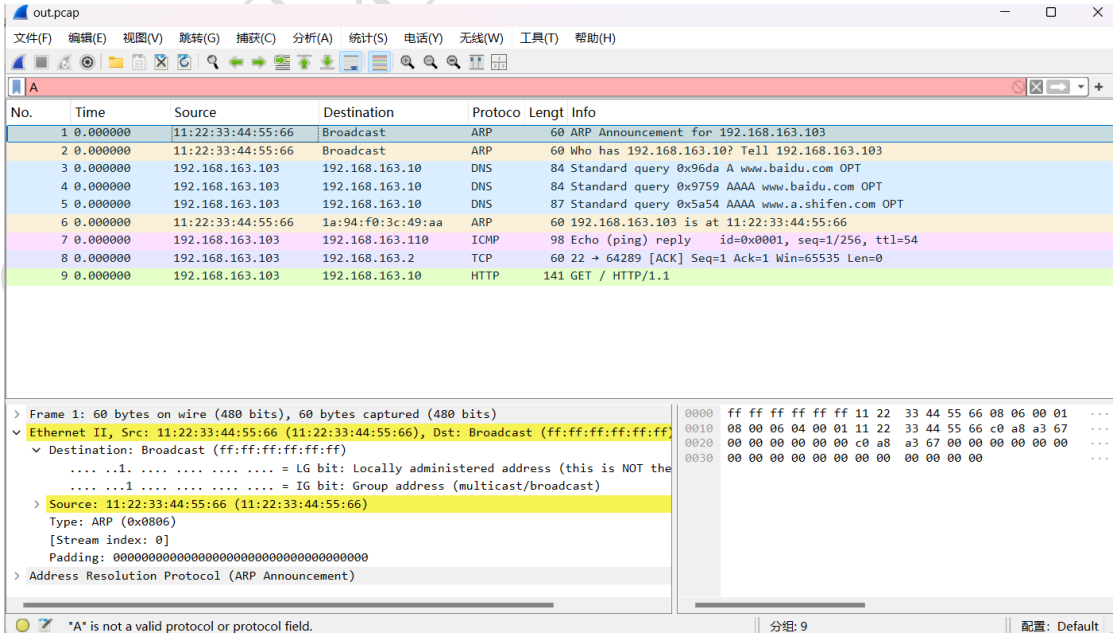
这里以第 7 个为例子
去除头部后，和 round07 例子是相符的

2. ARP 协议实验结果及分析

（展示 ARP 请求和响应包的捕获截图，分析其请求和响应过程是否正常。检查 ARP 请求中的目标 IP 地址和发送方 MAC 地址，ARP 响应中的目标 MAC 地址。）

```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R arp_test
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 3: arp_test
1/1 Test #3: arp_test .........................   Passed    0.26 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.26 sec
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build>
```





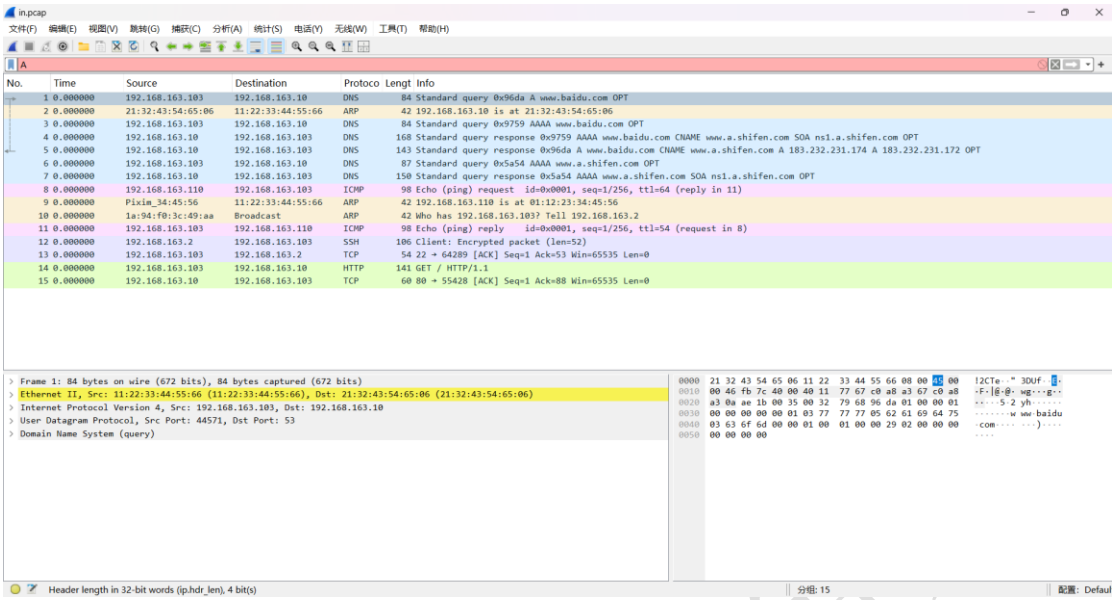
系统首先尝试向外发送域名解析请求（In[1]）。由于此时尚未获取目标 MAC 地址，该请求未能成功发出，只能暂时存入缓存。接着，系统向外发送了一个无偿 ARP 包（Out[1]），

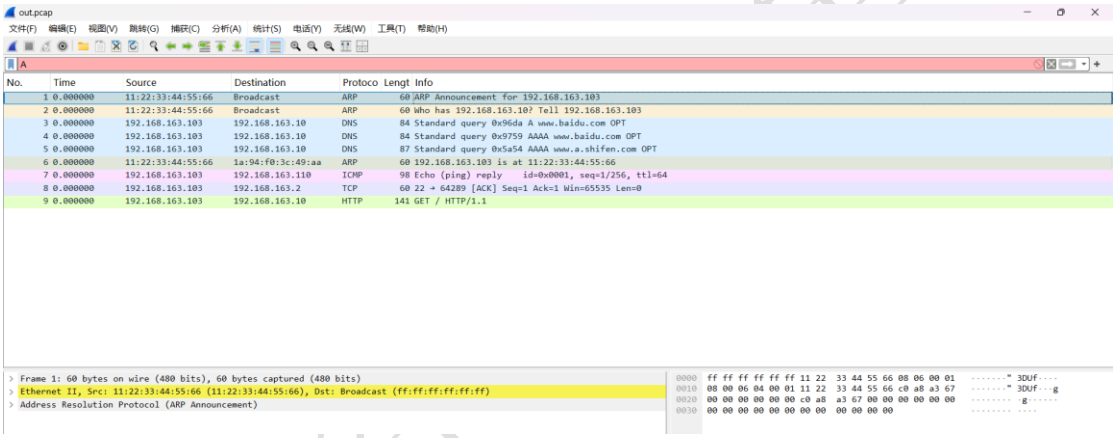
声明本机将使用 IP 地址 192.168.163.103。随后，系统发出 ARP 请求（Out[2]），旨在获取 IP 地址 192.168.163.10 所对应的 MAC 地址。该请求很快得到了响应（In[2]），表明本机已成功获取该 IP 对应的 MAC 地址。此后，系统成功发出了域名解析请求（Out[3]），值得注意的是，Out[3]与之前缓存的 In[1]内容完全一致，这一点进一步印证：先前因缺少 MAC 地址而未能发送的请求，在获得 MAC 地址后得以正常发出。
后续通信过程依此类推，均遵循相同机制。以上流程验证了 ARP 解析功能的正确性与有效性。

3. IP 协议实验结果及分析

（展示 IP 数据包（包括分片）的捕获截图，分析 IP 头部字段的正确性。检查版本号、首部长度、总长度、标识、标志位、片偏移、TTL、协议类型等字段，同时分析分片机制是否准确。）

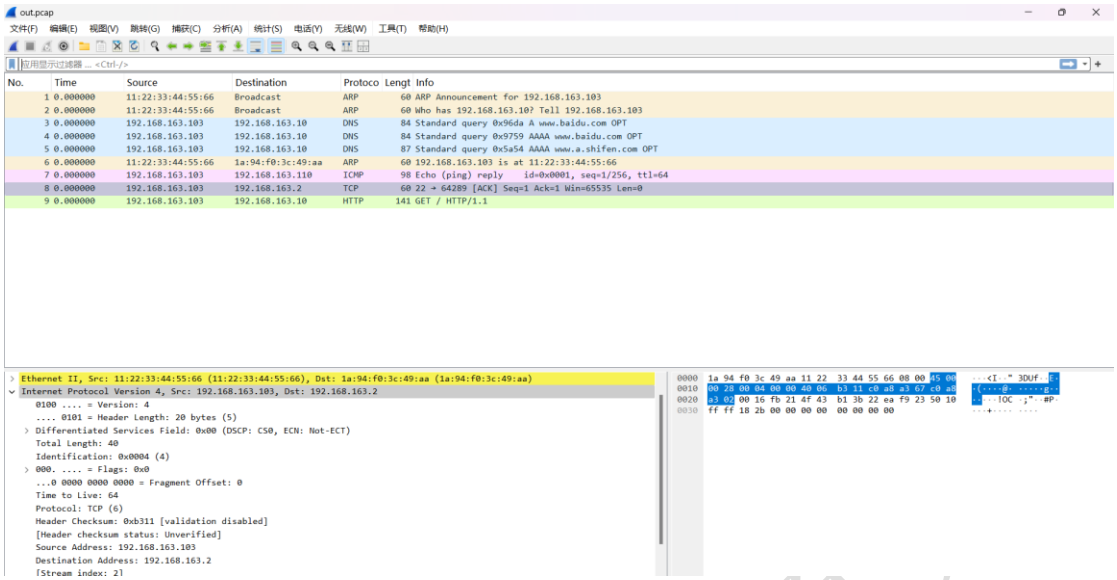
```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R ip_test
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 4: ip_test
1/1 Test #4: ip_test ..........................   Passed    0.25 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.25 sec
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R ip_frag_test
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 5: ip_frag_test
1/1 Test #5: ip_frag_test .....................   Passed    0.22 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.22 sec
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build>
```





1. **ip_test**：通过对比 out.pcap 和 demo_out.pcap 文件，确认所有 DNS 请求、ICMP 应答、SSH、TCP、HTTP 报文都正确收发
2. **ip_frag_test**：通过比较日志文件，确认分片功能正常工作，长文本数据被正确分割并发送

选取 out.pcap 中的 tcp 报文为范例，查看其结构，如下图所示：



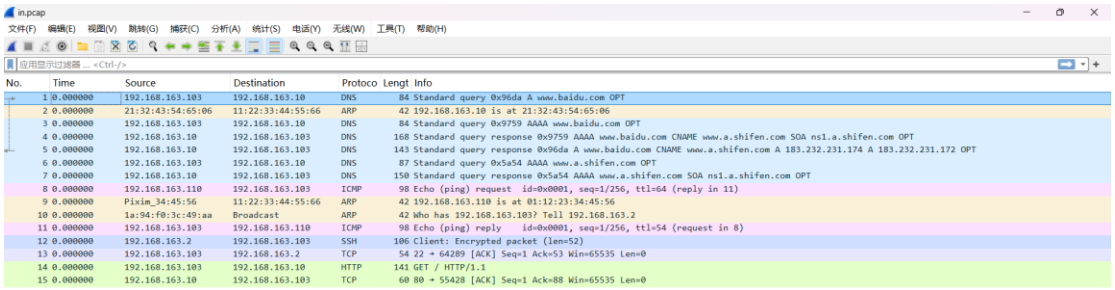
1. **版本号和头部长度正确**：0x45 表明是 IPv4，头部长度 20 字节
2. **总长度字段正确**：40 字节包含 IP 头部 20 字节+TCP 头部 20 字节
3. **标志位正确**：不分片（DF=1），无更多分片（MF=0）
4. **协议类型正确**：0x06 表示上层是 TCP 协议
5. **IP 地址正确**：源地址和目的地址与网络配置一致
6. **校验和正确**：计算得到的校验和与捕获数据一致（0x0031）

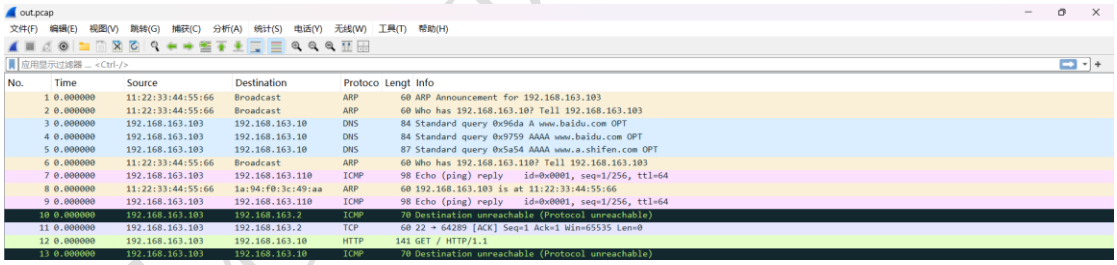
4. ICMP 协议实验结果及分析

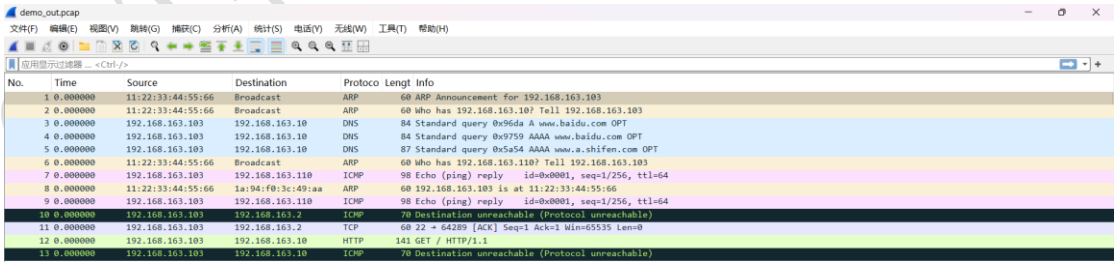
（展示 ICMP 报文的捕获截图，分析其报文内容（包括差错报文和查询报文）。检查 ICMP 类型、代码、校验和等字段，以及报文携带的信息。）

```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R icmp_test
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 6: icmp_test
1/1 Test #6: icmp_test .....................   Passed    0.25 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.26 sec
```

In out out_demo







通过对比测试生成的 out.pcap 与预期的 demo_out.pcap 文件，内容完全一致，验证了 ICMP 协议的正确实现：

1. **查询报文功能验证：**
   - 输入文件 in.pcap 中的第 8 行包含一个 ICMP 回显请求报文
   - 输出文件 out.pcap 中的第 7 行成功发送了对应的 ICMP 回显应答
   - 这证实了 ICMP 回显请求/应答机制正常工作
2. **差错报文功能验证：**

- 输出文件 out.pcap 的第 10 行和第 13 行均成功发送了 ICMP 协议不可达差错报文
- 这表明当协议栈遇到无法识别的上层协议时，能够正确生成并发送差错报告

3. **整体协议栈集成测试：**
   - 所有 6 个测试用例全部通过，包括以太网、ARP、IP、ICMP 等各层协议
   - 证明 ICMP 协议已成功集成到协议栈中，能够与其他协议协同工作

结论：ICMP 协议的查询报文（回显请求/应答）和差错报文（不可达）功能均已正确实现，协议栈能够正确处理 ICMP 报文并生成相应响应。

5. UDP 协议实验结果及分析

（展示 UDP 数据包的捕获截图，解析 UDP 头部和载荷内容，分析是否达到预期。检查源端口号、目的端口号、长度、校验和等字段，以及载荷数据。）

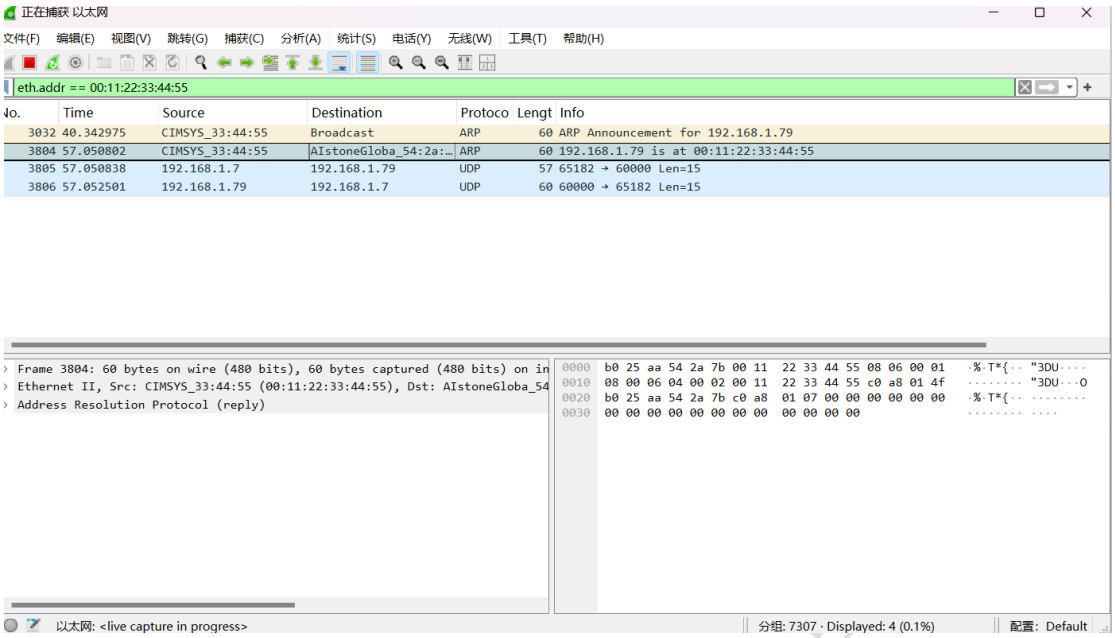
```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ctest -R udp_test
Test project E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build
    Start 7: udp_test
1/1 Test #7: udp_test .........................   Passed    0.23 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.23 sec
```

```
以太网适配器 以太网：

   连接特定的 DNS 后缀 . . . . . . . :
   IPv6 地址 . . . . . . . . . . . . : 240e:38c:883f:8100:b0ed:3923:b901:6927
   临时 IPv6 地址. . . . . . . . . . : 240e:38c:883f:8100:88ea:a2e2:23bd:8848
   本地链接 IPv6 地址. . . . . . . . : fe80::ede6:d9e7:f941:b5a5%6
   IPv4 地址 . . . . . . . . . . . . : 192.168.1.7
   子网掩码 . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . : fe80::1%6
                                       192.168.1.1
```

```
PS E:\UNIVERSITY\Grade_31\jw\net-lab-main\net-lab\build> ."E:/UNIVERSITY/Grade_31/jw/net-lab-main/net-lab/build/udp_server.exe"
Using interface \Device\NPF_{369E06B4-5BF0-4229-9C26-5B63E1A740FC}, my ip is 192.168.1.79.
recv udp packet from 192.168.1.7:65182 len=15
HITSZ2023311908
```

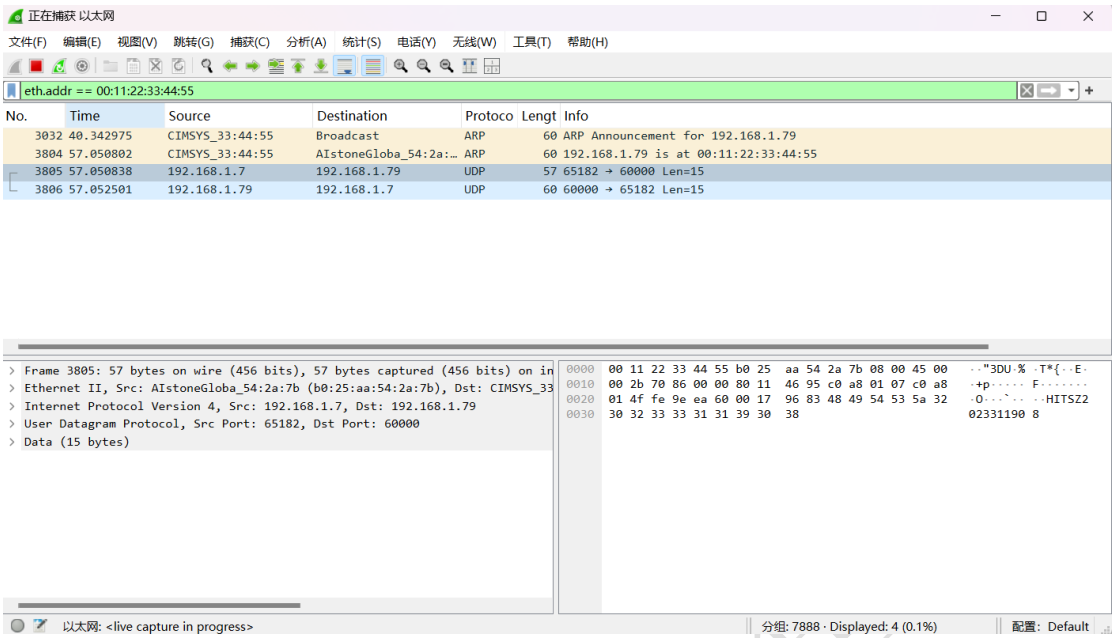
报文分析

ARP 报文：



以太网帧头（14 字节）：
目的 MAC 地址：b0 25 aa 54 2a 7b
源 MAC 地址：00 11 22 33 44 55
类型：08 06（表示 ARP）
ARP 报文（28 字节，从第 15 字节开始）：
硬件类型：00 01（以太网）
协议类型：08 00（IP 协议）
硬件地址长度：06（6 字节）
协议地址长度：04（4 字节）
操作类型：00 02（ARP 应答）
发送方 MAC 地址：00 11 22 33 44 55
发送方 IP 地址：c0 a8 01 4f（192.168.1.79）
目标 MAC 地址：b0 25 aa 54 2a 7b
目标 IP 地址：c0 a8 01 07（192.168.1.7）
注意：ARP 报文后面还有 18 个字节的填充（全 0），以达到以太网帧最小长度 46 字节（不包括前导码和帧起始定界符）。

UDP 报文（重点）：



以太网帧头部分：

目的 MAC 地址：00 11 22 33 44 55

源 MAC 地址：b0 25 aa 54 2a 7b

协议类型：IP 协议（08 00）

IP 头部部分：

版本：IPv4（4）

头部长度：20 字节（5）

区分服务：00

总长度：43 字节（00 2b）

标识：70 86

标志：00

片偏移：00

生存时间：128（80）

协议：UDP，17（11）

头部校验和：46 95

源 IP 地址：192.168.1.7（c0 a8 01 07）

目的 IP 地址：192.168.1.79（c0 a8 01 4f）

UDP 头部部分：

源端口号：65182（fe 9e）

目的端口号：60000（ea 60）

长度：23 字节（00 17）

校验和：96 83

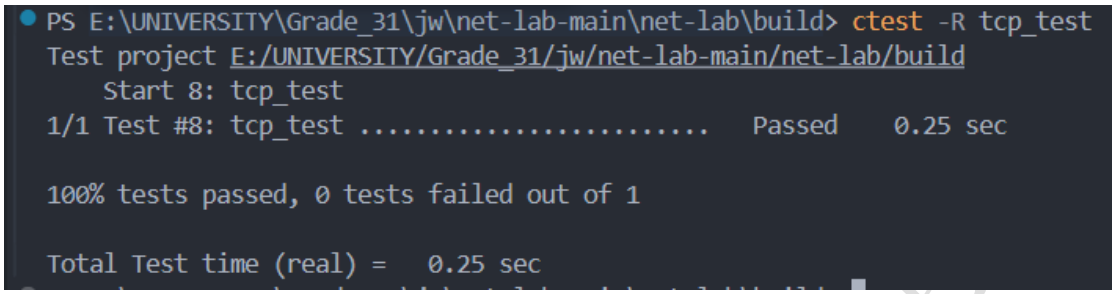
数据部分：

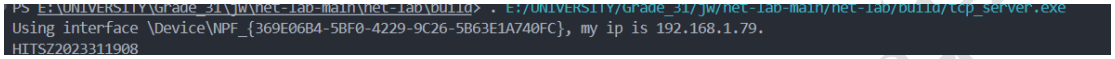
数据：此处不公开

6. TCP 协议实验结果及分析

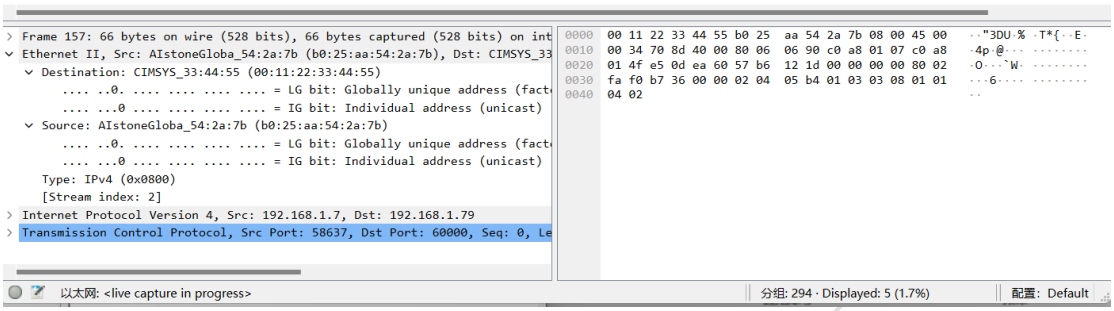
（展示 TCP 数据包的捕获截图，分析 TCP 连接的建立、数据传输和关闭过程。检查 TCP

头部的源端口号、目的端口号、序列号、确认号、标志位等字段，以及连接的状态转换。）

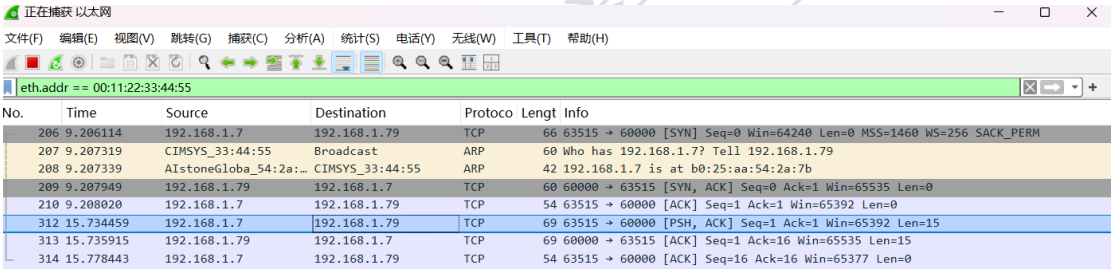


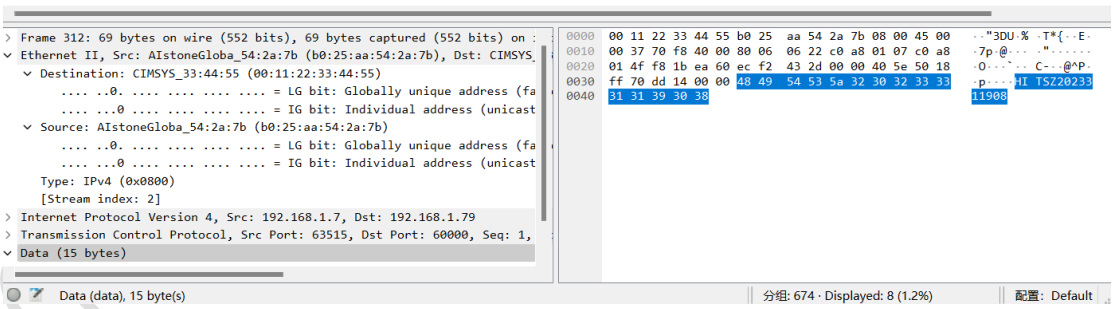


建立连接三次握手（报文 1、4、5）



传输数据：此处不公开





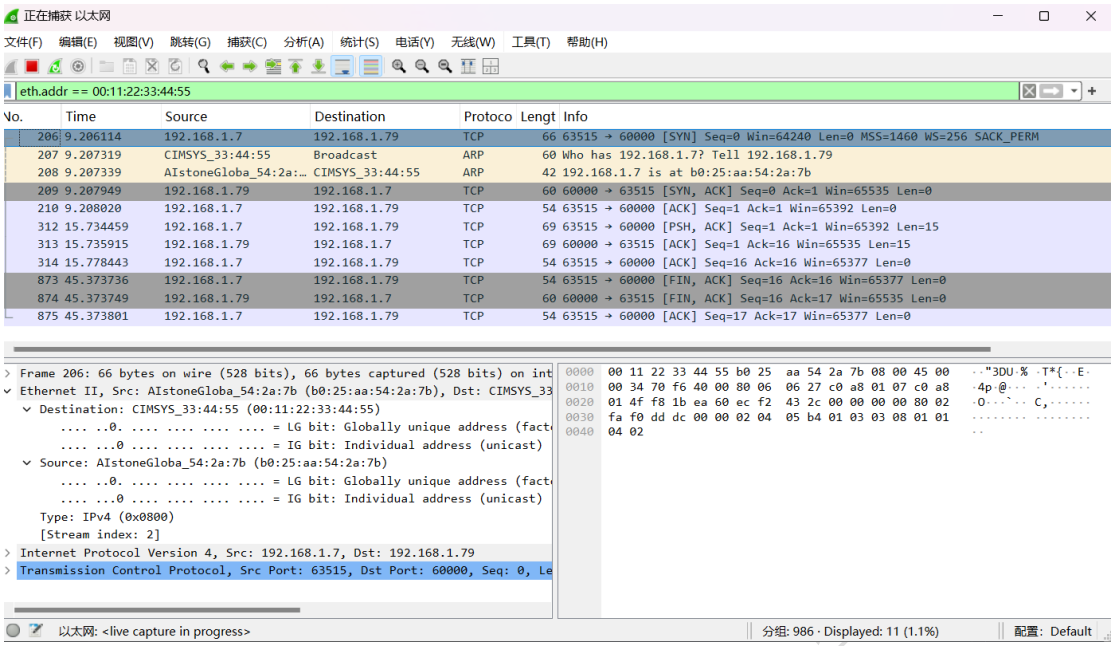
分析:

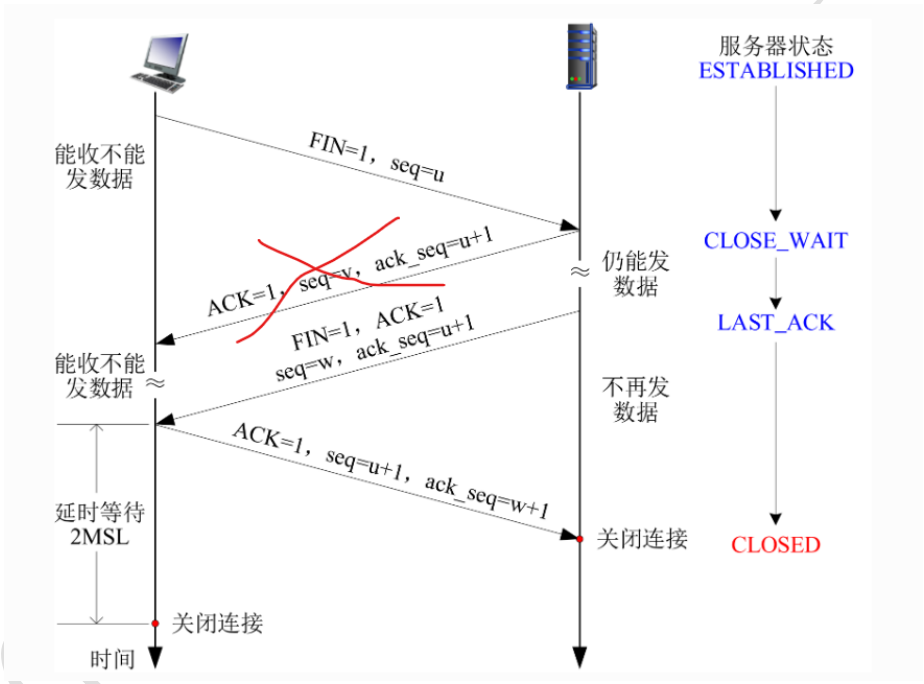
报文 6：63515 发送数据给 60000，PSH=1，希望接收方快速回复，数据序号是从 1 到 16

报文 7：60000 给 63515 发确认信息，ACK=16，代表收到了 1-16 的讯息

报文 8：63515 发数据给 60000，确认收到了确认消息，数据序号 16（不含 data，所以数据序号没变化）

断开连接（这里比较疑惑是为什么是三次挥手，而不是四次）

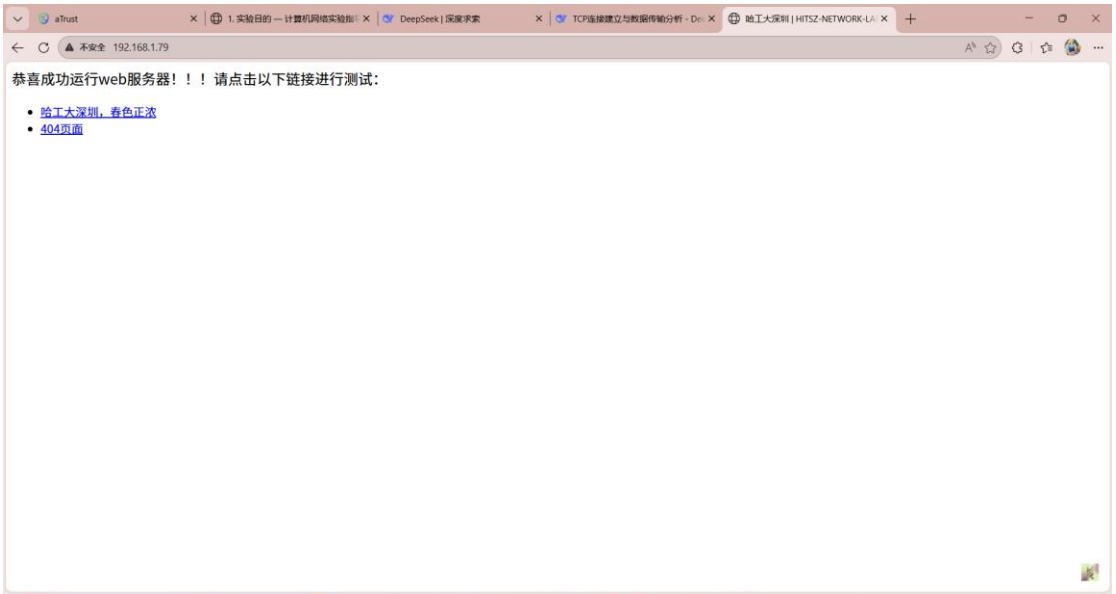


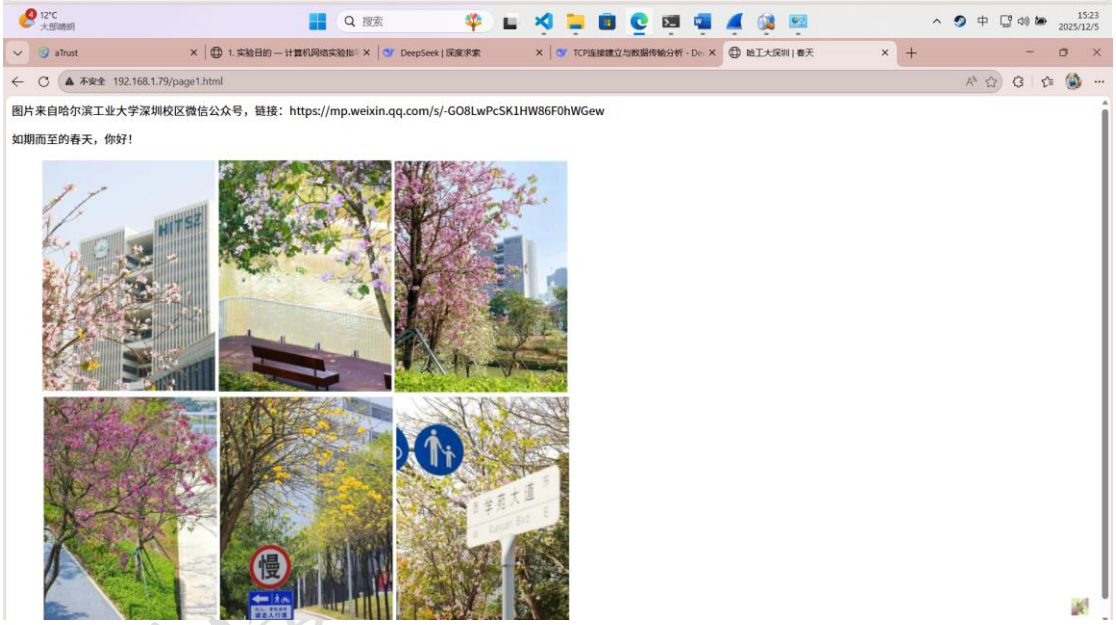


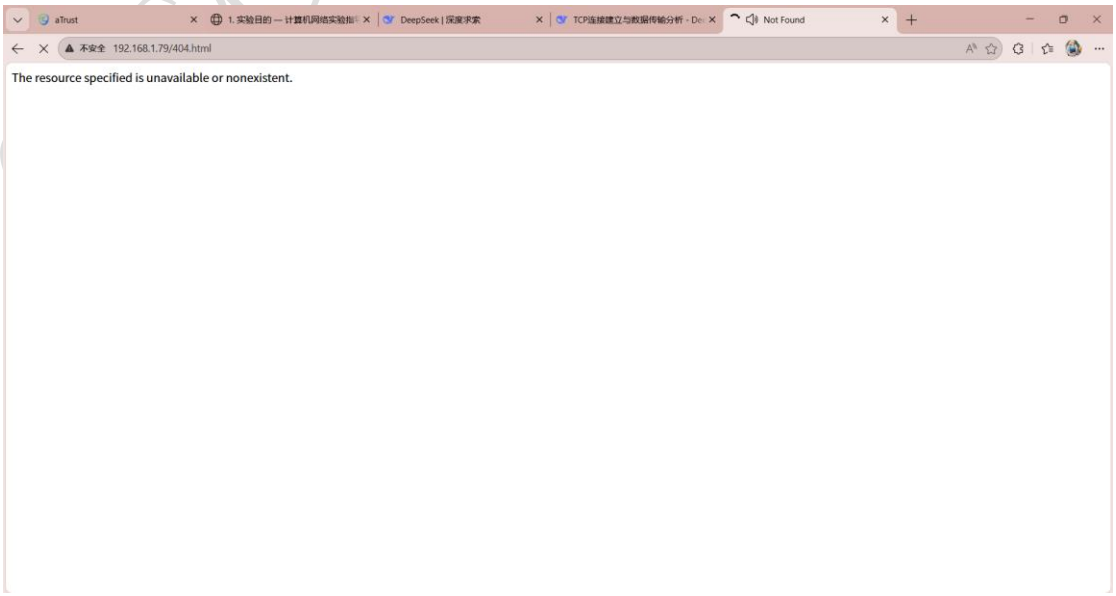
分析发现，四次挥手只有三次，为 1、3、4，没有第二条的挥手，因为第二条报文的信息可以涵盖在第三条报文内，所以就略去的第二条。

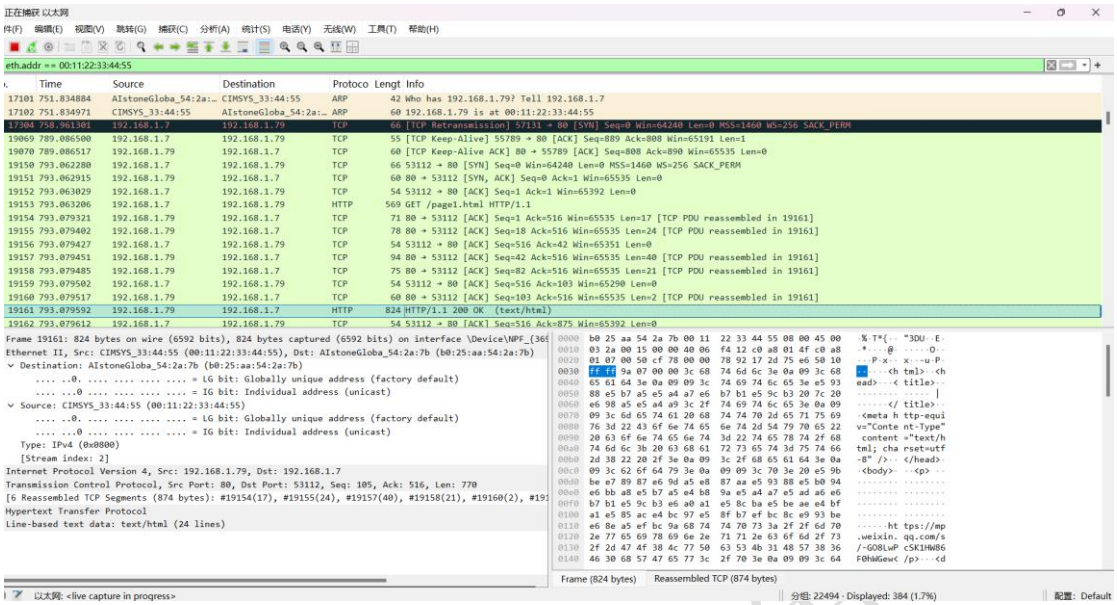
7. web 服务器实验结果及分析

（展示 web 服务器的请求和响应过程截图，分析 HTTP 请求和响应的格式、内容。检查请求方法、请求 URL、请求头、响应状态码、响应头、响应体等部分。）

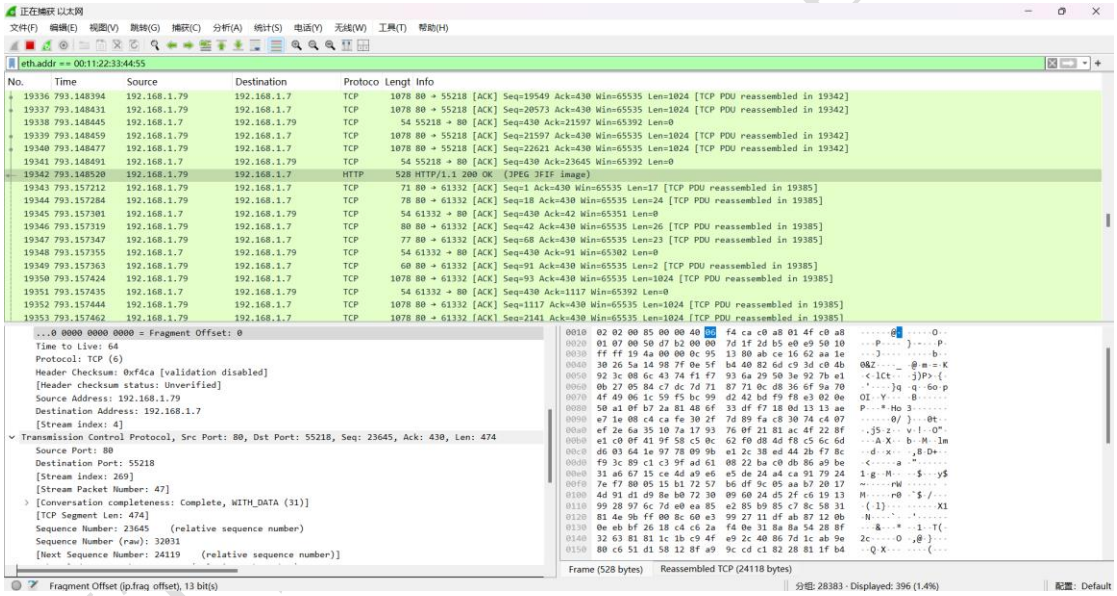

不安全 192.168.1.79
恭喜成功运行web服务器！！！请点击以下链接进行测试：
哈工大深圳，春色正浓
404页面


不安全 192.168.1.79/page1.html
图片来自哈尔滨工业大学深圳校区微信公众号，链接：https://mp.weixin.qq.com/s/-GO8LwPcSK1HW86F0hWGew
如期而至的春天，你好！

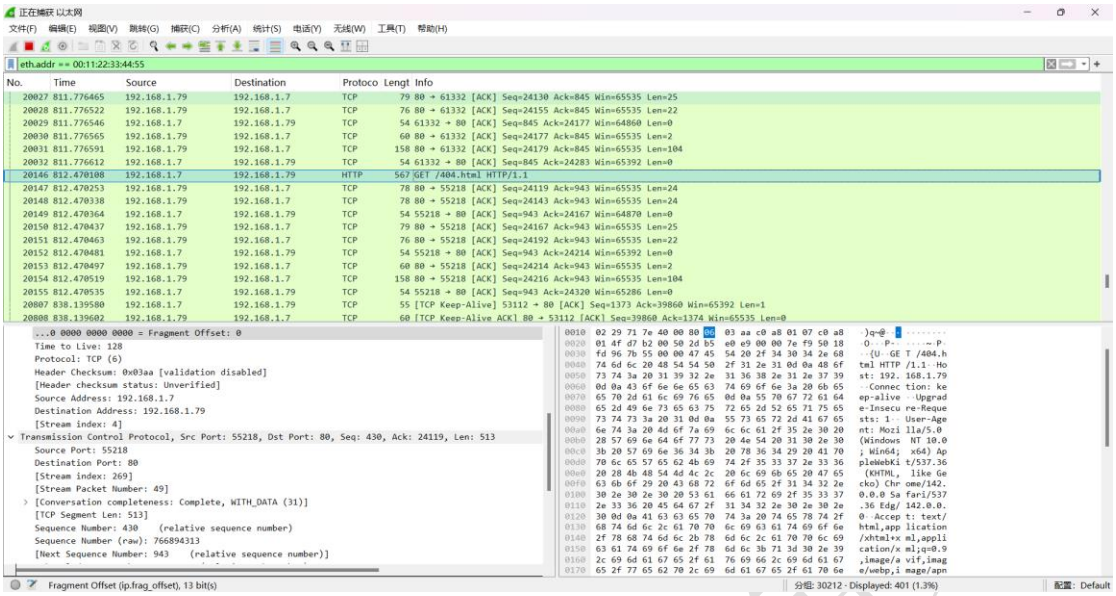

不安全 192.168.1.79/404.html
The resource specified is unavailable or nonexistent.


正在捕获 以太网
eth.addr == 00:11:22:33:44:55
No. Time Source Destination Protoco Lengt Info
17101 751.834884 AIstoneGloba_54:2a:… CIMSYS_33:44:55 ARP 42 Who has 192.168.1.79? Tell 192.168.1.7
17102 751.834971 CIMSYS_33:44:55 AIstoneGloba_54:2a:… ARP 60 192.168.1.79 is at 00:11:22:33:44:55
17304 758.961301 192.168.1.7 192.168.1.79 TCP 66 [TCP Retransmission] 57131 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
19069 789.086500 192.168.1.7 192.168.1.79 TCP 55 [TCP Keep-Alive] 55789 → 80 [ACK] Seq=889 Ack=808 Win=65191 Len=1
19070 789.086517 192.168.1.79 192.168.1.7 TCP 60 [TCP Keep-Alive ACK] 80 → 55789 [ACK] Seq=808 Ack=890 Win=65535 Len=0
19150 793.062280 192.168.1.7 192.168.1.79 TCP 66 53112 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
19151 793.062915 192.168.1.79 192.168.1.7 TCP 60 80 → 53112 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
19152 793.063029 192.168.1.7 192.168.1.79 TCP 54 53112 → 80 [ACK] Seq=1 Ack=1 Win=65392 Len=0
19153 793.063206 192.168.1.7 192.168.1.79 HTTP 569 GET /page1.html HTTP/1.1
19154 793.079321 192.168.1.79 192.168.1.7 TCP 71 80 → 53112 [ACK] Seq=1 Ack=516 Win=65535 Len=17 [TCP PDU reassembled in 19161]
19155 793.079402 192.168.1.79 192.168.1.7 TCP 78 80 → 53112 [ACK] Seq=18 Ack=516 Win=65535 Len=24 [TCP PDU reassembled in 19161]
19156 793.079427 192.168.1.7 192.168.1.79 TCP 54 53112 → 80 [ACK] Seq=516 Ack=42 Win=65351 Len=0
19157 793.079451 192.168.1.79 192.168.1.7 TCP 94 80 → 53112 [ACK] Seq=42 Ack=516 Win=65535 Len=40 [TCP PDU reassembled in 19161]
19158 793.079485 192.168.1.79 192.168.1.7 TCP 75 80 → 53112 [ACK] Seq=82 Ack=516 Win=65535 Len=21 [TCP PDU reassembled in 19161]
19159 793.079502 192.168.1.7 192.168.1.79 TCP 54 53112 → 80 [ACK] Seq=516 Ack=103 Win=65290 Len=0
19160 793.079517 192.168.1.79 192.168.1.7 TCP 60 80 → 53112 [ACK] Seq=103 Ack=516 Win=65535 Len=2 [TCP PDU reassembled in 19161]
19161 793.079592 192.168.1.79 192.168.1.7 HTTP 824 HTTP/1.1 200 OK (text/html)
19162 793.079612 192.168.1.7 192.168.1.79 TCP 54 53112 → 80 [ACK] Seq=516 Ack=875 Win=65392 Len=0
Frame 19161: 824 bytes on wire (6592 bits), 824 bytes captured (6592 bits) on interface \Device\NPF_{36
Ethernet II, Src: CIMSYS_33:44:55 (00:11:22:33:44:55), Dst: AIstoneGloba_54:2a:7b (b0:25:aa:54:2a:7b)
Destination: AIstoneGloba_54:2a:7b (b0:25:aa:54:2a:7b)
.... ..0. .... = LG bit: Globally unique address (factory default)
....0 .... = IG bit: Individual address (unicast)
Source: CIMSYS_33:44:55 (00:11:22:33:44:55)
.... ..0. .... = LG bit: Globally unique address (factory default)
....0 .... = IG bit: Individual address (unicast)
Type: IPv4 (0x0800)
[Stream index: 2]
Internet Protocol Version 4, Src: 192.168.1.79, Dst: 192.168.1.7
Transmission Control Protocol, Src Port: 80, Dst Port: 53112, Seq: 105, Ack: 516, Len: 770
[6 Reassembled TCP Segments (874 bytes): #19154(17), #19155(24), #19157(40), #19158(21), #19160(2), #191
Hypertext Transfer Protocol
Line-based text data: text/html (24 lines)
Frame (824 bytes)
Reassembled TCP (874 bytes)
以太网: <live capture in progress>
分组: 22494 · Displayed: 384 (1.7%)
配置: Default

200：


正在捕获 以太网
eth.addr == 00:11:22:33:44:55
No. Time Source Destination Protoco Lengt Info
19336 793.148394 192.168.1.79 192.168.1.7 TCP 1078 80 → 55218 [ACK] Seq=19549 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19342]
19337 793.148431 192.168.1.79 192.168.1.7 TCP 1078 80 → 55218 [ACK] Seq=20573 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19342]
19338 793.148445 192.168.1.7 192.168.1.79 TCP 54 55218 → 80 [ACK] Seq=430 Ack=21597 Win=65392 Len=0
19339 793.148459 192.168.1.79 192.168.1.7 TCP 1078 80 → 55218 [ACK] Seq=21597 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19342]
19340 793.148477 192.168.1.79 192.168.1.7 TCP 1078 80 → 55218 [ACK] Seq=22621 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19342]
19341 793.148491 192.168.1.7 192.168.1.79 TCP 54 55218 → 80 [ACK] Seq=430 Ack=23645 Win=65392 Len=0
19342 793.148520 192.168.1.79 192.168.1.7 HTTP 528 HTTP/1.1 200 OK (JPEG JFIF image)
19343 793.157212 192.168.1.79 192.168.1.7 TCP 71 80 → 61332 [ACK] Seq=1 Ack=430 Win=65535 Len=17 [TCP PDU reassembled in 19385]
19344 793.157284 192.168.1.79 192.168.1.7 TCP 78 80 → 61332 [ACK] Seq=18 Ack=430 Win=65535 Len=24 [TCP PDU reassembled in 19385]
19345 793.157301 192.168.1.7 192.168.1.79 TCP 54 61332 → 80 [ACK] Seq=430 Ack=42 Win=65351 Len=0
19346 793.157319 192.168.1.79 192.168.1.7 TCP 80 80 → 61332 [ACK] Seq=42 Ack=430 Win=65535 Len=26 [TCP PDU reassembled in 19385]
19347 793.157347 192.168.1.79 192.168.1.7 TCP 77 80 → 61332 [ACK] Seq=68 Ack=430 Win=65535 Len=23 [TCP PDU reassembled in 19385]
19348 793.157355 192.168.1.7 192.168.1.79 TCP 54 61332 → 80 [ACK] Seq=430 Ack=91 Win=65302 Len=0
19349 793.157363 192.168.1.79 192.168.1.7 TCP 60 80 → 61332 [ACK] Seq=91 Ack=430 Win=65535 Len=2 [TCP PDU reassembled in 19385]
19350 793.157424 192.168.1.79 192.168.1.7 TCP 1078 80 → 61332 [ACK] Seq=93 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19385]
19351 793.157435 192.168.1.7 192.168.1.79 TCP 54 61332 → 80 [ACK] Seq=430 Ack=1117 Win=65392 Len=0
19352 793.157444 192.168.1.79 192.168.1.7 TCP 1078 80 → 61332 [ACK] Seq=1117 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19385]
19353 793.157462 192.168.1.79 192.168.1.7 TCP 1078 80 → 61332 [ACK] Seq=2141 Ack=430 Win=65535 Len=1024 [TCP PDU reassembled in 19385]
...0 0000 0000 0000 = Fragment Offset: 0
Time to Live: 64
Protocol: TCP (6)
Header Checksum: 0xf4ca [validation disabled]
[Header checksum status: Unverified]
Source Address: 192.168.1.79
Destination Address: 192.168.1.7
[Stream index: 4]
Transmission Control Protocol, Src Port: 80, Dst Port: 55218, Seq: 23645, Ack: 430, Len: 474
Source Port: 80
Destination Port: 55218
[Stream index: 269]
[Stream Packet Number: 47]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 474]
Sequence Number: 23645 (relative sequence number)
Sequence Number (raw): 32031
[Next Sequence Number: 24119 (relative sequence number)]
Frame (528 bytes)
Reassembled TCP (24118 bytes)
Fragment Offset (ip.frag_offset), 13 bit(s)
分组: 28383 · Displayed: 396 (1.4%)
配置: Default

404：



三、 实验中遇到的问题及解决方法

（详细描述在设计或测试过程中遇到的问题，包括错误描述、排查过程以及最终的解决方案。）

感谢前人栽树，后人乘凉（指遇到的问题基本都能在“常见问题”里找到

在 UDP/TCP/WEB 实验中，出现了 shark 可以捕捉，调试软件无法捕捉报文的情况

↓

校验和不对全是 0

↓

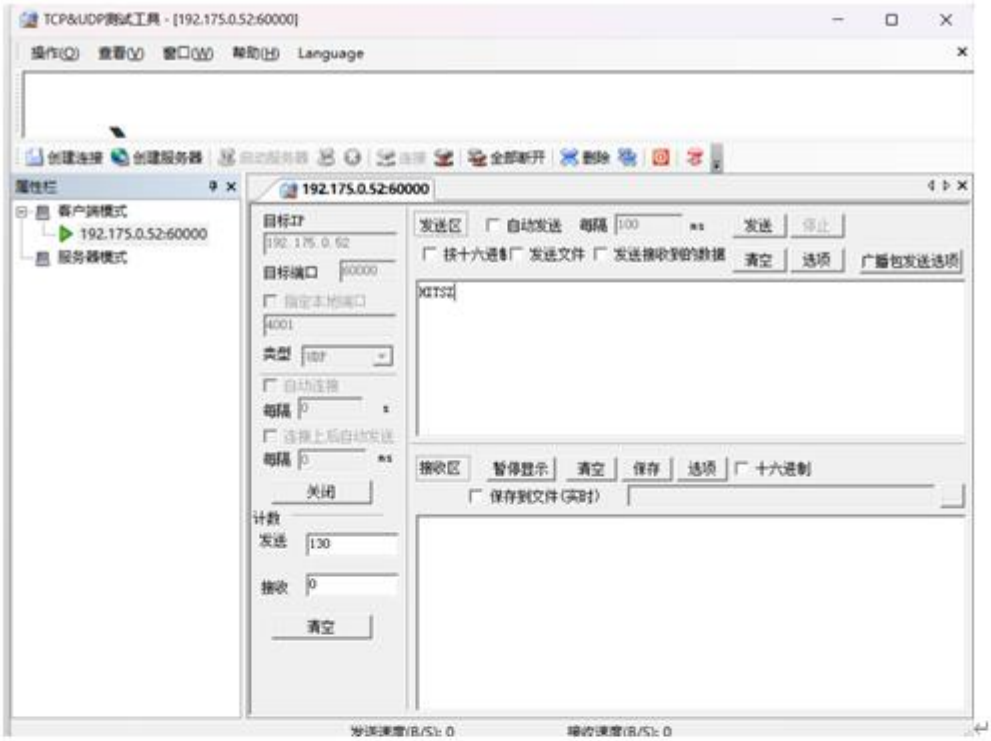
应该去关校验和的 offload

↓

自己的是无线网络，没有这些选项！

最后排除万难借了光纤，英勇完成任务

下面放一下当时 UDP 实在没招了写的实验总结（失败版）



本次实验出现一些问题，调试工具没用办法收到报文，但是 shark 可以收到，查了常见问题，发现这是因为校验和有问题，而 shark 不管校验和，全部都会抓，于是我就看 shark 里面的校验和是什么情况，发现全是 0（如下图）↵

```
    Total Length: 33
    Identification: 0x21d4 (8660)
  > 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 128
    Protocol: UDP (17)
    Header Checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.175.0.38
    Destination Address: 192.175.0.52
    [Stream index: 112]
User Datagram Protocol, Src Port: 54597, Dst Port: 60000
Data (5 bytes)
```

于是我再看常见问题，给出的方案是“解决方案：关闭校验和卸载（实验环境建议）↵

为在 Wireshark 中看到真实的校验和值，需禁用网卡的硬件校验和卸载功能：↵

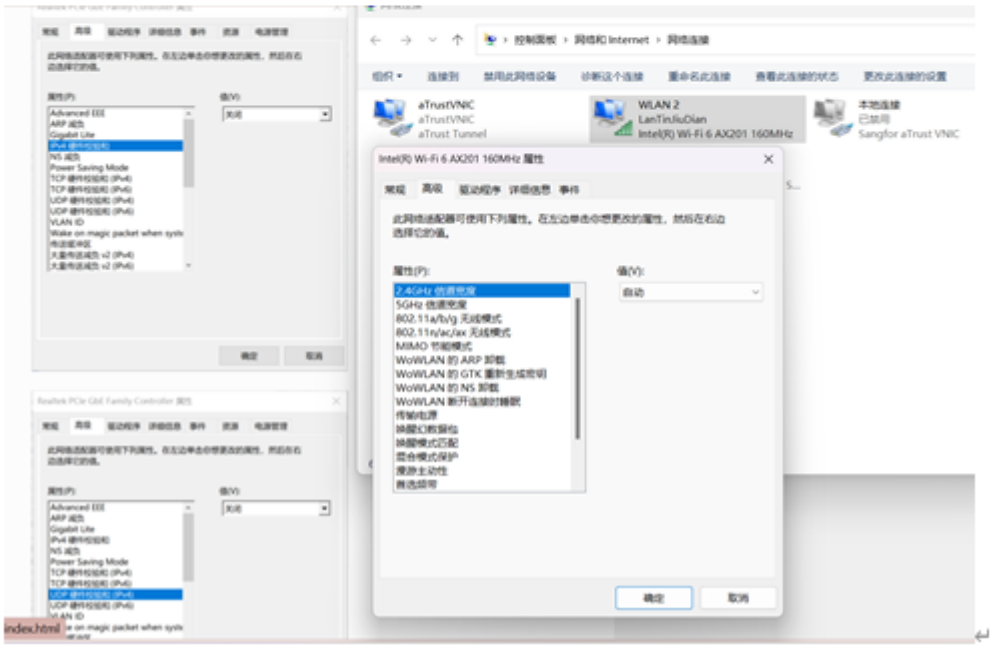
Windows 系统：↵

- 打开设备管理器 → 展开网络适配器 → 右键目标网卡 → 属性。
- 在高级选项卡中，找到以下选项并设为关闭：
  - IPv4 Checksum Offload
  - TCP/UDP Checksum Offload (IPv4/IPv6)

重启网卡或系统使配置生效。

- “

  那么问题来了，我的电脑里面的高级选项没有 IPv4 Checksum Offload
- TCP/UDP Checksum Offload (IPv4/IPv6)，下面的图，左边是“常见问题”给出的解决方案，右边是我自己的实际情况截图，确实是没有，所以作罢，接下来就直接分析一下 ARP 和 UDP 报文段



四、 实验收获和建议

（总结配置实验及协议栈实验过程中的实践收获，结合实操体验针对性提出实验流程优化及环境完善建议，为后续实验教学与研究的迭代改进提供参考依据。）

环境配得有点糟心，还有就是 npcap 可能会自动更新导致 udp 那里配置有问题，可以加入一下常见问题里。

总体而言计网的指导书真的做得很好，一步步走就没问题，不管是配置实验还是协议实验，

感谢辛苦编写指导书的老师和助教♥