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

实验报告

开课学期：2023 春季

课程名称：计算机网络

实验名称：协议栈之 IP、ICMP、UDP 协议实现

学生班级：5 班

学生学号：200110513

学生姓名：宗晴

评阅教师：

报告成绩：

实验与创新实践教育中心制

2023 年 3 月

一、 实验详细设计

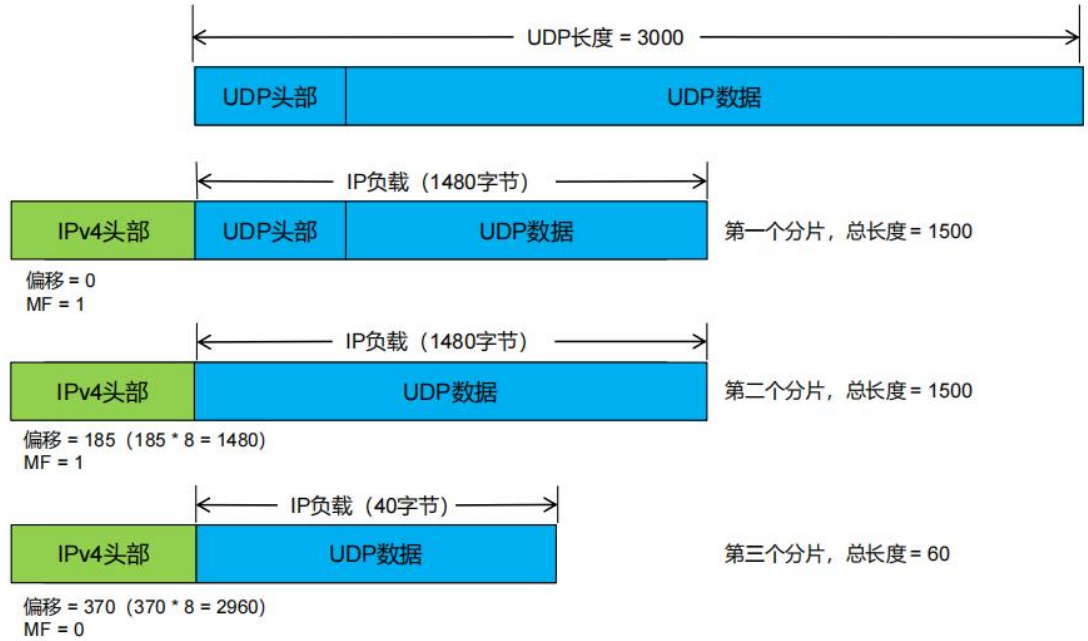
（注意不要完全照搬实验指导书上的内容，请根据你自己的设计方案来填写图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。）

1. IP 协议详细设计

数据链路层提供两个直连设备之间的通信功能，而一旦跨越不同的数据链路，就需要借助上层的网络层。IP 协议是 TCP/IP 协议网络层的协议，其主要作用是“实现终端节点之间的通信”，也叫“点对点通信”。

IP 数据包的格式如下图所示：

分片格式如下图所示：



本实验要求在完成协议栈之 ARP 协议的基础上，编写 IP 报文的发送、接收、IP 分片以及计算校验和函数，使其能够发送和接收 IP 数据报文，并且能通过实验评测系统的测试。

具体而言，需要补充完整 src/ip.c 文件中的 ip_in()函数、ip_out()函数和

ip_fragment_out()函数，以及 src/utils.c 文件中的 checksum16() 函数。

（1）IP 数据报输入处理，即实现 ip_in()函数

主要功能为进行合法性检测，对可能存在的填充字段进行处理。若是正确的数据包，则去掉报头并向上层传递，否则返回 ICMP 协议不可达信息。

算法实现如下图所示：

```
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 * @param src_mac 源mac地址
 */
void ip_in(buf_t *buf, uint8_t *src_mac)
{
    // TO-DO
    // Step1 : 如果数据包的长度小于IP头部长度，丢弃不处理。
    if(buf->len < sizeof(ip_hdr_t)) return;

    // Step2 : 接下来做报头检测，如果不符合这些要求，则丢弃不处理。
    ip_hdr_t* ip_head = (ip_hdr_t*)buf->data;
    // 检查IP头部的版本号是否为IPv4
    if(ip_head->version != IP_VERSION_4) return;
    // 检查总长度字段小于或等于收到的包的长度
    uint16_t total_len16 = swap16(ip_head->total_len16);
    if(total_len16 > buf->len) return;

    // Step3 : 先把IP头部的头部校验和字段用其他变量保存起来，
    // 接着将该头部校验和字段置0，然后调用checksum16函数来计算头部校验和，
    // 如果与IP头部的首部校验和字段不一致，丢弃不处理，
    // 如果一致，则再将该头部校验和字段恢复成原来的值。
    uint16_t old_checksum16 = ip_head->hdr_checksum16;
    ip_head->hdr_checksum16 = 0;
    uint16_t new_checksum16 = checksum16((uint16_t *)ip_head, sizeof(ip_hdr_t));
    if(new_checksum16 != old_checksum16) return;
    ip_head->hdr_checksum16 = old_checksum16;
```

首先需要判断数据包的长度是否小于 IP 头部长度，若小于，则丢弃不处理。然后进行报头检测，检查内容至少包括：IP 头部的版本号是否为 IPv4，总长度字段小于或等于收到的包的长度等，如果不符合这些要求，则丢弃不处理。然后把 IP 头部的头部校验和字段用其他变量保存起来，接着将该头部校验和字段置 0，然后调用 checksum16 函数来计算头部校验和，如果与 IP 头部的首部校验和字段不一致，则表明出错，丢弃不处理，如果一致，则再将该头部校验和字段恢复成原来的值。

注意此处需要先将校验和置零，然后再计算校验和进行比较，同时最后需要将校验和恢复成原来的值。

```
// Step4 : 对比目的IP地址是否为本机的IP地址，如果不是，则丢弃不处理。
if(memcmp(ip_head->dst_ip,net_if_ip,4) != 0) return;

// Step5 : 如果接收到的数据包的长度大于IP头部的总长度字段，则说明该数据包有填充字段，
// 可调用buf_remove_padding()函数去除填充字段。
if(buf->len > total_len16)
    buf_remove_padding(buf, buf->len - total_len16);

// Step6 : 调用buf_remove_header()函数去掉IP报头。
buf_remove_header(buf, sizeof(ip_hdr_t));

// Step7 : 调用net_in()函数向上层传递数据包。如果是不能识别的协议类型，
// 即调用icmp_unreachable()返回ICMP协议不可达信息。
if(net_in(buf, ip_head->protocol, ip_head->src_ip) == -1){
    buf_add_header(buf, sizeof(ip_hdr_t));
    memcpy(buf->data, ip_head, sizeof(ip_hdr_t));
    icmp_unreachable(buf, ip_head->src_ip, ICMP_CODE_PROTOCOL_UNREACH);
}
return;
```

然后，对比目的 IP 地址是否为本机的 IP 地址，如果不是，则丢弃不处理。如果接收到的数据包的长度大于 IP 头部的总长度字段，则说明该数据包有填充字段，可调用 buf_remove_padding()函数去除填充字段。接着，调用 buf_remove_header()函数去掉 IP 报头。最后，调用 net_in()函数向上层传递数据包。如果是不能识别的协议类型，则调用 icmp_unreachable()返回 ICMP 协议不可达信息。

（2）校验和算法，即实现 checksum16()函数

算法实现如下图所示：

```
 * @param buf 要计算的数据包
 * @param len 要计算的长度
 * @return uint16_t 校验和
 */
uint16_t checksum16(uint16_t *data, size_t len)
{
    // TO-DO
    // Step1 : 把data看成是每16个bit（即2个字节）组成一个数，相加
    uint32_t checksum = 0;
    for(size_t i = 0; i < len / 2; i ++ )
        checksum += (uint16_t)data[i];

    // Step2 : 如果最后还剩8个bit值，也要相加这个8bit值。
    if(len % 2 == 1) checksum += (uint8_t)data[len - 1];

    // Step3 : 判断相加后32bit结果值的高16位是否为0，如果不为0，
    // 则将高16位和低16位相加，依次循环，直至高16位为0为止。
    while(checksum >> 16 != 0) {
        uint16_t high16 = checksum >> 16;
        uint16_t low16 = (checksum << 16) >> 16;
        checksum = high16 + low16;
    }

    // Step4 : 将上述的和（低16位）取反，即得到校验和。
    return ~(uint16_t) checksum;
}
```

我们把 data 看成是每 16 个 bit（即 2 个字节）组成一个数，相加。如果最后还剩 8 个 bit 值，也要相加这个 8bit 值。若相加后结果超过 16 位，则最高位的进位也需要继续加在低 16 位上，依次循环，直至不超过 16 位。最后将上述的和取反，即得到待求的校验和。

此处需注意，考虑到 16 位加法的结果可能会超过 16 位，因此可用 32 位数来保存加法结果。

（3）IP 数据报输出处理，即实现 ip_out()函数

该函数的功能主要是进行一些合法性检测，根据数据长度进行合理的切片，然后将切片后的数据发送出去。

在此函数中，需要注意，我们首先要在函数外部定义一个静态的全局变量作为序列号，初始值为 0，在该函数的最后每次加一，这样可以保证每次获得的都是不同的逐次累加的序列号。

算法实现如下图所示：

```c
/**
 * @brief 处理一个要发送的ip数据包
 *
 * @param buf 要处理的包
 * @param ip 目标ip地址
 * @param protocol 上层协议
 */
static int id = 0;
void ip_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol)
{
    // TO-DO
    // Step1 : 首先检查从上层传递下来的数据报包长是否大于
    // IP协议最大负载包长（1500字节（MTU） 减去IP首部长度）。
    int max_data_len = (ETHERNET_MAX_TRANSPORT_UNIT - sizeof(ip_hdr_t)) / 8 * 8;
    int offset_unit = max_data_len / 8;
    if(buf->len > max_data_len){
        // Step2 : 如果超过IP协议最大负载包长，则需要分片发送。
        // 计算分片数量（向上取整）
        size_t n = (buf->len - 1) / max_data_len + 1;
        // 计算最后一个分片的数据包长度
        size_t last_data_len = buf->len % max_data_len;
        if(last_data_len == 0) last_data_len = max_data_len;
```

首先需要检查从上层传递下来的数据报包长是否大于 IP 协议最大负载包长，如果超过 IP 协议最大负载包长，则需要分片发送。此处我们需要计算分片的数量，以及最后一个分片的数据包长度。

```c
    buf_t ip_buf;
    for(int i = 0; i < n - 1; i++){
        // 首先调用buf_init()初始化一个ip_buf,将数据报包长截断
        buf_init(&ip_buf, max_data_len);
        memcpy(ip_buf.data, buf->data + i * max_data_len, max_data_len);
        // 调用ip_fragment_out()函数发送出去
        ip_fragment_out(&ip_buf, ip, protocol, id, i * offset_unit, 1);
    }
    // 最后一个分片
    // 调用buf_init()初始化一个ip_buf，大小等于该分片大小
    buf_init(&ip_buf, last_data_len);
    memcpy(ip_buf.data, buf->data + (n - 1) * max_data_len, last_data_len);
    // 调用ip_fragment_out()函数发送出去，最后一个分片的MF = 0
    ip_fragment_out(&ip_buf, ip, protocol, id, (n - 1) * offset_unit, 0);
}

// Step3 : 如果没有超过IP协议最大负载包长，则直接调用ip_fragment_out()函数发送出去。
else{
    ip_fragment_out(buf, ip, protocol, id, 0, 0);
}

id ++;
return;
```

我们调用 buf_init()初始化一个 ip_buf,将数据报包长截断，然后调用 ip_fragment_out()函数发送出去。此处需要注意，如果是截断后的最后一个分

片长度小于 IP 协议最大负载包长，则调用 buf_init()初始化 ip_buf 时，大小应该等于该分片的大小，再调用 ip_fragment_out()函数发送出去。同时需注意，最后一个分片的 MF = 0，其它均为 1。

（4）IP 数据报输出处理，即实现 ip_fragment_out()函数

该函数的主要功能是给要发送的分片增加 ip 数据报头部，并填写相关字段，计算校验和，把封装好的数据报发送出去。

算法实现如下图所示：

```
/**
 * @brief 处理一个要发送的ip分片
 *
 * @param buf 要发送的分片
 * @param ip 目标ip地址
 * @param protocol 上层协议
 * @param id 数据包id
 * @param offset 分片offset，必须被8整除
 * @param mf 分片mf标志，是否有下一个分片
 */
void ip_fragment_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol, int id, uint16_t offset, int mf)
{
    // TO-DO
    // Step1 ：调用buf_add_header()增加IP数据报头部缓存空间。
    buf_add_header(buf, sizeof(ip_hdr_t));

    // Step2 ：填写IP数据报头部字段。
    ip_hdr_t* ip_head = (ip_hdr_t*) buf->data;
    ip_head->hdr_len = 5;
    ip_head->version = IP_VERSION_4;
    ip_head->tos = 0;
    ip_head->total_len16 = swap16((uint16_t)buf->len);
    ip_head->id16 = swap16((uint16_t)id);
    ip_head->flags_fragment16 = swap16(((uint16_t)mf << 13) | offset);
    ip_head->ttl = 64;
    ip_head->protocol = protocol;
    memcpy(ip_head->dst_ip, ip, NET_IP_LEN);
    memcpy(ip_head->src_ip, net_if_ip, NET_IP_LEN);

    // Step3 ：先把IP头部的首部校验和字段填0，
    // 再调用checksum16函数计算校验和，然后把计算出来的校验和填入首部校验和字段。
    ip_head->hdr_checksum16 = 0;
    ip_head->hdr_checksum16 = checksum16((uint16_t*)ip_head, sizeof(ip_hdr_t));

    // Step4 ：调用arp_out函数()将封装后的IP头部和数据发送出去。
    arp_out(buf, ip);
    return;
```

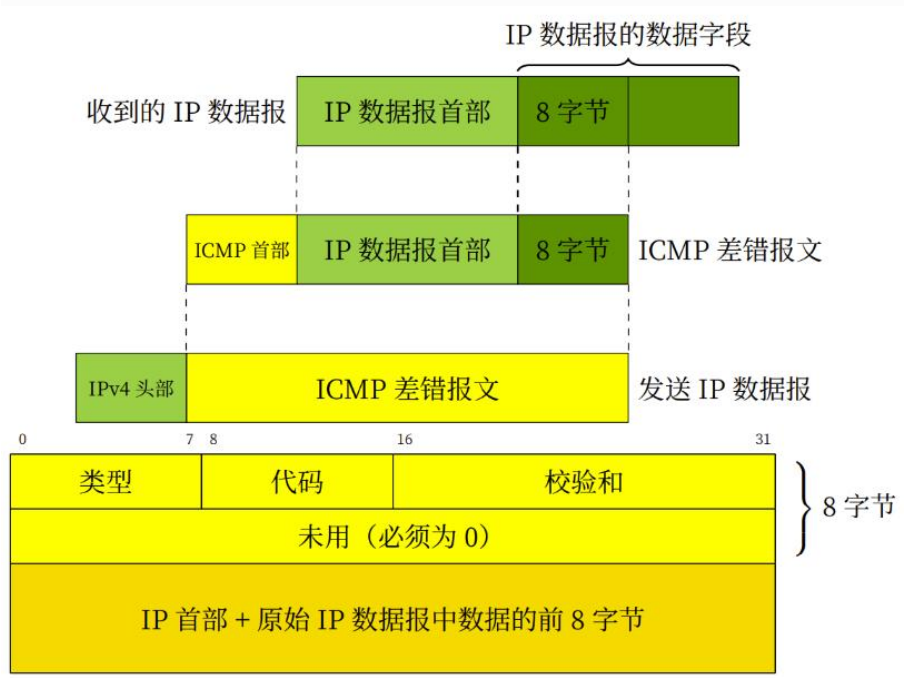
首先，调用 buf_add_header()增加 IP 数据报头部缓存空间。然后，填写 IP 数据报头部字段。接着，先把 IP 头部的首部校验和字段填 0，再调用 checksum16 函数计算校验和，然后把计算出来的校验和填入首部校验和字段。最后，调用 arp_out 函数()将封装后的 IP 头部和数据发送出去。

2. ICMP 协议详细设计

IP 旨在让最终目标主机收到数据包，但在实际通信中，仅凭 IP 远远不够，

它没有为终端设备提供直接的方法来发现发往目的地址失败的 IP 数据包，也不能进行问题诊断，还需要众多支持 IP 的相关技术才能够实现最终通信。将 ICMP 协议与 IP 结合使用，可以提供与 IP 协议层配置和 IP 数据包处理相关的诊断和控制信息。

ICMP 报文结构如下图所示：



本实验需要在完成协议栈之 IP 协议的基础上，编写 ICMP 报文的发送、接收函数，使其能够发送和接收 ICMP 数据报文，并且能通过实验评测系统的测试。

具体而言，需要补充完整 src/icmp.c 文件中的 icmp_in()函数、icmp_unreachable()函数。

（1）ICMP 数据报输入处理，即实现 icmp_in()函数

该函数的主要功能是进行 icmp 报头合法性的检测，同时对回显请求类型的 icmp 报文回送回显应答。

算法实现如下图所示：

```c
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 * @param src_ip 源ip地址
 */
void icmp_in(buf_t *buf, uint8_t *src_ip)
{
    // TO-DO
    // Step1 : 首先做报头检测，如果接收到的包长小于ICMP头部长度，则丢弃不处理。
    if(buf->len < sizeof(icmp_hdr_t)) return;

    // Step2 : 接着，查看该报文的ICMP类型是否为回显请求。
    icmp_hdr_t *icmp_head = (icmp_hdr_t *)buf->data;
    if(icmp_head->type == ICMP_TYPE_ECHO_REQUEST) {
        // Step3 : 如果是，则调用icmp_resp()函数回送一个回显应答（ping 应答）。
        icmp_resp(buf, src_ip);
    }
}
```

首先做报头检测，如果接收到的包长小于 ICMP 头部长度，则丢弃不处理。接着，查看该报文的 ICMP 类型是否为回显请求。如果是，则调用 icmp_resp()函数回送一个回显应答（ping 应答）。

（2）发送 ICMP 响应报文，即实现 icmp_resp()函数

该函数的主要功能是封装报头和数据，然后填写校验和并将数据报发送出去。

算法实现如下图所示：

```c
/**
 * @brief 发送icmp响应
 *
 * @param req_buf 收到的icmp请求包
 * @param src_ip 源ip地址
 */
static void icmp_resp(buf_t *req_buf, uint8_t *src_ip)
{
    // TO-DO
    // Step1 : 调用buf_init()来初始化txbuf
    buf_t *buf = &txbuf;
    buf_init(buf, req_buf->len);
    // 然后封装报头和数据，数据部分可以拷贝来自接收的回显请求报文中的数据
    icmp_hdr_t *icmp_head = (icmp_hdr_t *)buf->data;
    icmp_hdr_t *req_icmp_head = (icmp_hdr_t *)req_buf->data;
    icmp_head->type = ICMP_TYPE_ECHO_REPLY;
    icmp_head->code = 0;
    icmp_head->id16 = req_icmp_head->id16;
    icmp_head->seq16 = req_icmp_head->seq16;
    memcpy(buf->data + sizeof(icmp_hdr_t), req_buf->data + sizeof(icmp_hdr_t)
    , req_buf->len - sizeof(icmp_hdr_t));

    // Step2 : 填写校验和，ICMP的校验和和IP协议校验和算法是一样的。
    icmp_head->checksum16 = 0;
    icmp_head->checksum16 = checksum16((uint16_t *)buf->data, buf->len);

    // Step3 : 调用ip_out()函数将数据报发送出去。
    ip_out(buf, src_ip, NET_PROTOCOL_ICMP);
}
```

首先，调用 buf_init()来初始化 txbuf，然后封装报头和数据，数据部分拷贝来自接收的回显请求报文中的数据。然后，填写校验和，使用和 ip 协议中相同的函数。最后，调用 ip_out()函数将数据报发送出去。

此处填写校验和时同样需要注意要先将校验和设置为 0，然后再计算数据报的校验和并重新赋值。

（3）ICMP 数据报输出处理，即实现 icmp_unreachable()函数

在该函数中，只需要输出 icmp unreachable 的差错报文即可。

算法实现如下图所示：

```c
/**
 * @brief 发送icmp不可达
 *
 * @param recv_buf 收到的ip数据包
 * @param src_ip 源ip地址
 * @param code icmp code，协议不可达或端口不可达
 */
void icmp_unreachable(buf_t *recv_buf, uint8_t *src_ip, icmp_code_t code)
{
    // TO-DO
    // Step1 ：首先调用buf_init()来初始化txbuf，填写ICMP报头首部。
    buf_t *buf = &txbuf;
    buf_init(buf, sizeof(icmp_hdr_t) + sizeof(ip_hdr_t) + 8);
    icmp_hdr_t *icmp_head = (icmp_hdr_t *)buf->data;
    icmp_head->type = ICMP_TYPE_UNREACH;
    icmp_head->code = code;
    icmp_head->id16 = 0;
    icmp_head->seq16 = 0;
    icmp_head->checksum16 = 0;

    // Step2 ：接着，填写ICMP数据部分，
    // 包括IP数据报首部和IP数据报的前8个字节的数据字段，填写校验和。
    memcpy(buf->data + sizeof(icmp_hdr_t), recv_buf->data, sizeof(ip_hdr_t) + 8);
    icmp_head->checksum16 = checksum16((uint16_t *)buf->data, buf->len);

    // Step3 ：调用ip_out()函数将数据报发送出去。
    ip_out(buf, src_ip, NET_PROTOCOL_ICMP);
    return;
}
```

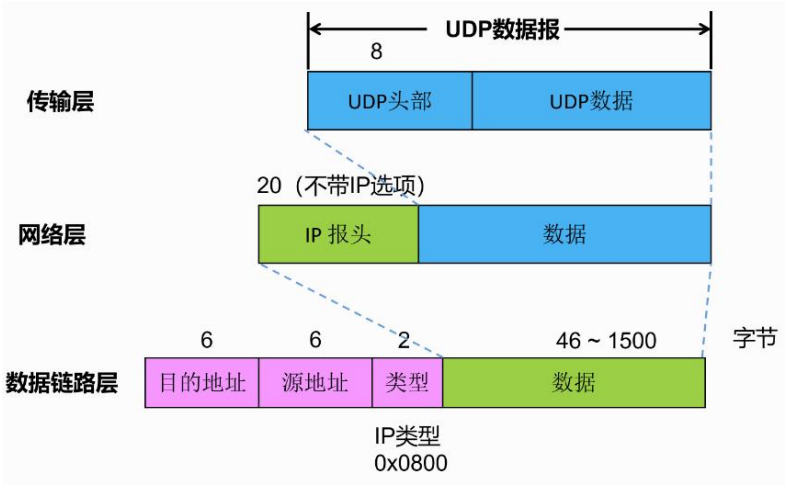
首先调用 buf_init()来初始化 txbuf，填写 ICMP 报头首部。接着，填写 ICMP 数据部分，包括 IP 数据报首部和 IP 数据报的前 8 个字节的数据字段，填写校验和。最后，调用 ip_out()函数将数据报发送出去。

此处的校验和的赋值方式的注意点和上述赋值相同。

3. UDP 协议详细设计

UDP 和 TCP 一样是 TCP/IP 中两个具有代表性的传输层协议。TCP 提供可靠的通信传输，而 UDP 是不具有可靠性的数据报协议。但 UDP 却适用于那些对高速传输和实时性有较高要求的通信或广播通信。

UDP 数据报的结构如下图所示：



本实验需要在完成协议栈之 ICMP 协议的基础上，编写 UDP 报文的发送、接收函数，使其能够发送和接收 UDP 数据报文，并且能通过实验评测系统的测试。

具体而言，需要补充完整 src/udp.c 文件中的 udp_in()函数、udp_out()函数以及 udp_checksum()函数。

（1）UDP 数据报输出处理，即实现 udp_out()函数

该函数的主要功能是添加 UDP 报头，填充相关首部字段，填充校验和，并发送数据报。

算法实现如下图所示：

```c
/**
 * @brief 处理一个要发送的数据包
 *
 * @param buf 要处理的包
 * @param src_port 源端口号
 * @param dst_ip 目的ip地址
 * @param dst_port 目的端口号
 */
void udp_out(buf_t *buf, uint16_t src_port, uint8_t *dst_ip, uint16_t dst_port)
{
    // TO-DO
    // Step1 : 首先调用buf_add_header()函数添加UDP报头。
    buf_add_header(buf, sizeof(udp_hdr_t));

    // Step2 : 接着，填充UDP首部字段。
    udp_hdr_t *udp_head = (udp_hdr_t *)buf->data;
    udp_head->src_port16 = swap16(src_port);
    udp_head->dst_port16 = swap16(dst_port);
    udp_head->total_len16 = swap16(buf->len);

    // Step3 : 先将校验和字段填充0，然后调用udp_checksum()函数计算出校验和，
    // 再将计算出来的校验和结果填入校验和字段。
    udp_head->checksum16 = 0;
    udp_head->checksum16 = udp_checksum(buf, net_if_ip, dst_ip);

    // Step4 : 调用ip_out()函数发送UDP数据报。
    ip_out(buf, dst_ip, NET_PROTOCOL_UDP);
}
```

首先调用 buf_add_header()函数添加 UDP 报头。接着，填充 UDP 首部字段。先将校验和字段填充 0，然后调用 udp_checksum()函数计算出校验和，再将计算出来的校验和结果填入校验和字段。最后调用 ip_out()函数发送 UDP 数据报。

注意此处计算校验和的函数与 ip 和 icmp 中计算校验和的函数不同，因为 udp 校验和的覆盖范围与 ip 首部校验和的覆盖范围是不一样的。UDP 校验和需要覆盖 UDP 头部、UDP 数据和伪头部。

（2）UDP 数据报输入处理，即实现 udp_in()函数

该函数的主要功能是进行数据报的合法性检查，通过校验和检查是否出错，检查是否有对应的处理函数并作出对应处理。

算法实现如下图所示：

```c
/**
 * @brief 处理一个收到的udp数据包
 *
 * @param buf 要处理的包
 * @param src_ip 源ip地址
 */
void udp_in(buf_t *buf, uint8_t *src_ip)
{
    // TO-DO
    // Step1 : 首先做包检查，检测该数据报的长度是否小于UDP首部长度
    if(buf->len < sizeof(udp_hdr_t)) return;
    // 接收到的包长度是否小于UDP首部长度字段给出的长度
    udp_hdr_t* udp_head = (udp_hdr_t*) buf->data;
    if(swap16(udp_head->total_len16) < sizeof(udp_hdr_t)) return;

    // Step2 : 接着重新计算校验和，先把首部的校验和字段保存起来，
    // 然后把该字段填充0，调用udp_checksum()函数计算出校验和，
    // 如果该值与接收到的UDP数据报的校验和不一致，则丢弃不处理。
    uint16_t old_checksum = udp_head->checksum16;
    udp_head->checksum16 = 0;
    uint16_t new_checksum = udp_checksum(buf, src_ip, net_if_ip);
    if(new_checksum != old_checksum) return;
    udp_head->checksum16 = old_checksum;

    // Step3 : 调用map_get()函数查询udp_table是否有
    // 该目的端口号对应的处理函数（回调函数）。
    uint16_t dst_port16 = swap16(udp_head->dst_port16);
    udp_handler_t* handler = map_get(&udp_table, &dst_port16);
```

首先做包检查，检测该数据报的长度是否小于 UDP 首部长度，或者接收到的包长度小于 UDP 首部长度字段给出的长度，如果是，则丢弃不处理。接着重新计算校验和，先把首部的校验和字段保存起来，然后把该字段填充 0，调用 udp_checksum()函数计算出校验和，如果该值与接收到的 UDP 数据报的校验和不一致，则丢弃不处理。然后，调用 map_get()函数查询 udp_table 是否有该目的端口号对应的处理函数（回调函数）。

```c
if(!handler) {
    // Step4 : 如果没有找到，则调用buf_add_header()函数增加IPv4数据报头部，
    // 再调用icmp_unreachable()函数发送一个端口不可达的ICMP差错报文。
    buf_add_header(buf, sizeof(ip_hdr_t));
    icmp_unreachable(buf, src_ip, ICMP_CODE_PORT_UNREACH);
} else {
    // Step5 : 如果能找到，则去掉UDP报头，调用处理函数来做相应处理。
    buf_remove_header(buf, sizeof(udp_hdr_t));
    (*handler)(buf->data, buf->len, src_ip, dst_port16);
}
```

如果没有找到对应的处理函数，则调用 buf_add_header()函数增加 IPv4 数据报头部，再调用 icmp_unreachable()函数发送一个端口不可达的 ICMP 差错报文。如果能找到，则去掉 UDP 报头，调用处理函数来做相应处理。

（3）UDP 校验和，即实现 udp_checksum()函数

如上述所说，udp 的校验和计算方式与另两种协议的计算方式不同，因此需要另外实现，算法实现如下图所示：

```
/**
 * @brief udp伪校验和计算
 *
 * @param buf 要计算的包
 * @param src_ip 源ip地址
 * @param dst_ip 目的ip地址
 * @return uint16_t 伪校验和
 */
static uint16_t udp_checksum(buf_t *buf, uint8_t *src_ip, uint8_t *dst_ip)
{
    // TO-DO
    // Step1 : 首先调用buf_add_header()函数增加UDP伪头部。
    udp_hdr_t *udp_head = (udp_hdr_t *)buf->data;
    buf_add_header(buf, sizeof(udp_peso_hdr_t));

    // Step2 : 将被UDP伪头部覆盖的IP头部拷贝出来，暂存IP头部，以免被覆盖。
    udp_peso_hdr_t* udp_peso_head = (udp_peso_hdr_t*) buf->data;
    udp_peso_hdr_t udp_peso_head_temp = *udp_peso_head;

    // Step3 : 填写UDP伪头部的12字节字段。
    memcpy(udp_peso_head->src_ip, src_ip, NET_IP_LEN);
    memcpy(udp_peso_head->dst_ip, dst_ip, NET_IP_LEN);
    udp_peso_head->placeholder = 0;
    udp_peso_head->protocol = NET_PROTOCOL_UDP;
    udp_peso_head->total_len16 = udp_head->total_len16;
    int pad = 0;
    if(buf->len % 2) {
        //若数据长度为奇数，则需要填充一个字节
        pad = 1;
        buf_add_padding(buf, 1);
    }
```

首先调用 buf_add_header()函数增加 UDP 伪头部。将被 UDP 伪头部覆盖的 IP 头部拷贝出来，暂存 IP 头部，以免被覆盖。然后填写 UDP 伪头部的 12 字节字段。

此处需注意，如果数据的长度为奇数，则需要填充一个字节。

```
// Step4 : 计算UDP校验和。
uint16_t checksum = checksum16((uint16_t *)buf->data, buf->len);
// 算完校验和后再将pad去掉
if(pad) buf_remove_padding(buf, 1);

// Step5 : 再将 Step2 中暂存的IP头部拷贝回来。
*udp_peso_head = udp_peso_head_temp;

// Step6 : 调用buf_remove_header()函数去掉UDP伪头部。
buf_remove_header(buf, sizeof(udp_peso_hdr_t));

// Step7 : 返回计算出来的校验和值。
return checksum;
```

接着计算 UDP 校验和。算完校验和之后再将之前可能填充的字节去掉。将之前暂存的 IP 头部拷贝回来，再调用 buf_remove_header()函数去掉 UDP 伪头部。最后返回计算出来的校验和值。

二、 实验结果截图及分析

（对你自己实验的测试结果进行评价）

1. IP 协议实验结果及分析

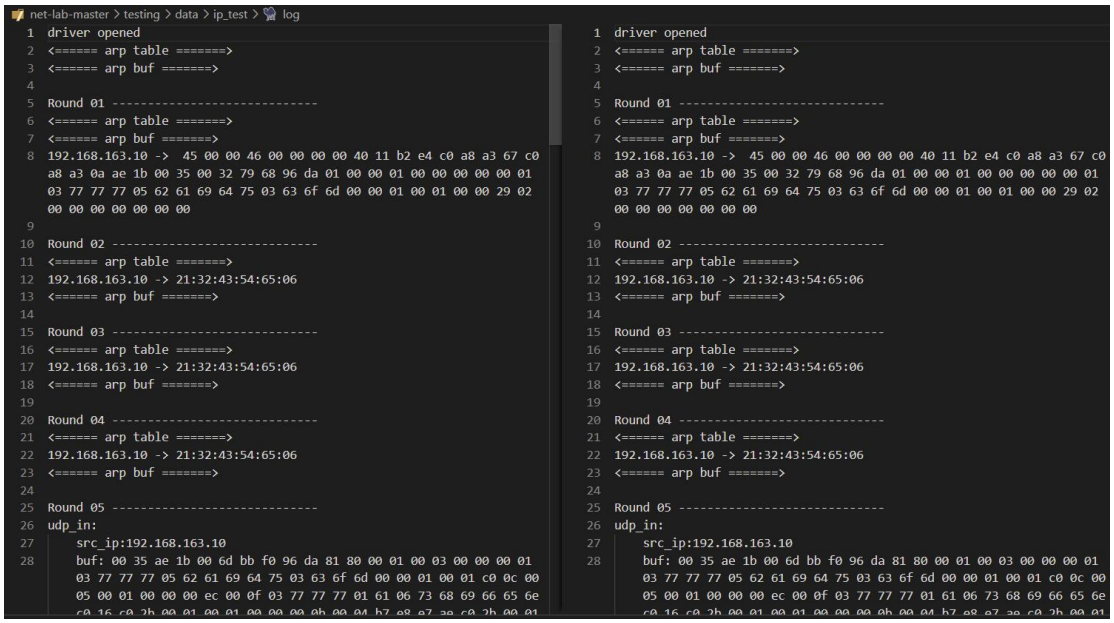
ctest -R ip_test

```
PS E:\lab2\net-lab-master\build> ctest -R ip_test
Test project E:/lab2/net-lab-master/build
    Start 4: ip_test
1/1 Test #4: ip_test ..........................   Passed    0.04 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.06 sec
PS E:\lab2\net-lab-master\build>
```

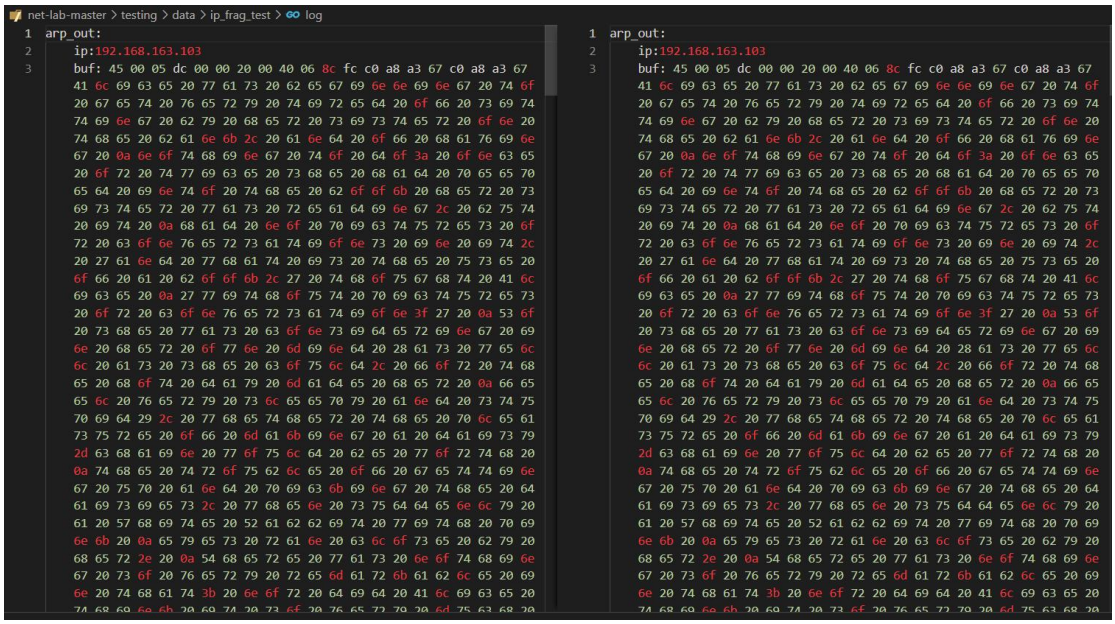
ctest -R ip_frag_test

```
PS E:\lab2\net-lab-master\build> ctest -R ip_frag_test
Test project E:/lab2/net-lab-master/build
    Start 5: ip_frag_test
1/1 Test #5: ip_frag_test .....................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.02 sec
```

在 VSCode 工程目录下，将 testing/ip_test 目录下的 demo_log 和 log 这两个文件进行比对，发现二者一致：



在 VSCode 工程目录下，将 testing/ip_frag_test 目录下的 demo_log 和 log 这两个文件进行比对，发现二者一致：

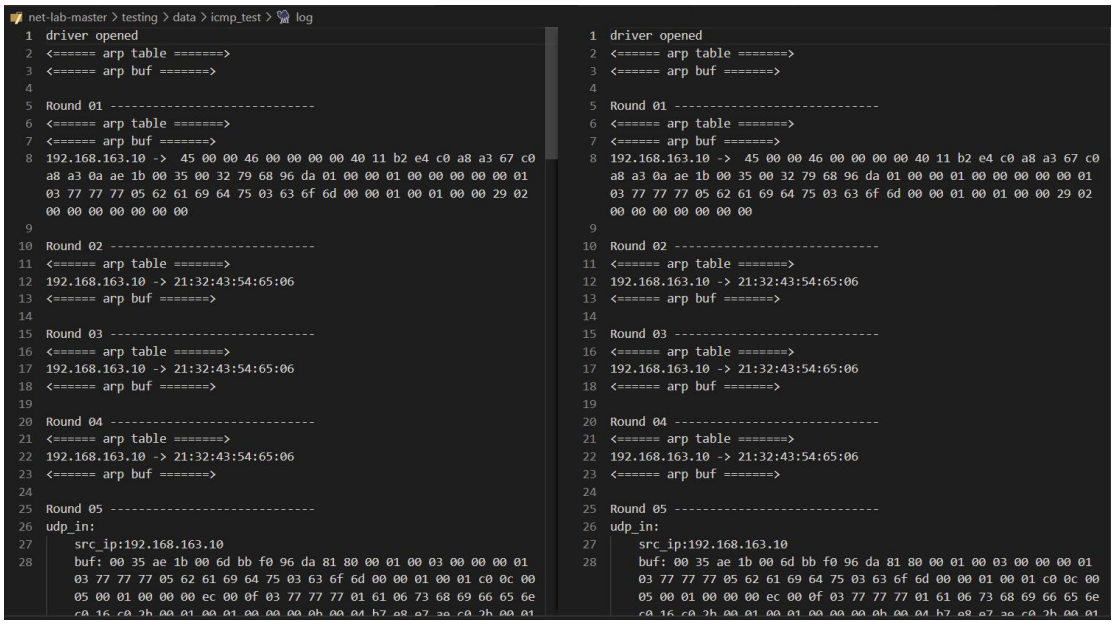


2. ICMP 协议实验结果及分析

ctest -R icmp_test

```
PS E:\lab2\net-lab-master\build> ctest -R icmp_test
Test project E:/lab2/net-lab-master/build
    Start 6: icmp_test
1/1 Test #6: icmp_test .......................   Passed    0.04 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.04 sec
PS E:\lab2\net-lab-master\build>
```

在 VSCode 工程目录下，将 testing/icmp_test 目录下的 demo_log 和 log 这两个文件进行比对，发现二者一致：

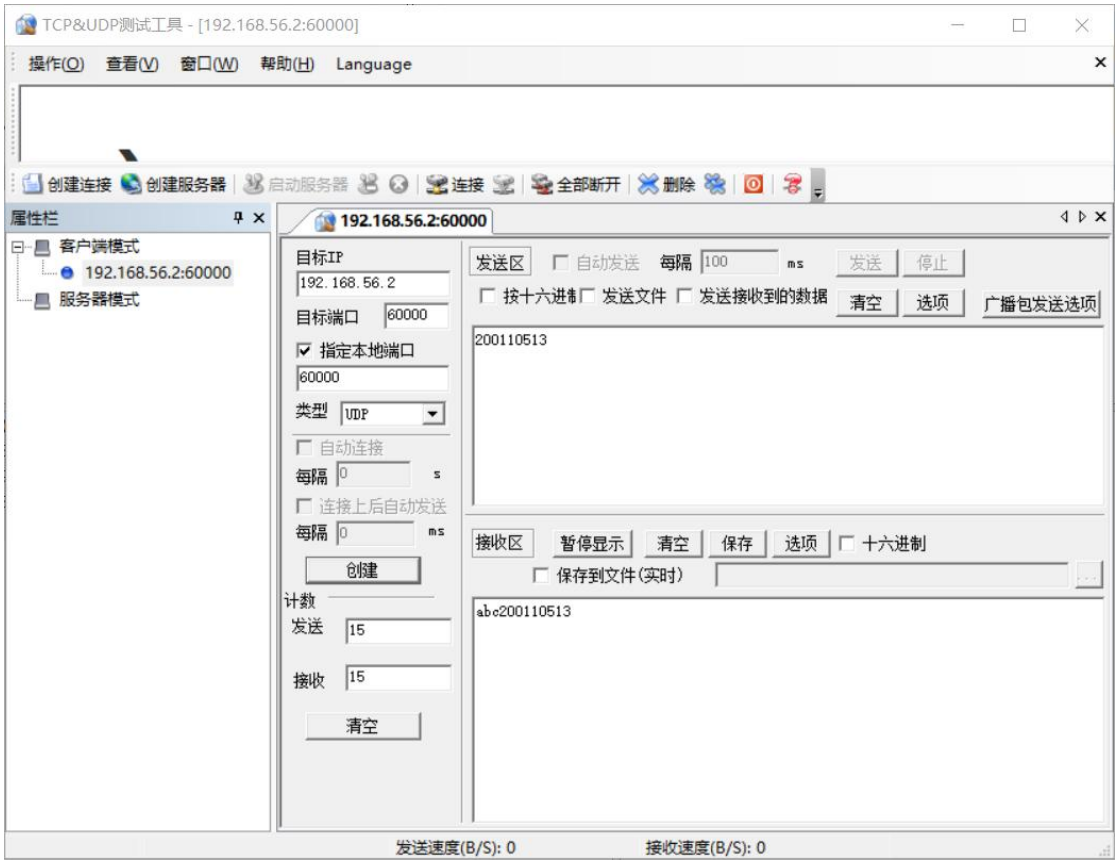


```
driver opened
<====== arp table =======>
<====== arp buf =======>

Round 01 ----------------------------
<====== arp table =======>
<====== arp buf =======>
192.168.163.10 ->  45 00 00 46 00 00 00 00 40 11 b2 e4 c0 a8 a3 67 c0
a8 a3 0a ae 1b 00 35 00 32 79 68 96 da 01 00 00 01 00 00 00 00 00 01
03 77 77 77 05 62 61 69 64 75 03 63 6f 6d 00 00 01 00 01 00 00 29 02
00 00 00 00 00 00 00

Round 02 ----------------------------
<====== arp table =======>
192.168.163.10 -> 21:32:43:54:65:06
<====== arp buf =======>

Round 03 ----------------------------
<====== arp table =======>
192.168.163.10 -> 21:32:43:54:65:06
<====== arp buf =======>

Round 04 ----------------------------
<====== arp table =======>
192.168.163.10 -> 21:32:43:54:65:06
<====== arp buf =======>

Round 05 ----------------------------
udp_in:
    src_ip:192.168.163.10
    buf: 00 35 ae 1b 00 6d bb f0 96 da 81 80 00 01 00 03 00 00 00 01
    03 77 77 77 05 62 61 69 64 75 03 63 6f 6d 00 00 01 00 01 c0 0c 00
    05 00 01 00 00 00 ec 00 0f 03 77 77 77 01 61 06 73 68 69 66 65 6e
```

3. UDP 协议实验结果及分析

（本小节还需要分析你自己用 Wireshark 抓包工具捕获到的相关报文（包含 UDP 和 ARP 报文），解析报文内容）

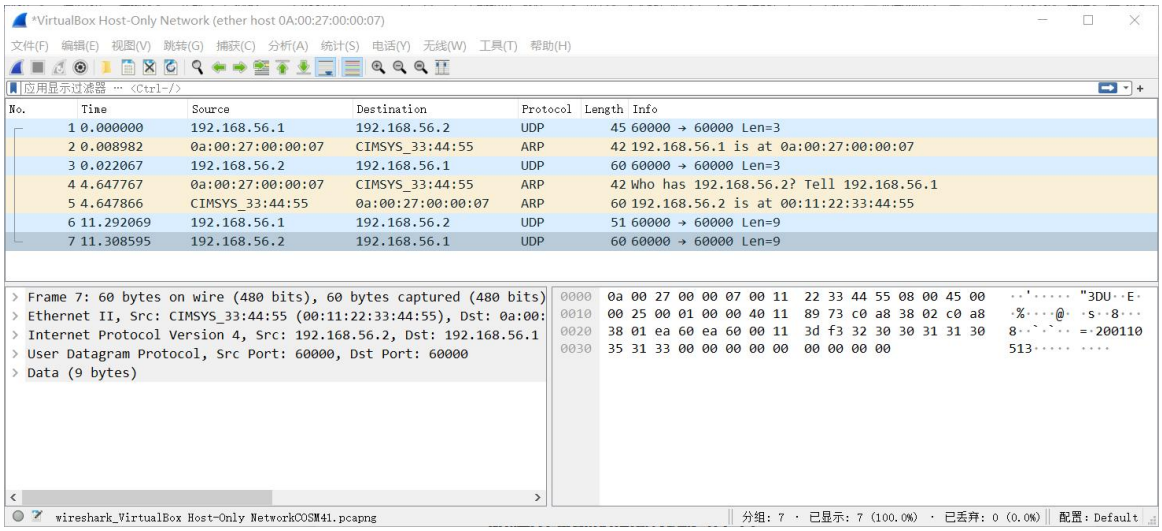
我通过 TCP&UDPDebug 调试工具发送了两条消息：“abc”和“200110513”，如下图所示：



在 vscode 终端中同步收到了这两条消息，如下图所示。显示消息均来自 ip 地址 192.168.56.1，第一条消息长度为 3，内容为“abc”；第二条消息长度为 9，内容为“200110513”。

```
PS E:\lab2\net-lab-master\build> ."E:/lab2/net-lab-master/build/main.exe"
Using interface \Device\NPF_{313335BF-2B53-4379-BD77-A2DE650819CD}, my ip is 192.168.56.2.
tcp open
tcp open
recv udp packet from 192.168.56.1:60000 len=3
abc
recv udp packet from 192.168.56.1:60000 len=9
200110513
```

此时，Wireshark 上也有数据包的交互，如下图所示。



可以看出，第一行是一个 UDP 报文，通过本机的真实网卡向虚拟网卡发送消息，消息长度为 3，即上述的第一条消息“abc”。然后第二行是一个 ARP 报文，从本机的真实网卡发出，内容为报告本机的 ip 地址所对应的 mac 地址。第三行是一个 UDP 报文，通过虚拟网卡向本机的真实网卡发送消息，消息长度为 3，这仍是上述的第一条消息“abc”。这是因为，虚拟网卡接收到 UDP 报文后会在终端显示报文相关信息，并向源 ip 地址发送相同的 UDP 报文，如下图所示：

```
#ifdef UDP
void udp_handler(uint8_t* data, size_t len, uint8_t* src_ip, uint16_t src_port)
{
    printf("recv udp packet from %s:%u len=%zu\n", iptos(src_ip), src_port, len);
    for (int i = 0; i < len; i++)
        putchar(data[i]);
    putchar('\n');
    udp_send(data, len, 60000, src_ip, src_port); //发送udp包
}
#endif
```

第四行是来自本机的真实网卡发出的 ARP 报文，内容是请求对方 ip 地址所对应的 mac 地址。第五行是对上一条报文的回复，也是 ARP 报文，告诉其 mac 地址。第六行是 UDP 报文，通过本机的真实网卡向虚拟网卡发送消息，消息长度为 9，即上述的第二条消息“200110513”。最后一条是 UDP 报文，通过虚拟网卡向本机的真实网卡发送消息，消息长度为 9，这仍是上述的第二条消息“200110513”，理由不再赘述。

三、 实验中遇到的问题及解决方法

（包括设计过程中的错误及测试过程中遇到的问题）

在写 checksum 相关的代码时，出现了比较多的错误，如一开始没有把 checksum 置零，以及在通过 checksum 验证是否出错后，未把其恢复成原来的值等。

以及最后对于 Wireshark 和 TCP&UDPDebug 调试工具的使用比较手忙脚乱，

研究了比较久的时间才弄明白应该如何操作。

四、 实验收获和建议

（实验过程中的收获、感受、问题、建议等）

在实验中，我感受到实验的设置和理论课上所学的内容联系非常紧密，并且在写代码的过程中，我对理论课上所学的抽象的知识有了更具体的理解。在这三个小实验中，我体会到了分层的重要性，它使得代码的逻辑变得更加简明清晰，便于理解。

在实验中遇到的问题就是最后对于 Wireshark 和 TCP&UDPDebug 调试工具的使用不太熟悉，并且在指导书上对于 Wireshark 的操作步骤还有一些缺失，比如不知道选取哪个网卡，不知道在哪里设置过滤条件，过滤条件的格式不正确等等，需要自己探索，初次使用比较摸不着头脑。

建议指导书上可以把 Wireshark 的软件操作部分写的更加详细一点，可以便于同学们快速上手。