

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。)

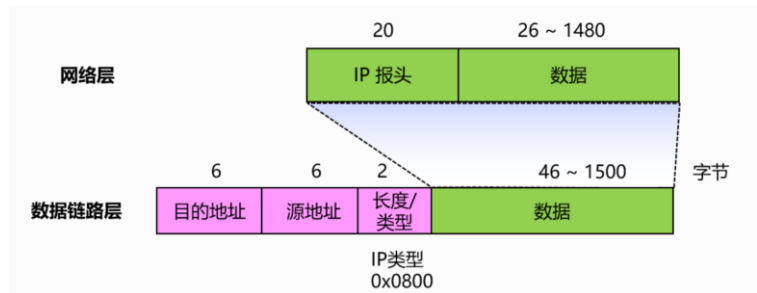
1. IP 协议详细设计

【整体设计思路】

IP 协议位于网络层，提供了一种**尽力而为、无连接**的数据包交付服务，即不保证数据报能成功到达目的地，也不维护路由器中数据报相关的任何链接状态信息，每个数据报可独立处理，可以不按顺序交付。由此，IP 协议可实现跨越不同数据链路的两个终端节点之间的通信。**IP 协议主要分为三大模块，包括 IP 寻址、IP 分包与组包、路由转发。其中，路由转发模块在本次实验当中没有涉及。**

因此，本小节实验的**主要任务**在于：**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。**另外，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。

网络层的数据包结构如下图所示：



本次实验基于 IPv4 协议完成。IPv4 的头部结构如下图所示：



因此，本小节实验在**填充或解析 IP 头部**的时候，需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。**

(1) IP 数据报输入处理函数 ip_in 的设计与实现

流程设计如下:

- 【第一步】如果数据包的长度小于 IP 头部长度，丢弃不处理。
- 【第二步】创建备份，便于发送 ICMP 协议不可达信息。
- 【第三步】进行报头检测，检测内容包括版本号是否为 IPv4、总长度字段是否小于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。
- 【第四步】先把 IP 头部的头部校验和字段用其它变量保存起来，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。如果不一致，丢弃不处理；如果一致，恢复头部校验和字段为原来的值。
- 【第五步】对比目的 IP 地址是否为本机 IP 地址，如果不是，则丢弃不处理。
- 【第六步】如果数据包长度大于 IP 头部的总长度字段，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。
- 【第七步】调用 buf_remove_header 函数去除 IP 报头。
- 【第八步】调用 net_in 函数向上层传递数据包。

说明以及值得注意的地方:

- 大部分操作都是“按图索骥”地根据 IP 头的结构来完成对 IP 头部字段的解析。
- 由于需要进行 ICMP 协议不可达信息的发送，因此可以在一开始创建一个关于 IP 头部的备份，以便于后续检测到不可达时进行发送。
- 在计算校验和之前，需要先将 IP 头部的校验和字段设置为 0，计算完毕后再将校验和字段恢复为原有的值（当然，若校验失败则直接丢弃，不必恢复了）。
- 与此前的协议栈实验同理，还需要注意处理好长度超过一个字节的字段的大小端序转换问题。

代码实现如下:

```
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: 创建备份，便于发送 ICMP 协议不可达信息
    buf_t copy;
    buf_copy(&copy, buf, buf->len);

    // Step3: 进行报头检测，检测内容包括版本号是否为 IPv4、总长度字段是否小于
    // 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
    ip_hdr_t *hdr = (ip_hdr_t *) buf->data;
    if (hdr->version != IP_VERSION_4) return;
    if (swap16(hdr->total_len16) > buf->len) return;

    // Step4: 先把 IP 头部的头部校验和字段用其它变量保存起来，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。如果不一致，丢弃不处理；如果
    一致，恢复头部校验和字段为原来的值
    uint16_t hdr_checksum16_backup = hdr->hdr_checksum16;
    hdr->hdr_checksum16 = 0;
    uint16_t hdr_checksum16 = checksum16((uint16_t *)hdr,
sizeof(ip_hdr_t));
    if (hdr_checksum16 != hdr_checksum16_backup) return;
    hdr->hdr_checksum16 = hdr_checksum16_backup;

    // Step5: 对比目的 IP 地址是否为本机 IP 地址，如果不是，则丢弃不处理
    if (memcmp(hdr->dst_ip, net_if_ip, NET_IP_LEN) != 0) return;

    // Step6: 如果数据包长度大于 IP 头部的总长度字段，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 > swap16(hdr->total_len16)) buf_remove_padding(buf,
buf->len - swap16(hdr->total_len16));

    // Step7: 调用 buf_remove_header 函数去除 IP 报头
    uint8_t protocol = hdr->protocol;
    uint8_t *src_ip = hdr->src_ip;
    buf_remove_header(buf, hdr->hdr_len * 4);

    // Step8: 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 函数返回 ICMP 协议不
    可达信息。
    int flag = net_in(buf, protocol, src_ip);
    if (flag == -1)
        icmp_unreachable(&copy, src_ip, ICMP_CODE_PROTOCOL_UNREACH);
}

```

(2) 校验和算法处理函数 checksum16 的设计与实现

流程设计如下：

- 【第一步】把 data 看出是每两个字节组成一个数，不断累加；若最后剩下一个字节
的值，也要相加这个值。由于 16 位加法可能会溢出，因此结果 sum 要用 32 位来保存。
- 【第二步】判断相加后的结果的高 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。
- 【第三步】将上述和的低 16 位进行取反，得到校验和。

说明以及值得注意的地方：

- 计算校验和的主要过程，就是把 IP 头部分成每两个字节一组来进行叠加（若最终剩余
了一个字节，也要将其相加），再不断让得到的 32 位结果的高 16 位和低 16 位相加，
直至高 16 位为 0 为止。最后，将低 16 位结果取反，就可以得到校验和了。
- 校验和字段虽然是 16 位的数据，但是在本次实验中不需要考虑大小端序的转换。这是
因为，校验和将 16 位的加法结果用 32 位来存储，高 16 位的进位最终都会被加回到低

16 位。因此，最终经过反转后，大小端不匹配的加法和正常加法结果都相等。我们的老师也在课程群里面分享了上一届学生关于这个问题的推导过程，具体如下图所示：

网络中的数据是按照大端模式。如果操作系统是小端模式怎么办？是不是应该取出数据之后先做字节反转，加完后再做字节反转换回来呢？事实上，在校验和设计的过程中，其步骤就已经考虑到了这一点。↵

考虑两个数 (高字节低字节由竖线分离开来)↵

$$x_1, x_2$$

$$x_1 = \alpha_1 | \beta_1$$

$$x_2 = \alpha_2 | \beta_2$$

正常的加法如下↵

$$x_1 + x_2 = (\alpha_1 + \alpha_2) | (\beta_1 + \beta_2) \quad (1)$$

如果有进位的话↵

$$x_1 + x_2 = (\alpha_1 + \alpha_2 + k_p) \bmod 10000H | (\beta_1 + \beta_2) \bmod 10000H \quad (2)$$

如果我们大小端不匹配，直接进行加法的话↵

$$x_1 + x_2 = (\beta_1 + \beta_2 + k_a) \bmod 10000H | (\alpha_1 + \alpha_2) \bmod 10000H \quad (3)$$

反转后↵

$$x_1 + x_2 = (\alpha_1 + \alpha_2) \bmod 10000H | (\beta_1 + \beta_2 + k_a) \bmod 10000H \quad (4)$$

显然 (2) 与 (4) 结果不同↵

但是校验有一个机制：十六位的加法结果用三十二位来存储，高十六位的进位最后要加至低位。我们看看这样会发生什么：↵

正常加法↵

$$x_1 + x_2 = (\alpha_1 + \alpha_2 + k_p) \bmod 10000H | (\beta_1 + \beta_2 + k_a) \bmod 10000H \quad (5)$$

大小端不匹配加法↵

$$x_1 + x_2 = (\beta_1 + \beta_2 + k_a) \bmod 10000H | (\alpha_1 + \alpha_2 + k_p) \bmod 10000H \quad (6)$$

翻转后↵

$$x_1 + x_2 = (\alpha_1 + \alpha_2 + k_p) \bmod 10000H | (\beta_1 + \beta_2 + k_a) \bmod 10000H \quad (7)$$

这就说明，实际上 checksum 无需关心大小端，直接进行加法即可，最后结果一样。↵

- 综上所述，在进行反码求和校验不需要考虑字节序。

代码实现如下：

```
uint16_t checksum16(uint16_t *data, size_t len)
{
    // TO-DO
    uint32_t sum = 0;
    // Step1: 把 data 看出是每两个字节组成一个数，不断累加；若最后剩下一个字节
    // 的值，也要相加这个值
    // 由于 16 位加法可能会溢出，因此结果 sum 要用 32 位来保存
    for (int i = 0; i < len; i += 2)
    {
        if (i == len - 1)
        {
            // i 为 (len - 1) 说明此时还剩余 8 位没有加，故需要再加上
            sum += *((uint8_t *)data + i);
        }
        else
        {
            sum += data[i / 2];
        }
    }

    // Step2: 判断相加后的结果的高 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 ((sum >> 16) != 0)
{
    sum = (sum & 0xFFFF) + (sum >> 16);
}

// Step3: 将上述和的低 16 位进行取反，得到校验和
return (~sum) & 0xFFFF;
}

```

(3) IP 数据报输出处理函数 ip_out 的设计与实现

流程设计如下：

- 【第一步】求出 IP 协议最大负载包长（1500 字节即 MTU 大小，再减去 IP 首部长度）。
- 【第二步】如果数据包长度超过 IP 协议的最大负载包长，则需要分片发送。
- 【第三步】对于没有超过 IP 协议最大负载包长的数据包，或者分片后的最后的一个分片小于或等于 IP 协议最大负载包长的数据包，统一再进行一次发送。由于两种情况下都是发送最后一个分片，因此需要设置 MF 为 0。

说明以及值得注意的地方：

- 由于存在 MTU 的限制，长度超过 MTU 的数据报都要被分片，实际传输的 IP 分片数据报的长度远远没有达到理论上可行的最大值。因此，可以先实现发送每一个 IP 分片的函数 ip_fragment_out，然后再在 ip_out 函数中进行对其调用。
- 填充 IP 头部字段的工作已经在 ip_fragment_out 函数中实现，故函数 ip_out 需要再分别处理分片发送、不分片发送这两种情况。对于两种情况下所发送的最后一个分片，都需要注意设置“更多分片”标志 MF 为 0。

代码实现如下：

```

void ip_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol)
{
    // TO-DO
    // Step1: 求出 IP 协议最大负载包长（1500 字节即 MTU 大小，再减去 IP 首部长度）
    int max_load_length = 1500 - sizeof(ip_hdr_t);

    // Step2: 如果数据包长度超过 IP 协议的最大负载包长，则需要分片发送
    int i;
    static int id = 0;
    for (i = 0; (i + 1) * max_load_length < buf->len; i++)
    {
        buf_t ip_buf;
        buf_init(&ip_buf, max_load_length);
        memcpy(ip_buf.data, buf->data + i * max_load_length,
max_load_length);
    }
}

```

```

        ip_fragment_out(&ip_buf, ip, protocol, id, i * (max_load_length >>
3), 1);
    }

    // Step3: 对于没有超过 IP 协议最大负载包长的数据包, 或者分片后的最后的一个
    分片小于或等于 IP 协议最大负载包长的数据包, 统一再进行一次发送
    // 由于两种情况下都是发送最后一个分片, 因此需要设置 MF 为 0
    buf_t ip_buf;
    buf_init(&ip_buf, buf->len - i * max_load_length);
    memcpy(ip_buf.data, buf->data + i * max_load_length, buf->len - i *
max_load_length);
    ip_fragment_out(&ip_buf, ip, protocol, id, i * (max_load_length >> 3),
0);

    id++;
}

```

(4) IP 分片输出处理函数 ip_fragment_out 的设计与实现

流程设计如下:

- 【第一步】调用 buf_add_header 增加 IP 数据报头部缓存空间。
- 【第二步】填写 IP 数据报头部字段。
- 【第三步】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。
- 【第四步】调用 arp_out 函数将封装后的 IP 头部和数据发送出去。

说明以及值得注意的地方:

- 大部分操作都是“按图索骥”地根据 IP 头的结构来完成对 IP 头部字段的填充。
- 与 ip_in 函数同理, 计算首部校验和之前需要将原本的首部校验和字段设置为 0, 在填充各字段时也要注意是否需要进行大小端序转换的问题。

代码实现如下:

```

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 *hdr = (ip_hdr_t *) buf->data;
    hdr->version = IP_VERSION_4;
    hdr->hdr_len = sizeof(ip_hdr_t) / 4;
    hdr->tos = 0;
    hdr->total_len16 = swap16(buf->len);
}

```

```

hdr->id16 = swap16(id);
hdr->flags_fragment16 = swap16((mf ? IP_MORE_FRAGMENT : 0) | offset);
hdr->ttl = 64;
hdr->protocol = protocol;
memcpy(hdr->src_ip, net_if_ip, NET_IP_LEN);
memcpy(hdr->dst_ip, 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
hdr->hdr_checksum16 = 0;
uint16_t new_checksum = checksum16((uint16_t *)hdr, sizeof(ip_hdr_t));
hdr->hdr_checksum16 = new_checksum;

// 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);
}

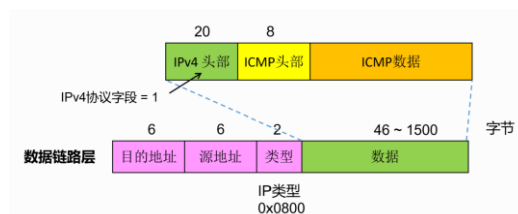
```

2. ICMP 协议详细设计

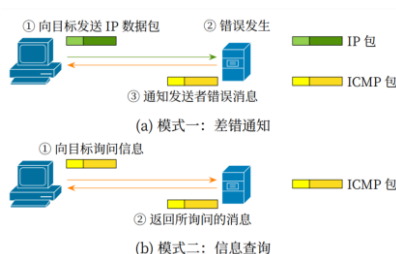
【整体设计思路】

上一个小组的实验实现了 IP 协议，但在实际通信中，仅有 IP 协议远远不够。例如，IP 协议本身并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。为解决这些不足，可以将 ICMP 协议与 IP 协议结合使用，提供与 IP 协议层配置、IP 数据包处理相关的诊断和控制信息。故 ICMP 的主要功能包括：确认 IP 包是否成功送达目标地址、通知发送过程当中 IP 包被废弃的具体原因、改善网络设置等。

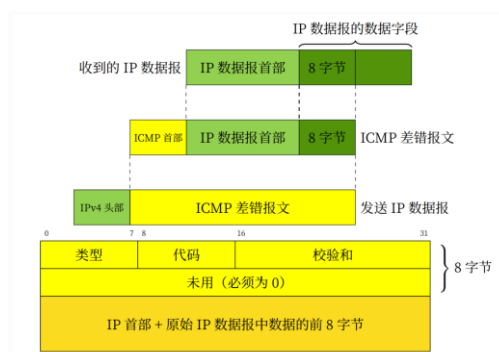
ICMP 报文被封装在 IP 数据报内传输，格式结构如下图所示：



ICMP 的消息主要分为两类，包括差错报文、查询/信息报文，分别用于通知出错原因、查询信息，如下图所示：



ICMP 差错报文会保留收到的 IP 数据报的前 8 个字节，以便于反馈源端口号，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：



因此，本小节实验的主要任务是：分别实现 ICMP 差错报文、ICMP 查询报文的发送函数，填写相应的字段；在收到 ICMP 报文的时候，判断是否为回显请求报文（如果是则发送回显应答）。相应需要考虑的字段主要包括：类型、代码、校验和、标识符、序列号等。

（1）ICMP 数据报输入处理函数 icmp_in 的设计与实现

流程设计如下：

- 【第一步】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。
- 【第二步】接着，查看该报文的 ICMP 类型是否为回显请求。
- 【第三步】如果是，则调用 icmp_resp()函数回送一个回显应答。

代码实现如下：

```
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 *hdr = (icmp_hdr_t *)buf->data;

    // Step3: 如果是，则调用 icmp_resp()函数回送一个回显应答
    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

流程设计如下：

- 【第一步】调用 buf_init()来初始化 txbuf，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。
- 【第二步】填写校验和，ICMP 的校验和和 IP 协议校验和算法是一样的。
- 【第三步】调用 ip_out()函数将数据报发送出去。

代码实现如下:

```
static void icmp_resp(buf_t *req_buf, uint8_t *src_ip)
{
    // TO-DO
    // Step1: 调用 buf_init()来初始化 txbuf, 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
    buf_init(&txbuf, req_buf->len);
    memcpy(txbuf.data, req_buf->data, req_buf->len);
    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;
    hdr->code = 0;
    hdr->id16 = req_hdr->id16;
    hdr->seq16 = req_hdr->seq16;

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

    // Step3: 调用 ip_out()函数将数据报发送出去
    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

流程设计如下:

- 【第一步】首先调用 buf_init()来初始化 txbuf, 填写 ICMP 报头首部。
- 【第二步】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。
- 【第三步】调用 ip_out()函数将数据报发送出去。

代码实现如下:

```
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
    int total_size = sizeof(icmp_hdr_t) + sizeof(ip_hdr_t) + 8;
    buf_init(&txbuf, total_size);
    icmp_hdr_t *icmp_hdr = (icmp_hdr_t *)txbuf.data;
    icmp_hdr->type = ICMP_TYPE_UNREACH;
    icmp_hdr->code = code;
    icmp_hdr->id16 = 0;
    icmp_hdr->seq16 = 0;
```

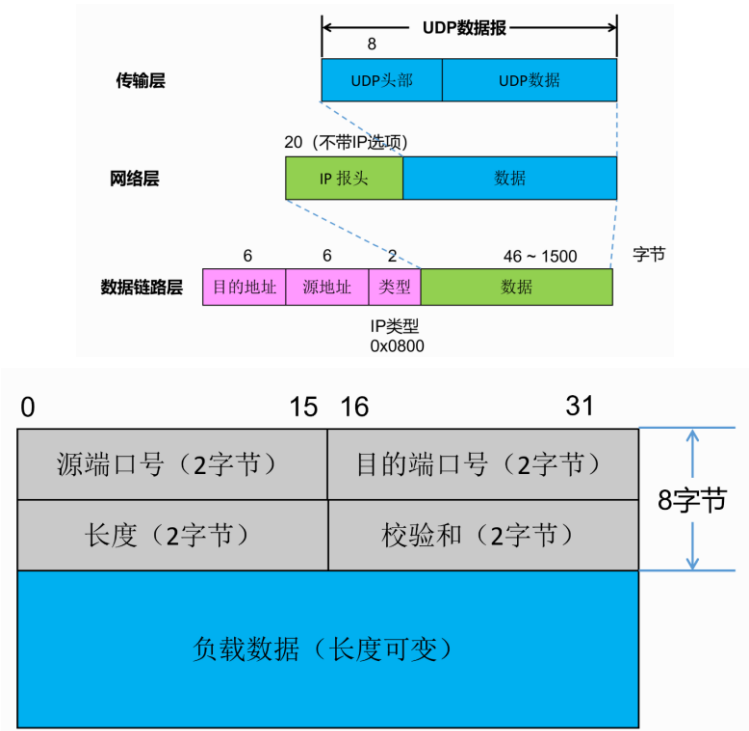
```
// Step2: 填写 ICMP 数据部分, 包括 IP 数据报头部和 IP 数据报的前 8 个字节的数据
// 填写首部
ip_hdr_t *ip_hdr = (ip_hdr_t *) (icmp_hdr + 1);
ip_hdr_t *recv_ip_hdr = (ip_hdr_t *) recv_buf->data;
memcpy(ip_hdr, recv_ip_hdr, sizeof(ip_hdr_t));
// 填写数据
uint8_t *data = (uint8_t *) (ip_hdr + 1);
uint8_t *recv_ip_data = (uint8_t *) (recv_ip_hdr + 1);
memcpy(data, recv_ip_data, 8);
// 填写校验和
icmp_hdr->checksum16 = 0;
icmp_hdr->checksum16 = checksum16((uint16_t *) txbuf.data, total_size);

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

3. UDP 协议详细设计

【整体设计思路】

UDP 是 TCP/IP 中的一个具有代表性的传输层协议，是不具有可靠性的数据报协议，适用于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、UDP 头部的结构分别如下两图所示：



因此，在解析或填充 UDP 头部的时候，需要考虑的字段主要包括有：源端口号、目的端口号、长度、校验和等。

另一方面，在计算 UDP 校验和时，不能直接使用计算 IP 校验和的方法。这是因为，

UDP 校验和需要覆盖 UDP 头部、UDP 数据和一个伪头部。伪头部是在 UDP 头部之前添加了 12 字节的头部信息，只会在计算校验和时添加到 UDP 头部之前，计算完毕后会移除(不会与 UDP 数据报一起被发送出去，也不会被计入 UDP 长度)。伪头部的结构如下图所示：



关于为什么计算校验和还要考虑伪头部，是因为：仅凭目标端口识别某一个通信是远远不够的，通常使用五元组（源 IP 地址、目的 IP 地址、协议号、源端口号、目的端口号）来区分不同的通信。因此，只要有一项遇到损坏，就有可能导致应该收包的应用程序收不到包，不该收到包的应用程序却收到了包。为了避免这一问题，计算校验和时需要引入伪头部。

综上所述，本小节实验的主要任务是：收到数据包时对 UDP 头部进行解析，获得相关信息，然后再去除 UDP 头部；发送数据包时，对 UDP 头部的字段进行填充，然后进行发送。另外，本小节实验还需要再完成 UDP 伪头部校验和计算函数 的设计。

(1) UDP 数据报输出处理函数 `udp_out` 的设计与实现

流程设计如下：

- **【第一步】** 首先调用 `buf_add_header` 函数添加 UDP 报头。
- **【第二步】** 接着，填充 UDP 首部字段。
- **【第三步】** 计算校验和。
- **【第四步】** 调用 `ip_out()` 函数发送 UDP 数据报。

代码实现如下：

```
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 *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 = 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);
}

```

(2) UDP 数据包输入处理函数 `udp_in` 的设计与实现

流程设计如下:

- 【第一步】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。
- 【第二步】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。
- 【第三步】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）。
- 【第四步】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。
- 【第五步】如果能找到，则去掉 UDP 报头，调用处理函数来做相应处理。

代码实现如下:

```

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，
    // 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
    if (buf->len < 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
    udp_hdr_t *hdr = (udp_hdr_t *) buf->data;
    uint16_t checksum_backup = hdr->checksum16;
    hdr->checksum16 = 0;
    uint16_t checksum = udp_checksum(buf, src_ip, net_if_ip);
    if (checksum != checksum_backup) return;
    hdr->checksum16 = 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_port = swap16(hdr->dst_port16);
    udp_handler_t *handler = map_get(&udp_table, &dst_port);
    if (handler == NULL)
    {
        // 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, net_if_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, swap16(hdr->src_port16));
    }
}

```

(3) UDP 校验和计算函数 udp_checksum 的设计与实现

设计流程如下:

- 【第一步】调用 buf_add_header 函数增加 UDP 伪头部, 由于计算长度时不包含伪头部和任何填充的数据, 因此需要预先暂存原有长度。
- 【第二步】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。
- 【第三步】填写 UDP 伪头部的 12 字节字段。
- 【第四步】计算 UDP 校验和。
- 【第五步】再将第二步中暂存的 IP 头部拷贝回来。
- 【第六步】调用 buf_remove_header 函数去掉 UDP 伪头部。

代码实现如下:

```

static uint16_t udp_checksum(buf_t *buf, uint8_t *src_ip, uint8_t *dst_ip)
{
    // TO-DO
    // Step1: 调用 buf_add_header 函数增加 UDP 伪头部
    // 由于计算长度时不包含伪头部和任何填充的数据, 因此需要预先暂存原有长度
    uint16_t len_backup = swap16(buf->len);
    udp_hdr_t *hdr = (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 phdr_backup;
    memcpy(&phdr_backup, buf->data, sizeof(udp_peso_hdr_t));

    // Step3: 填写 UDP 伪头部的 12 字节字段
    udp_peso_hdr_t *phdr = (udp_peso_hdr_t *) buf->data;
    memcpy(phdr->src_ip, src_ip, NET_IP_LEN);
    memcpy(phdr->dst_ip, dst_ip, NET_IP_LEN);
    phdr->placeholder = 0;
    phdr->protocol = NET_PROTOCOL_UDP;
    phdr->total_len16 = len_backup;
}

```

```

// Step4: 计算 UDP 校验和
hdr->checksum16 = 0;
hdr->checksum16 = checksum16((uint16_t *) buf->data, buf->len);

// Step5: 再将 Step2 中暂存的 IP 头部拷贝回来
memcpy(buf->data, &phdr_backup, sizeof(udp_peso_hdr_t));

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

return hdr->checksum16;
}

```

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

(1) 实验结果

通过了与 IP 协议有关的全部测试，截图如下所示：

```

PS D:\Users\ywbishebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest -R ip_test
Test project D:/Users/ywbishebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
  Start 4: ip_test
1/1 Test #4: ip_test ..... Passed    13.77 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 13.78 sec

```

```

PS D:\Users\ywbishebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest -R ip_frag_test
Test project D:/Users/ywbishebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
  Start 5: ip_frag_test
1/1 Test #5: ip_frag_test ..... Passed    0.05 sec

100% tests passed, 0 tests failed out of 1

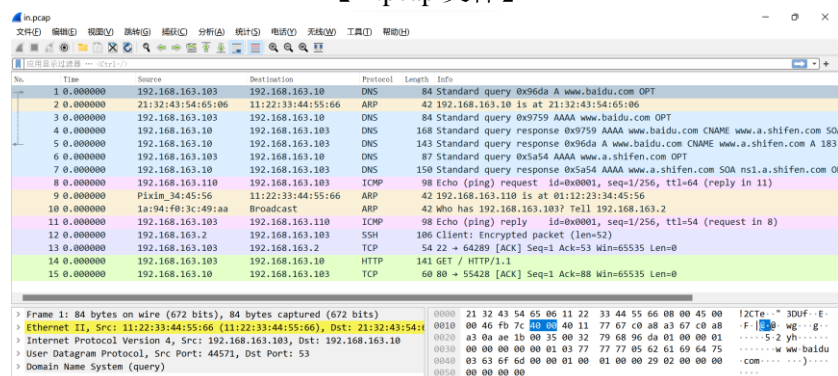
Total Test time (real) = 0.06 sec

```

(2) 结果分析

ip_test 的结果分析：

【in.pcap 文件】

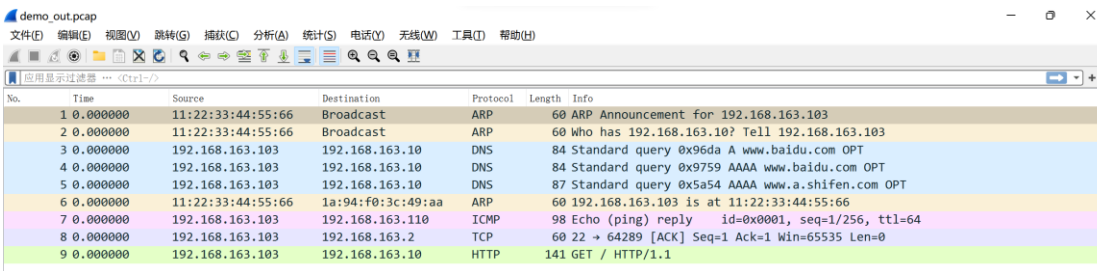


【out.pcap 文件】



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	ARP Announcement for 192.168.163.103
2	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	who has 192.168.163.10? Tell 192.168.163.103
3	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x96da A www.baidu.com OPT
4	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x9759 AAAA www.baidu.com OPT
5	0.000000	192.168.163.103	192.168.163.10	DNS	87	Standard query 0x5a54 AAAA www.a.shifen.com OPT
6	0.000000	11:22:33:44:55:66	1a:94:f0:3c:49:aa	ARP	60	192.168.163.103 is at 11:22:33:44:55:66
7	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64
8	0.000000	192.168.163.103	192.168.163.2	TCP	60	22 → 64289 [ACK] Seq=1 Ack=1 Win=65535 Len=0
9	0.000000	192.168.163.103	192.168.163.10	HTTP	141	GET / HTTP/1.1

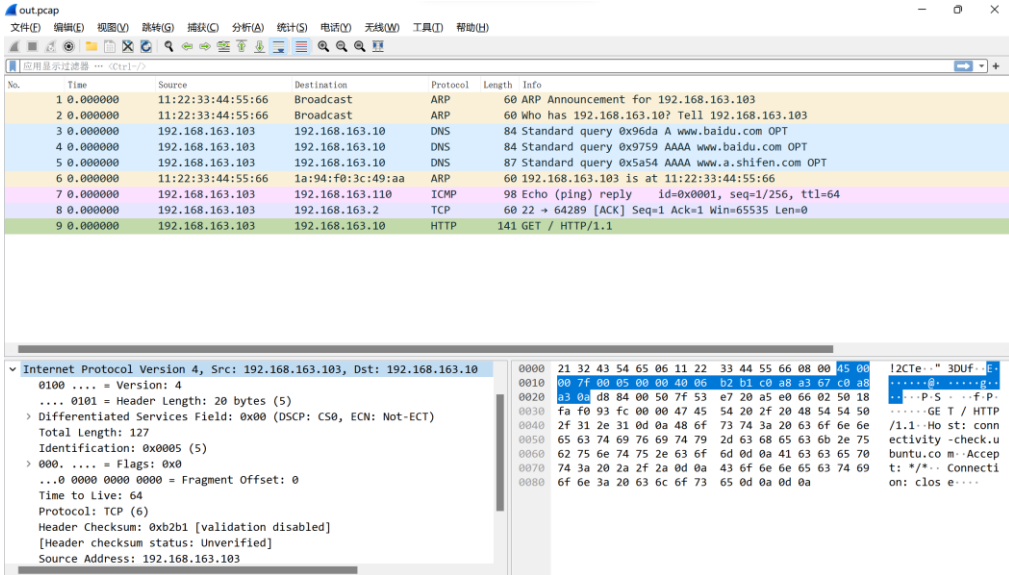
【demo_out.pcap 文件】



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	ARP Announcement for 192.168.163.103
2	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	who has 192.168.163.10? Tell 192.168.163.103
3	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x96da A www.baidu.com OPT
4	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x9759 AAAA www.baidu.com OPT
5	0.000000	192.168.163.103	192.168.163.10	DNS	87	Standard query 0x5a54 AAAA www.a.shifen.com OPT
6	0.000000	11:22:33:44:55:66	1a:94:f0:3c:49:aa	ARP	60	192.168.163.103 is at 11:22:33:44:55:66
7	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64
8	0.000000	192.168.163.103	192.168.163.2	TCP	60	22 → 64289 [ACK] Seq=1 Ack=1 Win=65535 Len=0
9	0.000000	192.168.163.103	192.168.163.10	HTTP	141	GET / HTTP/1.1

文件 out.pcap 与文件 demo_out.pcap 中的内容相一致，故测试得以通过。对比文件 out.pcap 与文件 in.pcap 不难发现，out.pcap 中第 3、4、5 行所期望输出的 DNS 请求都被正确地发送了，并在 in.pcap 中的第 4、5、7 行得到了 DNS 应答。同理，在收到了 in.pcap 第 8 行的 ICMP 请求后，也成功发送了 out.pcap 中第 7 行的 ICMP 应答。再之后的包含有的 SSH、TCP、HTTP 协议的报文也都相应被正确地接收或发送。

选取 out.pcap 中的一个报文（以 HTTP 为例），查看其结构，如下图所示：



The screenshot shows the packet details pane for the selected HTTP packet (No. 9). The left pane shows the protocol hierarchy: Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10. The middle pane shows the expanded details of the IP header, including Version: 4, Header Length: 20 bytes (5), Total Length: 127, Identification: 0x0005 (5), Flags: 0x00, Fragment Offset: 0, Time to Live: 64, Protocol: TCP (6), Header Checksum: 0xb2b1 [validation disabled], and Source Address: 192.168.163.103. The right pane shows the raw packet data in hexadecimal and ASCII. The selected IP header bytes are highlighted in blue: 45 00 00 7f ... a8 a3 0a.

可以观察到，图中被选中的“45 00 00 7f ... a8 a3 0a”部分就是 IP 头部，长度为 20 字节，并能从该 IP 头部中获得版本号为 4、校验和为 0xb2b1 等信息。由此可见，正确实现了 IP 数据包的发送、接收，IP 头部字段的解析、填充，以及头部校验和计算等功能。

综上，ip_test 功能实现正确。

ip_frag_test 的结果分析:

查看 ip_frag_test.c 文件不难发现, 该测试程序并没有检查 pcap 文件, 而是给定了一个包含了较长文本的文件 in.txt, 先将其中的内容读入到 buf 中, 然后将 buf 传递给 ip_out 函数作为输入, 由此检测 ip_out 函数是否能正确输出分片后的结果。

对比可发现 demo_log 和 log 文件的内容完全一致, 因此可以通过测试。故分片功能实现正确。

综上, ip_frag_test 的功能实现正确。

2. ICMP 协议实验结果及分析

(1) 实验结果

通过了与 ICMP 协议有关的全部测试, 截图如下所示:

```
PS D:\Users\ywbishtebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest -R icmp_test
Test project D:/Users/ywbishtebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
Start 6: icmp_test
1/1 Test #6: icmp_test ..... Passed    5.03 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 5.05 sec
```

到此为止, 运行“ctest”命令, 可以发现全部的 6 个测试都已经通过了:

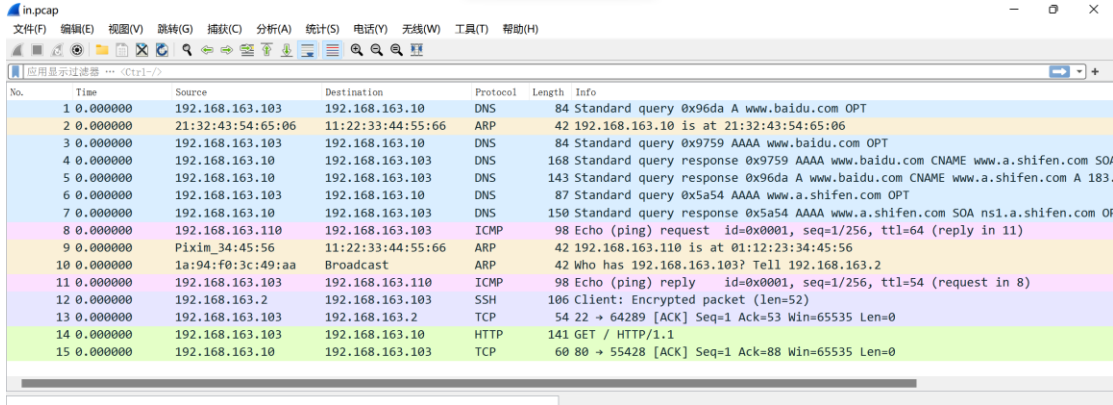
```
PS D:\Users\ywbishtebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest
Test project D:/Users/ywbishtebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
Start 1: eth_in
1/6 Test #1: eth_in ..... Passed    16.11 sec
Start 2: eth_out
2/6 Test #2: eth_out ..... Passed    5.83 sec
Start 3: arp_test
3/6 Test #3: arp_test ..... Passed    5.49 sec
Start 4: ip_test
4/6 Test #4: ip_test ..... Passed    0.11 sec
Start 5: ip_frag_test
5/6 Test #5: ip_frag_test ..... Passed    0.18 sec
Start 6: icmp_test
6/6 Test #6: icmp_test ..... Passed    0.19 sec

100% tests passed, 0 tests failed out of 6

Total Test time (real) = 27.95 sec
```

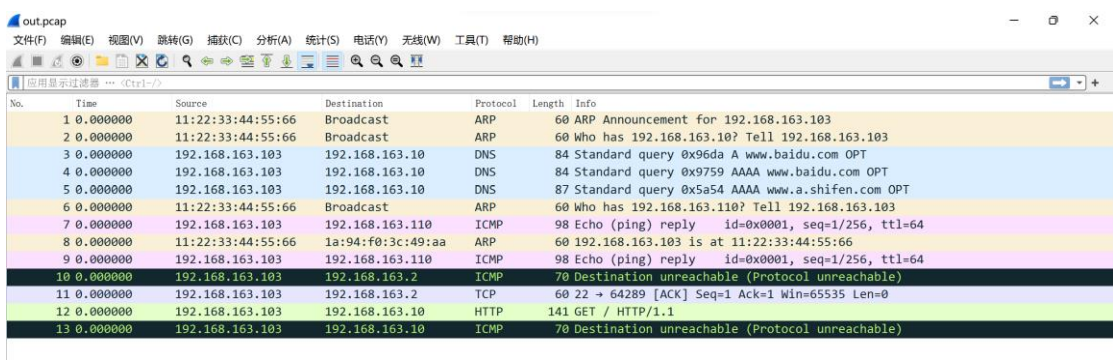
(2) 结果分析

【in.pcap】



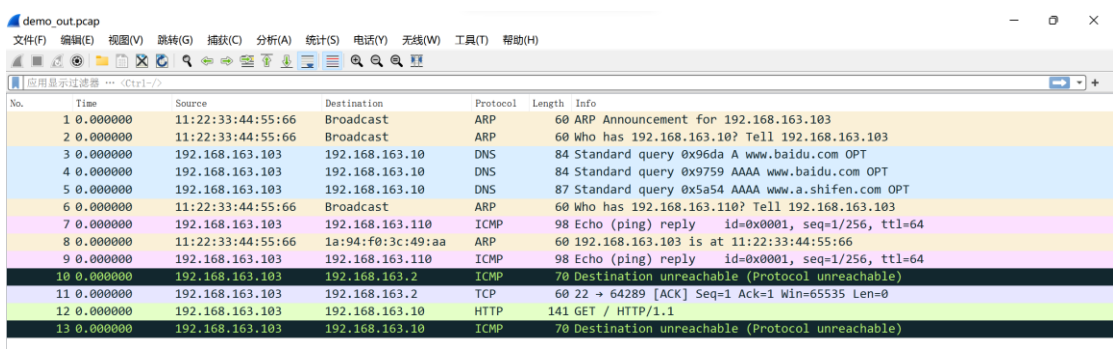
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x96da A www.baidu.com OPT
2	0.000000	21:32:43:54:65:06	11:22:33:44:55:66	ARP	42	192.168.163.10 is at 21:32:43:54:65:06
3	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x9759 AAAA www.baidu.com OPT
4	0.000000	192.168.163.10	192.168.163.103	DNS	168	Standard query response 0x9759 AAAA www.baidu.com CNAME www.a.shifen.com SOA
5	0.000000	192.168.163.10	192.168.163.103	DNS	143	Standard query response 0x96da A www.baidu.com CNAME www.a.shifen.com A 183.
6	0.000000	192.168.163.103	192.168.163.10	DNS	87	Standard query 0x5a54 AAAA www.a.shifen.com OPT
7	0.000000	192.168.163.10	192.168.163.103	DNS	150	Standard query response 0x5a54 AAAA www.a.shifen.com SOA ns1.a.shifen.com OF
8	0.000000	192.168.163.110	192.168.163.103	ICMP	98	Echo (ping) request id=0x0001, seq=1/256, ttl=64 (reply in 11)
9	0.000000	Pixim_34:45:56	11:22:33:44:55:66	ARP	42	192.168.163.110 is at 01:12:23:34:45:56
10	0.000000	1a:94:f0:3c:49:aa	Broadcast	ARP	42	Who has 192.168.163.10? Tell 192.168.163.2
11	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=54 (request in 8)
12	0.000000	192.168.163.2	192.168.163.103	SSH	106	Client: Encrypted packet (len=52)
13	0.000000	192.168.163.103	192.168.163.2	TCP	54	22 → 64289 [ACK] Seq=1 Ack=53 Win=65535 Len=0
14	0.000000	192.168.163.103	192.168.163.10	HTTP	141	GET / HTTP/1.1
15	0.000000	192.168.163.10	192.168.163.103	TCP	60	80 → 55428 [ACK] Seq=1 Ack=88 Win=65535 Len=0

【out.pcap】



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	ARP Announcement for 192.168.163.103
2	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 192.168.163.10? Tell 192.168.163.103
3	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x96da A www.baidu.com OPT
4	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x9759 AAAA www.baidu.com OPT
5	0.000000	192.168.163.103	192.168.163.10	DNS	87	Standard query 0x5a54 AAAA www.a.shifen.com OPT
6	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 192.168.163.110? Tell 192.168.163.103
7	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64
8	0.000000	11:22:33:44:55:66	1a:94:f0:3c:49:aa	ARP	60	192.168.163.103 is at 11:22:33:44:55:66
9	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64
10	0.000000	192.168.163.103	192.168.163.2	ICMP	70	Destination unreachable (Protocol unreachable)
11	0.000000	192.168.163.103	192.168.163.2	TCP	60	22 → 64289 [ACK] Seq=1 Ack=1 Win=65535 Len=0
12	0.000000	192.168.163.103	192.168.163.10	HTTP	141	GET / HTTP/1.1
13	0.000000	192.168.163.103	192.168.163.10	ICMP	70	Destination unreachable (Protocol unreachable)

【demo_out.pcap】



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	ARP Announcement for 192.168.163.103
2	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 192.168.163.10? Tell 192.168.163.103
3	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x96da A www.baidu.com OPT
4	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x9759 AAAA www.baidu.com OPT
5	0.000000	192.168.163.103	192.168.163.10	DNS	87	Standard query 0x5a54 AAAA www.a.shifen.com OPT
6	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 192.168.163.110? Tell 192.168.163.103
7	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64
8	0.000000	11:22:33:44:55:66	1a:94:f0:3c:49:aa	ARP	60	192.168.163.103 is at 11:22:33:44:55:66
9	0.000000	192.168.163.103	192.168.163.110	ICMP	98	Echo (ping) reply id=0x0001, seq=1/256, ttl=64
10	0.000000	192.168.163.103	192.168.163.2	ICMP	70	Destination unreachable (Protocol unreachable)
11	0.000000	192.168.163.103	192.168.163.2	TCP	60	22 → 64289 [ACK] Seq=1 Ack=1 Win=65535 Len=0
12	0.000000	192.168.163.103	192.168.163.10	HTTP	141	GET / HTTP/1.1
13	0.000000	192.168.163.103	192.168.163.10	ICMP	70	Destination unreachable (Protocol unreachable)

文件 out.pcap 和 demo_out.pcap 中的内容相一致，故测试可以通过。观察不难发现，in.pcap 的第 8 行成功接收了一个 ICMP 回显请求、第 11 行成功接收了一个 ICMP 回显应答，out.pcap 的第 7 行、第 9 行均成功发送了 ICMP 回显应答。由此可见，ICMP 的查询/信息报文功能实现正确。

另一方面，由于本课程的协议栈实验还存在部分协议没有实现，故 out.pcap 的第 10 行、第 13 行均成功发送了 ICMP 差错报文（协议不可达），表明此时出现了不能识别的协议类型。由此可见，ICMP 的差错报文功能实现正确。

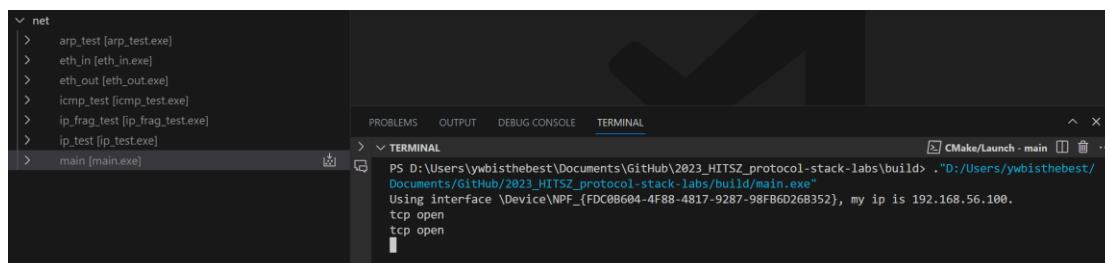
综上，icmp_test 功能实现正确。

3. UDP 协议实验结果及分析

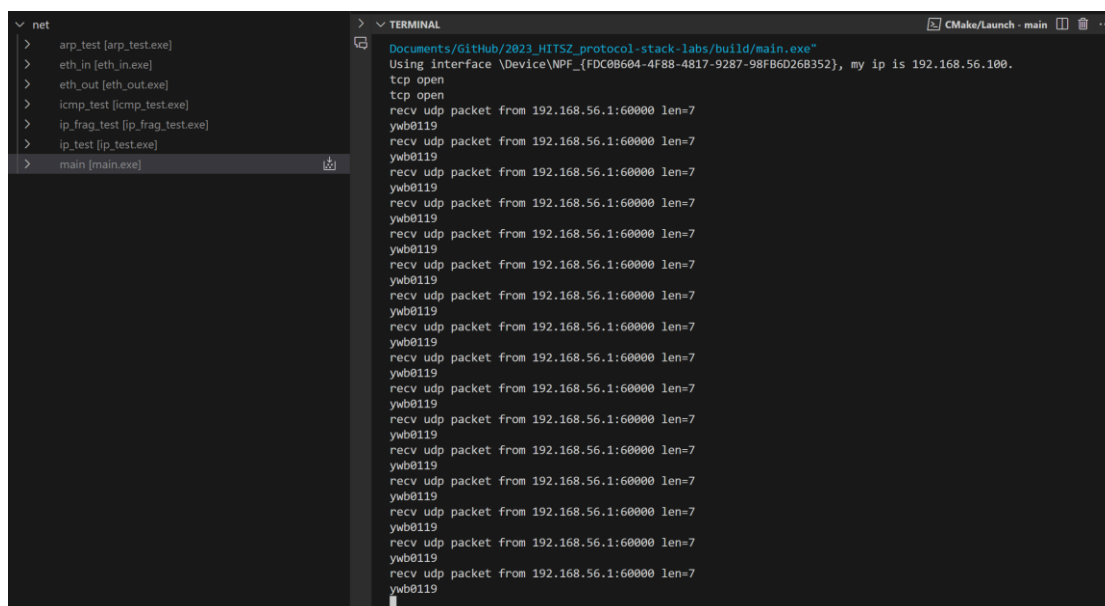
(本小节还需要分析你自己用 Wireshark 抓包工具捕获到的相关报文(包含 UDP 和 ARP 报文), 解析报文内容)

(1) 实验结果

在终端上运行 main, 如下图所示:

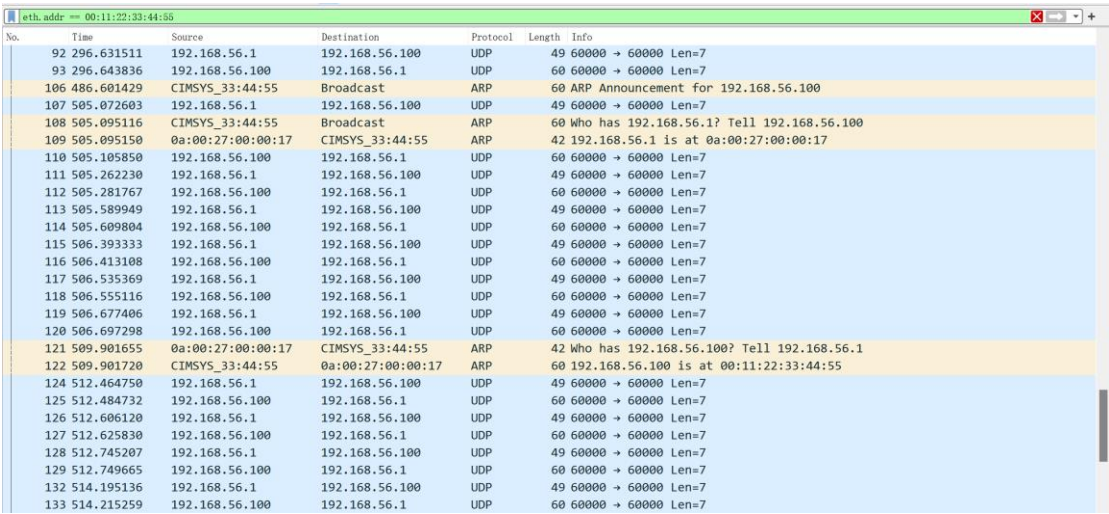


打开 TCP&UDP 测试工具, 建立 UDP 连接, 并发送若干次测试字符串“ywb0119”, 如下图所示:



图示中发送总次数为 $112/7=16$ 次。不难发现, 所发送的 16 个 UDP 数据包均被成功接收到。由此可见 UDP 协议的功能实现正确。

设定过滤条件为“eth.addr == 00:11:22:33:44:55”，其中“00:11:22:33:44:55”为自定义的虚拟网卡 MAC 地址，可以在 wireshark 中观察到相应的报文，如下图所示：

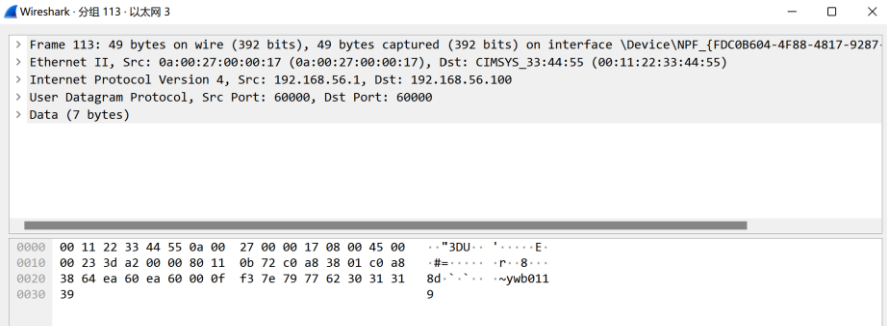


其中，“CIMSYS_33:44:55”为自定义网卡 MAC 地址，即“00:11:22:33:44:55”；“0a:00:27:00:00:17”为本机网卡 MAC 地址，与下图输入了“ipconfig /all”命令后查询到的物理地址相一致；“192.168.56.100”是自定义网卡 IP 地址，“192.168.56.1”则是本机网卡 IP 地址。

```
以太网适配器 以太网 3:
    连接特定的 DNS 后缀 . . . . . :
    描述. . . . . : VirtualBox Host-Only Ethernet Adapter
    物理地址. . . . . : 0A-00-27-00-00-17
```

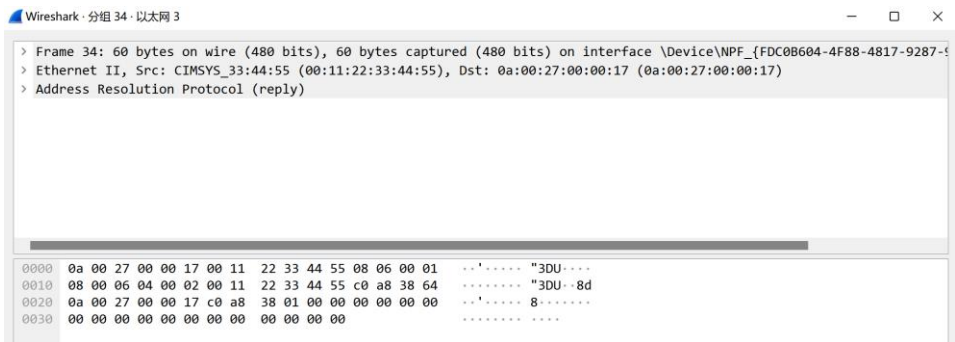
(2) 结果分析

解析捕获到的 UDP 报文，如下所示：(标红部分对应报文中的各个字节，用十六进制表示)



- | | |
|---------------------------------------|------------------------------------|
| 目的 MAC 地址：00 11 22 33 44 55 | 源 MAC 地址：0a 00 27 00 00 17 |
| 协议类型：IP 协议 (08 00) | 协议版本号：IPv4 (4) |
| 首部长度：20 字节 (5) | 区分服务：00 |
| 总长度：35 字节 (00 23) | 标识：3d a2 |
| 标志：00 | 片偏移：00 |
| TTL：128 (80) | 上层协议号：UDP，17 (11) |
| 头部校验和：0b 72 | 源 IP 地址：192.168.56.1 (c0 a8 38 01) |
| 目的 IP 地址：192.168.56.100 (c0 a8 38 64) | 源端口号：60000 (ea 60) |
| 目的端口号：60000 (ea 60) | 长度：15 字节 (00 0f) |
| 校验和：f3 7e | 数据：ywb0119 (79 77 62 30 31 31 39) |

解析捕获到的 ARP 报文，如下所示：（标红部分对应报文中的各个字节，用十六进制表示）



目的 MAC 地址：0a 00 27 00 00 17 源 MAC 地址：00 11 22 33 44 55
协议类型：ARP 协议（08 06） 硬件类型：以太网（00 01）
上层协议类型：IP 协议（08 00） MAC 地址长度：6 字节（06）
IP 地址长度：4 字节（04） 操作类型：ARP 应答报文（00 02）
源 MAC 地址：00 11 22 33 44 55 源 IP 地址：192.168.56.100（c0 a8 38 64）
目的 MAC 地址：0a 00 27 00 00 17 目的 IP 地址：192.168.56.1（c0 a8 38 01）
填充 padding：18 个字节的 0（00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00）

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）

问题 1：校验和的大小端序问题处理错误

一开始对校验和的计算没有进行较为深入的思考，看到了校验和字段的大小为 2 个字节（16 位），觉得和之前实验的做法一样，直接想当然地进行了 swap，结果遇到报错。

网络中的数据是按照大端模式，如果操作系统是小端模式怎么办？是不是应该取出数据之后先做字节反转，加完后再做字节反转换回来呢？事实上，在校验和设计的过

程之中，其步骤就已经考虑到了这一点。

考虑两个数 (高字节低字节由竖线分离开来)

$$x_1, x_2$$
$$x_1 = \alpha_1 | \beta_1$$
$$x_2 = \alpha_2 | \beta_2$$

正常的加法如下

$$x_1 + x_2 = (\alpha_1 + \alpha_2) | (\beta_1 + \beta_2) \quad (1)$$

如果有进位的话

$$x_1 + x_2 = (\alpha_1 + \alpha_2 + k_\beta) \bmod 10000H | (\beta_1 + \beta_2) \bmod 10000H \quad (2)$$

如果我们大小端不匹配，直接进行加法的话

$$x_1 + x_2 = (\beta_1 + \beta_2 + k_\alpha) \bmod 10000H | (\alpha_1 + \alpha_2) \bmod 10000H \quad (3)$$

反转后

$$x_1 + x_2 = (\alpha_1 + \alpha_2) \bmod 10000H | (\beta_1 + \beta_2 + k_\alpha) \bmod 10000H \quad (4)$$

显然 (2) 与 (4) 结果不同

但是校验和有一个机制：十六位的加法结果用三十二位来存储，高十六位的进位最后要加至低位。我们看看这样会发生什么：

正常加法

$$x_1 + x_2 = (\alpha_1 + \alpha_2 + k_\beta) \bmod 10000H | (\beta_1 + \beta_2 + k_\alpha) \bmod 10000H \quad (5)$$

大小端不匹配加法

$$x_1 + x_2 = (\beta_1 + \beta_2 + k_\alpha) \bmod 10000H | (\alpha_1 + \alpha_2 + k_\beta) \bmod 10000H \quad (6)$$

翻转后

$$x_1 + x_2 = (\alpha_1 + \alpha_2 + k_\beta) \bmod 10000H | (\beta_1 + \beta_2 + k_\alpha) \bmod 10000H \quad (7)$$

这就说明，实际上 checksum 无需关心大小端，直接进行加法即可，最后结果一样。

问题 1 解决方法：通过翻看课程群的聊天记录，发现老师和同学们曾经就“校验和是否要考虑字节序”进行了讨论，并了解到上一届的同学已经对该问题背后的数学原理进行了较为深入的挖掘，如上图所示。因此，将代码修改为不考虑校验和的字节序转换，顺利地通过了 IP 协议的测试程序。

问题 2：不理解 UDP 部分指导书 Step2 “暂存 IP 头部”的用意，无从下手实现对应代码在实现 `udp_checksum` 函数的时候，发现指导书的 Step2 如下图所示。

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。

当时很纳闷，这不是已经到传输层了吗，IP 头部不是已经被去除了吗？这上哪里去找 IP 头部啊。由此也不太理解怎么实现对应的代码。

问题 2 解决办法：

通过反复阅读所给的框架代码和相应的 API，发现本次实验所实现的头部去除函数并没有真正去除掉头部，仅仅只是移动了指针，而没有清空相应的内存，如下图所示。

```
int buf_remove_header(buf_t *buf, size_t len)
{
    if (buf->len < len)
    {
        fprintf(stderr, "Error in buf_remove_header:%zu-%zu\n", buf->len, len);
        return -1;
    }
    buf->len -= len;
    buf->data += len;
    return 0;
}
```

也就是说，实际上在 `buf` 的指针前面一小部分区域仍然存储了该 IP 头部，IP 头部只是在“逻辑上”被去除了，在“物理上”并没有被去除。明白这一点后，我的做法是先暂存会被 UDP 伪头部覆盖的那一部分，计算完校验和后再恢复，成功解决了该问题（如下图所示）。

```
// 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 phdr_backup;
memcpy(&phdr_backup, buf->data, sizeof(udp_peso_hdr_t));
```

```
// Step5: 再将 Step2 中暂存的IP头部拷贝回来
memcpy(buf->data, &phdr_backup, sizeof(udp_peso_hdr_t));
```

四、实验收获和建议

（实验过程中的收获、感受、问题、建议等）

收获以及感受：

- 本学期的“协议栈”系列实验让我收获较大。通过完成一系列的迭代开发任务，我对理论课上所传授的 Eth、ARP、IP、ICMP、UDP 等协议有了更深刻的认识。乍一看，仅仅两次实验课就要完成这么多协议的编程任务，似乎比较困难，但事实上，这些协议遵循着相似的“套路”，很大一部分操作都是在围绕着“发送”和“接收”这两个主题进行。在发送时，封装报头，并填充和本层协议有关的字段；在接收时，去除报头，

并对与本层协议有关的字段进行**解析**。在此基础上，各层协议的实现会略有不同。掌握这一套路之后，就会发现其实各层协议的实现往往都是大同小异。

- 该系列实验较多地涉及了对各协议字段（例如，源/目的 IP 地址、源/目的 MAC 地址、校验和、协议号等）的填充或解析操作。通过处理这些协议字段的细节，我在无形之中对各层协议的数据报结构有了更透彻的认识。正所谓“纸上得来终觉浅”，如果仅仅停留在“看”，而没有上手 coding，是很难有这样的深刻认识的。
- 此外，该系列实验的编程工具、配套图文讲解都较为完善，老师甚至录制了演示视频帮助我们快速上手实验框架。这也让我们有了良好的实验体验，减少浪费在环境配置、工具熟悉上的时间，从而得以把更多的精力投入到对实验原理、实验设计的思考上。另外，实验中常用的 wireshark 工具给我带来了很大的帮助。这一工具较好地解释了数据报背后的协议、格式、语义等信息，把数据报的一个个字符和对应字段的含义一一对应起来，展示了各层的数据报是怎么拆解、封装的，让我获得了更生动直观的理解。
- 总的来说，本学期“协议栈”系列实验还是让我收获较大。

问题以及建议：

- 希望能够像以往的《操作系统》、《计算机组成原理》等课程的实验一样，在 piazza 平台上组织相关答疑。这样既能提高答疑效率，同时也方便让更多同学看到有价值的讨论结果（例如前面所讲的计算校验和可以不考虑字节序的数学原理）。