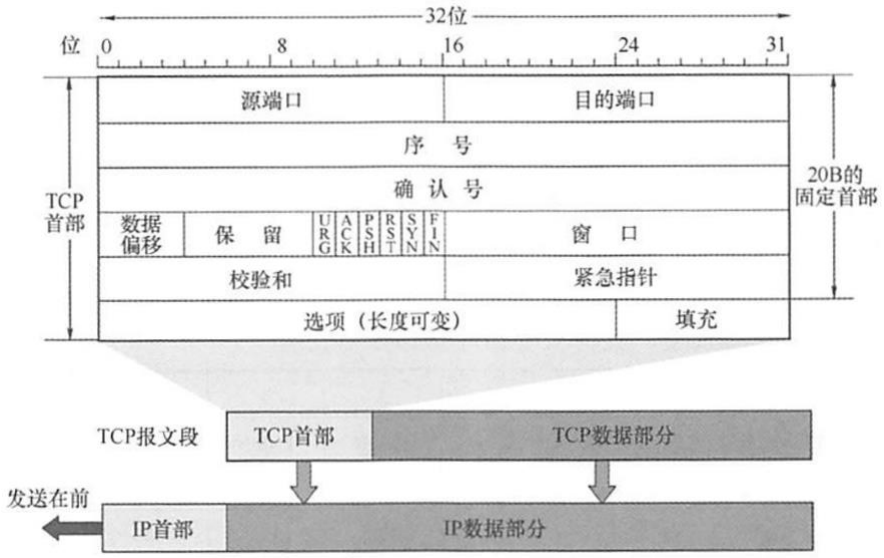
一、 实验详细设计

（一） 整体设计思路

TCP 协议位于**传输层**，是一种面向字节流的、面向连接的协议，提供可靠、按序的交付服务，主要解决传输的**可靠、有序、无丢失、不重复**的问题，适用于**对传输可靠性要求高**的场景。

1. TCP 的**报文段结构**



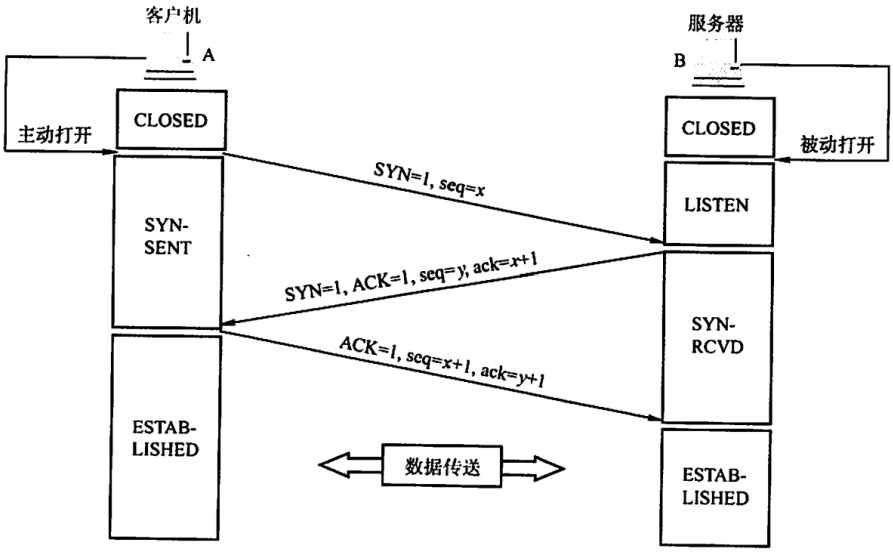
各个字段的**含义**如下：

- 源端口号、目的端口号。两者各占 2B。端口是传输层与应用层的服务接口，传输层的复用和奋勇功能均是通过端口实现。
- 序号。占 4B。TCP 面向字节流，因此要求字节流中的每个字节都按顺序编号。序号字段的值是本报文段所发送数据的第一个字节的序号。
- 确认号。占 4B。确认号字段是期望收到对方下一个报文段的第一个数据字节的序号。若确认号为 N，表明到序号(N-1)为止的所有数据都已经正确收到。
- 数据偏移（即首部长度）。占 4 位。此处不是 IP 数据报分片的那个数据偏移，而是表示首部长度。它指出 TCP 报文段的数据起始处距离 TCP 整个报文段的起始处有多远。数据偏移的单位是 32 位（即以 4B 为计算单位）。
- 保留字段。占 6 位。当前置为 0，供今后使用。
- 紧急位 URG。该位为 1 的时候，表明紧急指针字段有效，可以用于告诉系统此报文中有紧急数据，应当尽快传送、提高其优先级。
- 确认位 ACK。仅当 ACK 为 1 的时候，确认号字段才有效。当 ACK 为 0 的时候，确认号无效。TCP 规定，连接建立后所有传送的报文段都必须把 ACK 置为 1。
- 推送位 PSH。接收方 TCP 收到 PSH 为 1 的报文段，就尽快地交付给接收应用进程，而不再等到整个缓存都填满了后再向上交付。
- 复位位 RST。当 RST 为 1，表明 TCP 连接中出现严重差错，必须释放连接，然后再重新建立运输连接。

- 同步位 SYN。当 SYN 为 1，表明这是一个连接请求或连接接收报文。当 SYN 为 1，ACK 为 0 时，表明这是一个连接请求报文，对方若同意建立连接，则应当在响应报文中使用 SYN=1，ACK=1。
- 终止位 FIN。用来释放一个连接。当 FIN 为 1 的时候，表明此报文的发送方已发送完毕数据，要求释放连接。
- 窗口。占 2B。用于指出现在允许对方发送的数据量。由于接收方的数据缓存空间有限，因此用窗口值作为接收方让发送方设置其发送窗口的依据。
- 校验和。占 2B。检验的范围包括首部和数据两部分。因此和 UDP 一样，TCP 校验和的计算也需要引入伪首部。
- 紧急指针。占 2B。仅当 URG=1 的时候才有意义，用于指出本报文段中紧急数据一共有多少字节（紧急数据在报文段数据的最前面）。
- 选项。长度可变。
- 填充。用于使整个首部的长度是 4B 的整数倍。

2. TCP 连接的**建立**

TCP 连接的建立经历 **3 个步骤**，通常称为**三次握手**，如下图所示。



建立连接前，服务器处于 **LISTEN 状态（收听）**，等待客户机的连接请求。

第一步：客户机发送**连接请求报文段**。该报文的 SYN 为 1，同时选择一个初始序号 seq=x。发送该报文的同时，客户机进程进入 **SYN-SENT 状态（同步已发送）**。

第二步：服务器收到该**连接请求报文段**后，如果同意建立连接，则向客户机发送**确认报文段**。该报文的 SYN 为 1，ACK 为 1，确认号 ack=x+1，同时也选择一个初始序号 seq=y。发送该报文的同时，服务器进程进入 **SYN-RCVD 状态（同步收到）**。

第三步：客户机收到**确认报文段**后，还需要再向服务器给出**确认**。该报文的 ACK 为 1，确认号 ack=y+1，序号 seq=x+1。此时，客户端进行进入 **ESTABLISHED 状态（已建立连接）**。

成功进行以上三步后，就建立了 TCP 连接，接下来双方就可以传送数据。

3. TCP 连接的**释放**

TCP 连接的释放经历 **4 个步骤**，通常称为**四次握手**。

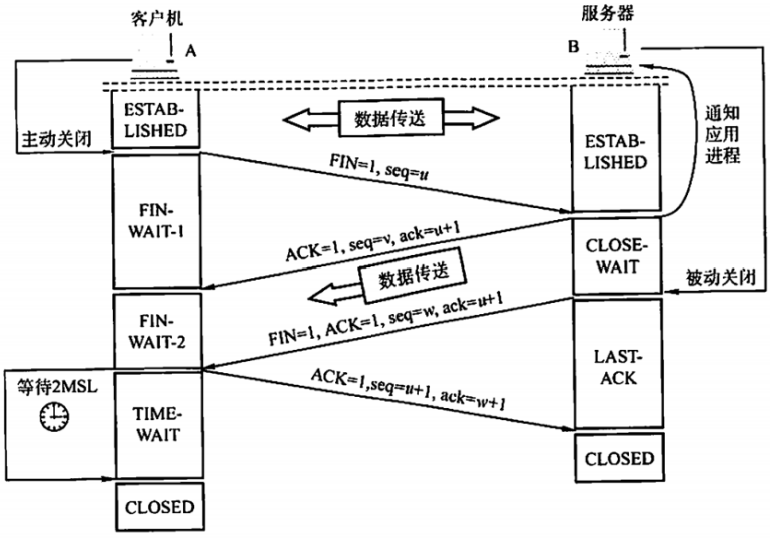
第一步：客户机希望关闭连接的时候，发送**连接释放报文段**，并停止发送数据。该报文的 FIN 为 1，序号 seq=u（前面已传送过的数据的最后一个字节序号加 1）。同时，客户机进程进入 **FIN-WAIT-1 状态（终止等待 1）**。

第二步：服务器接收到**连接释放报文段**后，发出**确认**。确认号 ack 为 u+1，序号 seq=v（前面已传送过的数据的最后一个字节的序号加 1）。同时，服务器进程进入 **CLOSE-WAIT 状态（关闭等待）**。

第三步：若服务器已没有要向客户机发送的数据，通知其释放连接，发送**连接释放报文段**。该报文 FIN 为 1，序号为 w（半关闭状态下服务器可能又发送了一些数据），确认号 ack=u+1。同时，服务器进入 **LAST-ACK 状态（最后确认）**。

第四步：客户机收到连接释放报文段后，发出**确认**。确认报文段的 ACK 为 1，确认号 ack=w+1，序号 seq=u+1。经过时间等待器设置的时间 2MSL 后，客户机才进入 **CLOSED 状态（连接关闭）**。

上述过程如下图所示：



4. 总体设计思路

本次实验**聚焦于 TCP 协议的实现**。所给的模板代码中已经有一个 **TCP 协议框架**。该框架较为完善，已经包含了 tcp_init、tcp_open、tcp_close、tcp_checksum、new_tcp_key 等函数。故本次实验只要求我们补充完整 tcp.c 文件中的 tcp_in 函数，即完成与**服务器端 TCP 收包**有关功能的实现。

结合前 3 小点介绍的与 TCP 协议有关的内容，我们需要实现的部分包括：完成对 TCP 报头的端口号、校验和、序列号、确认号、flags 等字段的解析，并实现**各连接建立状态（包括 TCP_LISTEN、TCP_SYN_RCVD、TCP_ESTABLISHED）、各连接释放状态（包括 TCP_CLOSE_WAIT、TCP_FIN_WAIT_1、TCP_FIN_WAIT_2、TCP_LAST_ACK）**之间的相互转换，并实现好在各个状态下应当完成的操作。

以上便是对本次实验任务整体设计思路的介绍，接下来将会详细分析各部分细节的设计与实现。

（二） 各部分详细设计与实现

1. 进行大小检查，检查 buf 长度是否小于 tcp 头部，如果是，则丢弃。

```
if (buf->len < sizeof(tcp_hdr_t)) return;
```

2. 检查 checksum 字段，如果 checksum 出错，则丢弃。

```
tcp_hdr_t* hdr = (tcp_hdr_t *) buf->data;
uint16_t checksum_backup = hdr->checksum16;
hdr->checksum16 = 0;
uint16_t checksum = tcp_checksum(buf, src_ip, net_if_ip);
if (checksum_backup != checksum) return;
hdr->checksum16 = checksum_backup;
```

3. 从 tcp 头部字段中获取并解析字段 source port、destination port、sequence number、acknowledge number、flags 等信息。注意，部分字段需要进行适当的大小端转换。

```
uint16_t src_port = swap16(hdr->src_port16);
uint16_t dst_port = swap16(hdr->dst_port16);
uint32_t seq_number = swap32(hdr->seq_number32);
uint32_t ack_number = swap32(hdr->ack_number32);
tcp_flags_t flags = hdr->flags;
uint32_t get_seq = seq_number;
```

4. 调用 map_get 函数，根据 destination port 查找对应的 handler 函数。

```
tcp_handler_t* handler = map_get(&tcp_table, &dst_port);
```

5. 调用 new_tcp_key 函数，根据通信五元组中的源 IP 地址、目标 IP 地址、目标端口号确定一个 tcp 链接 key。

```
tcp_key_t tcp_key = new_tcp_key(src_ip, src_port, dst_port);
```

6. 调用 map_get 函数，根据 key 查找一个 tcp_connect_t* connect，如果没有找到，则调用 map_set 建立新的链接，并设置为 CONNECT_LISTEN 状态，然后调用 mag_get 获取到该连接。

```
tcp_connect_t* connect = map_get(&connect_table, &tcp_key);
if (connect == NULL)
{
    tcp_connect_t new_connect;
    new_connect.state = TCP_LISTEN;
    map_set(&connect_table, &tcp_key, &new_connect);
    connect = map_get(&connect_table, &tcp_key);
}
```

7. 从 TCP 头部字段中获取对方的窗口大小，注意大小端转换。

```
uint16_t window_size = swap16(hdr->window_size16);
```

8. 如果为 TCP_LISTEN 状态，则需要完成如下功能：
（1） 如果收到的 flag 带有 rst，则 close_tcp 关闭 tcp 链接。
（2） 如果收到的 flag 不是 syn，则 reset_tcp 复位通知。因为收到的第一个包必须是 syn。
（3） 调用 init_tcp_connect_rcvd 函数，初始化 connect，将状态设为 TCP_SYN_RCVD。
（4） 填充 connect 字段，包括：local_port、remote_port、ip、unack_seq（设为随机值）。由于是对 syn 的 ack 应答包，next_seq 与 unack_seq 一致，ack 设为对方的 sequence number 加上 1。设置 remote_win 为对方的窗口大小，注意大小端转换。
（5） 调用 buf_init 初始化 txbuf。
（6） 调用 tcp_send 将 txbuf 发送出去，也就是回复一个 tcp_flags_ack_syn（SYN+ACK）报文。
（7） 处理结束，返回。

```
if (connect->state == TCP_LISTEN)
{
    if (flags.rst) goto close_tcp;
    if (!flags.syn) goto reset_tcp;
    init_tcp_connect_rcvd(connect);
    connect->local_port = dst_port;
    connect->remote_port = src_port;
    memcpy(connect->ip, src_ip, NET_IP_LEN);
    srand((unsigned) time(NULL));
    connect->unack_seq = rand();            // 设为随机值
    connect->next_seq = connect->unack_seq; // 设为与 unack_seq 相同的随机值
    connect->ack = seq_number + 1;
    connect->remote_win = window_size;
    buf_init(connect->tx_buf, 0);
    tcp_send(connect->tx_buf, connect, tcp_flags_ack_syn);
    return;
}
```

9. 检查接收到的 sequence number。如果与 ack 序号不一致，则跳转至 reset_tcp，进行复位通知。

```
if (seq_number != connect->ack) goto reset_tcp;
```

10. 检查 flags 是否有 rst 标志。如果有，则跳转至 close_tcp，进行连接重置。

```
if (flags.rst) goto close_tcp;
```

11. 序号一致时，先调用 buf_remove_header 去除头部（剩下的都是数据）。然后开始进行状态转换。

```
buf_remove_header(buf, 4 * ((uint16_t) hdr->data_offset));
```

```
    /* 状态转换
    */
    switch (connect->state) {
    case TCP_LISTEN:
        panic("switch TCP_LISTEN", __LINE__);
        break;
```

12. 在 RCVD 状态，如果收到的包没有 ack flag，则不做任何处理。

```
    case TCP_SYN_RCVD:
        if (!flags.ack) break;
```

13. 否则，如果是 ack 包，需要完成如下功能：
（1） 将 unack_seq +1。
（2） 将状态转成 ESTABLISHED。
（3） 调用回调函数，完成三次握手，进入连接状态 TCP_CONN_CONNECTED。

```
        connect->unack_seq++;
        connect->state = TCP_ESTABLISHED;
        (* handler)(connect, TCP_CONN_CONNECTED);
        break;
```

14. 在 TCP_ESTABLISHED 状态，如果收到的包没有 ack 且没有 fin 这两个标志，则不做任何处理。

```
    case TCP_ESTABLISHED:
        if ((!flags.ack) && (!flags.fin)) break;
```

15. 否则，先处理 ACK 的值。如果是 ack 包，且 unack_seq 小于 sequence number（说明有部分数据被对端接收确认了，否则可能是之前重发的 ack，可以不处理），且 next_seq 大于 sequence number，则调用 buf_remove_header 函数，去掉被对端接收确认的部分数据，并更新 unack_seq 值。

```
        if ((flags.ack) && (connect->unack_seq < ack_number))
        {
            buf_remove_header(connect->tx_buf, ack_number -
connect->unack_seq);
            connect->unack_seq = ack_number;
        }
```

16. 然后接收数据，调用 tcp_read_from_buf 函数，把 buf 放入 rx_buf 中。

```
        uint16_t read_buf_len = tcp_read_from_buf(connect, buf);
```

17. 再然后，根据当前的标志位进一步处理：
（1） 首先调用 buf_init 初始化 txbuf。
（2） 判断是否收到关闭请求（FIN）。如果是，将状态改为 TCP_LAST_ACK，把 ack 加 1，再发送一个 ACK + FIN 包，并退出，这样就无需进入 CLOSE_WAIT，直接等待对方的 ACK。

（3） 如果不是 FIN，则看看是否有数据。如果有，则发 ACK 响应，并调用 handler 回调函数进行处理。

（4） 调用 tcp_write_to_buf 函数，看看是否有数据需要发送。如果有，同时发数据和 ACK。

（5） 没有收到数据，可能对方只发一个 ACK，可以不响应。

```
        buf_init(&txbuf, 0);
        buf_init(connect->tx_buf, 0);
        if (flags.fin)
        {
            connect->state = TCP_LAST_ACK;
            connect->ack++;
            tcp_send(connect->tx_buf, connect, tcp_flags_ack_fin);
        }
        else
        {
            if (read_buf_len > 0)
            {
                (* handler)(connect, TCP_CONN_DATA_RECV);
                tcp_send(&txbuf, connect, tcp_flags_ack);
            }
            uint16_t write_buf_len = tcp_write_to_buf(connect, &txbuf);
            if (write_buf_len > 0)
            {
                tcp_send(&txbuf, connect, tcp_flags_ack);
            }
        }
        break;

    case TCP_CLOSE_WAIT:
        panic("switch TCP_CLOSE_WAIT", __LINE__);
        break;
```

18. 对于 TCP_FIN_WAIT_1 状态，如果收到 FIN && ACK，则 close_tcp 直接关闭 TCP；如果只收到 ACK，则将状态转为 TCP_FIN_WAIT_2。

```
    case TCP_FIN_WAIT_1:
        if (flags.fin && flags.ack) goto close_tcp;
        if (flags.ack) connect->state = TCP_FIN_WAIT_2;
        break;
```

19. 对于 TCP_FIN_WAIT_2 状态，如果不是 FIN，则不做处理；如果是，则将 ACK +1，调用 buf_init 初始化 txbuf，调用 tcp_send 发送一个 ACK 数据包，再 close_tcp 关闭 TCP。

```
    case TCP_FIN_WAIT_2:
        if (flags.fin)
```

```
        {
            connect->ack++;
            buf_init(connect->tx_buf, 0);
            tcp_send(connect->tx_buf, connect, tcp_flags_ack);
            goto close_tcp;
        }
        break;
```

20. 对于 TCP_LAST_ACK 状态，如果不是 ACK，则不做处理；如果是，则调用 handler 函数，进入 TCP_CONN_CLOSED 状态，，再 close_tcp 关闭 TCP。

```
    case TCP_LAST_ACK:
        if (flags.ack)
        {
            (* handler)(connect, TCP_CONN_CLOSED);
            goto close_tcp;
        }
        break;
```

21. 最后，定义好 default 分支的行为，结束整个状态转换部分并返回。对于 reset_tcp 代码段，将 next_seq 设置为 0、ack 设置为 get_req 加 1，初始化 txbuf 后将其发送出去。对于 close_tcp 代码段，调用 release_tcp_connect 函数释放当前连接，并调用 map_delete 函数从 connect_table 中删除掉该连接。

```
    default:
        panic("connect->state", __LINE__);
        break;
    }
    return;

reset_tcp:
    printf("!!! reset tcp !!!\n");
    connect->next_seq = 0;
    connect->ack = get_seq + 1;
    buf_init(&txbuf, 0);
    tcp_send(&txbuf, connect, tcp_flags_ack_rst);
close_tcp:
    release_tcp_connect(connect);
    map_delete(&connect_table, &tcp_key);
    return;
```

二、 实验结果截图及分析

（1） 完成上述代码的实现后，终端下运行 main，如下图所示：

```
PS D:\Users\ywbisthebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ."D:/Users/ywbisthebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build/main.exe"
Using interface \Device\NPF_{FDC0B604-4F88-4817-9287-98FB6D26B352}, my ip is 192.168.56.100.
tcp open
tcp open
```

（2） 打开 TCP&UDP 测试工具，建立 TCP 连接，并发送若干测试字符串“200110119TCP”，如下图所示：

```
TERMINAL                                                    CMake/Launch - main
Using interface \Device\NPF_{FDC0B604-4F88-4817-9287-98FB6D26B352}, my ip is 192.168.56.100.
tcp open
tcp open
flags: ack syn
recv tcp packet from 192.168.56.1:64985 len=0

recv tcp packet from 192.168.56.1:64985 len=12
200110119TCP
flags: ack
flags: ack
recv tcp packet from 192.168.56.1:64985 len=12
200110119TCP
flags: ack
flags: ack
recv tcp packet from 192.168.56.1:64985 len=12
200110119TCP
flags: ack
flags: ack
recv tcp packet from 192.168.56.1:64985 len=12
200110119TCP
flags: ack
flags: ack
recv tcp packet from 192.168.56.1:64985 len=12
200110119TCP
flags: ack
flags: ack
recv tcp packet from 192.168.56.1:64985 len=12
200110119TCP
flags: ack
flags: ack
flags: ack fin
recv tcp packet from 192.168.56.1:64985 len=0
```

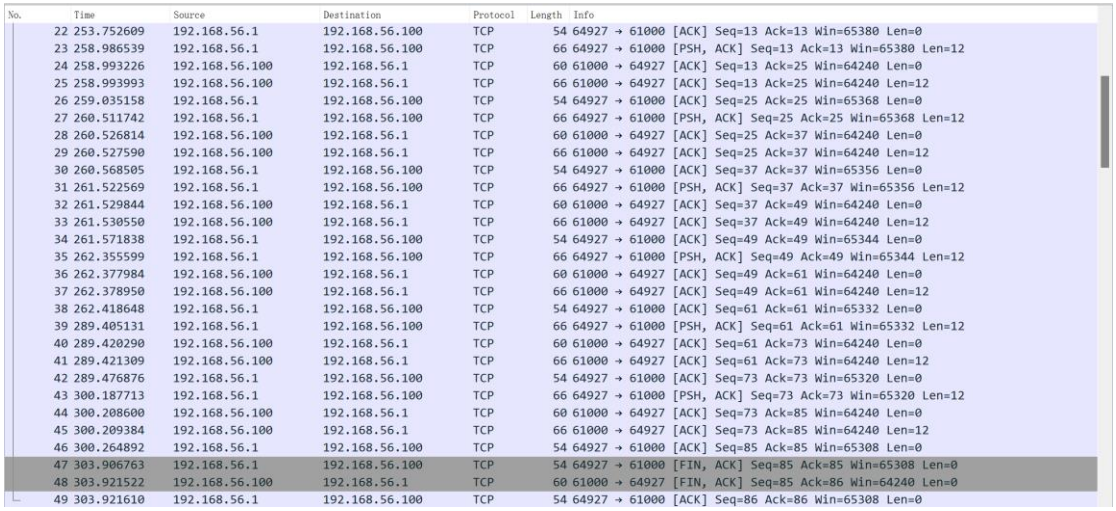
从以上两图不难看出。一开始我们使用测试工具请求建立 TCP 连接，此时终端处打印出了**“recv tcp packet from 192.168.56.1:64985 len=0”**，表明虚拟网卡接收到了 len 为 0 的连接请求报文。

成功建立连接后，我们一共发送了 **84/12=7 次**测试字符串**“200110119TCP”**。这 7 个测试字符串均能被虚拟网卡成功接收到，并被打印在终端上。同时，测试工具也能接收到**虚拟网卡回送的数据包**。

最后，我们点击测试工具的“断开连接”按钮，请求终止当前的 TCP 连接。此时终端处打印出了**“flags: ack fin”**、**“recv tcp packet from 192.168.56.1:64985 len=0”**，表明虚拟网卡接收到了 len 为 0 的连接释放报文，随后成功释放了当前 TCP 连接。

（3） Wireshark 上也能捕获到上述过程的报文，如下图所示：

No.	Time	Source	Destination	Protocol	Length	Info
12	228.689135	192.168.56.1	192.168.56.100	TCP	74	64927 → 61000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM TSval=
13	228.706827	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0
14	228.706961	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=1 Ack=1 Win=65392 Len=0
19	253.689678	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=1 Ack=1 Win=65392 Len=12
20	253.711934	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=1 Ack=13 Win=64240 Len=0
21	253.712641	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=1 Ack=13 Win=64240 Len=12
22	253.752609	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=13 Ack=13 Win=65380 Len=0
23	258.986539	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=13 Ack=13 Win=65380 Len=12
24	258.993226	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=13 Ack=25 Win=64240 Len=0
25	258.993993	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=13 Ack=25 Win=64240 Len=12
26	259.035158	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=25 Ack=25 Win=65368 Len=0
27	260.511742	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=25 Ack=25 Win=65368 Len=12
28	260.526814	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=25 Ack=37 Win=64240 Len=0
29	260.527590	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=25 Ack=37 Win=64240 Len=12
30	260.568505	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=37 Ack=37 Win=65356 Len=0
31	261.522569	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=37 Ack=37 Win=65356 Len=12
32	261.529844	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=37 Ack=49 Win=64240 Len=0
33	261.530550	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=37 Ack=49 Win=64240 Len=12
34	261.571838	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=49 Ack=49 Win=65344 Len=0
35	262.355599	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=49 Ack=49 Win=65344 Len=12
36	262.377984	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=49 Ack=61 Win=64240 Len=0
37	262.378950	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=49 Ack=61 Win=64240 Len=12
38	262.418648	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=61 Ack=61 Win=65332 Len=0
39	289.405131	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=61 Ack=61 Win=65332 Len=12
40	289.420290	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=61 Ack=73 Win=64240 Len=0
41	289.421309	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=61 Ack=73 Win=64240 Len=12
42	289.476876	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=73 Ack=73 Win=65320 Len=0
43	300.187713	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=73 Ack=73 Win=65320 Len=12
44	300.208600	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=73 Ack=85 Win=64240 Len=0



No.	Time	Source	Destination	Protocol	Length	Info
22	253.752609	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=13 Ack=13 Win=65380 Len=0
23	258.986539	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=13 Ack=13 Win=65380 Len=12
24	258.993226	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=13 Ack=25 Win=64240 Len=0
25	258.993993	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=13 Ack=25 Win=64240 Len=12
26	259.035158	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=25 Ack=25 Win=65368 Len=0
27	260.511742	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=25 Ack=25 Win=65368 Len=12
28	260.526814	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=25 Ack=37 Win=64240 Len=0
29	260.527590	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=25 Ack=37 Win=64240 Len=12
30	260.568505	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=37 Ack=37 Win=65356 Len=0
31	261.522569	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=37 Ack=37 Win=65356 Len=12
32	261.529844	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=37 Ack=49 Win=64240 Len=0
33	261.530550	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=37 Ack=49 Win=64240 Len=12
34	261.571838	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=49 Ack=49 Win=65344 Len=0
35	262.355599	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=49 Ack=49 Win=65344 Len=12
36	262.377984	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=49 Ack=61 Win=64240 Len=0
37	262.378950	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=49 Ack=61 Win=64240 Len=12
38	262.418648	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=61 Ack=61 Win=65332 Len=0
39	289.405131	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=61 Ack=61 Win=65332 Len=12
40	289.420290	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=61 Ack=73 Win=64240 Len=0
41	289.421309	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=61 Ack=73 Win=64240 Len=12
42	289.476876	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=73 Ack=73 Win=65320 Len=0
43	300.187713	192.168.56.1	192.168.56.100	TCP	66	64927 → 61000 [PSH, ACK] Seq=73 Ack=73 Win=65320 Len=12
44	300.208600	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [ACK] Seq=73 Ack=85 Win=64240 Len=0
45	300.209384	192.168.56.100	192.168.56.1	TCP	66	61000 → 64927 [ACK] Seq=73 Ack=85 Win=64240 Len=12
46	300.264892	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=85 Ack=85 Win=65308 Len=0
47	303.906763	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [FIN, ACK] Seq=85 Ack=85 Win=65308 Len=0
48	303.921522	192.168.56.100	192.168.56.1	TCP	60	61000 → 64927 [FIN, ACK] Seq=85 Ack=86 Win=64240 Len=0
49	303.921610	192.168.56.1	192.168.56.100	TCP	54	64927 → 61000 [ACK] Seq=86 Ack=86 Win=65308 Len=0

其中，IP 地址为 192.168.56.1、端口号为 64927 的一方为本机真实网卡；IP 地址为 192.168.56.100、端口号为 61000 的一方为虚拟网卡。

主要交互过程如下：

- 12 号数据报，本机真实网卡向虚拟网卡发送 SYN=1，seq=0，请求建立连接。
- 13 号数据报，虚拟网卡向本机真实网卡发送 SYN=1，ACK=1，seq=0，ack=1，对 12 号数据报的请求进行确认。
- 14 号数据报，本机真实网卡向虚拟网卡发送 ACK=1，seq=1，ack=1，进行再次确认。到此为止，成功完成了三次 TCP 握手，建立起 TCP 连接。
- 19 号数据报至 46 号数据报，均是本机真实网卡先向虚拟网卡发送含有 12 个字节的测试字符串“200110119TCP”的数据报，然后虚拟网卡再向本机真实网卡发出应答。由于一共发送了 7 次测试字符串，这一过程重复进行 7 次。
- 47 号数据报，本机真实网卡向虚拟网卡发送 FIN=1，ACK=1，seq=85，ack=85，请求释放报文（由于点击了“断开连接”按钮）。
- 48 号数据报，虚拟网卡向本机真实网卡发送 FIN=1，ACK=1，seq=85，ack=86，对 47 号数据报的释放请求进行确认。
- 49 号数据报，本机真实网卡向虚拟网卡发送 ACK=1，seq=86，ack=86，再次对连接释放请求进行确认。

综上所述，本次实验的 TCP 协议功能实现正确。