



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

# 实验报告

开课学期: 2025 春季

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: 计科 7 班

学生学号: 220110729

学生姓名: 黄楚涵

评阅教师: \_\_\_\_\_

报告成绩: \_\_\_\_\_

实验与创新实践教育中心制

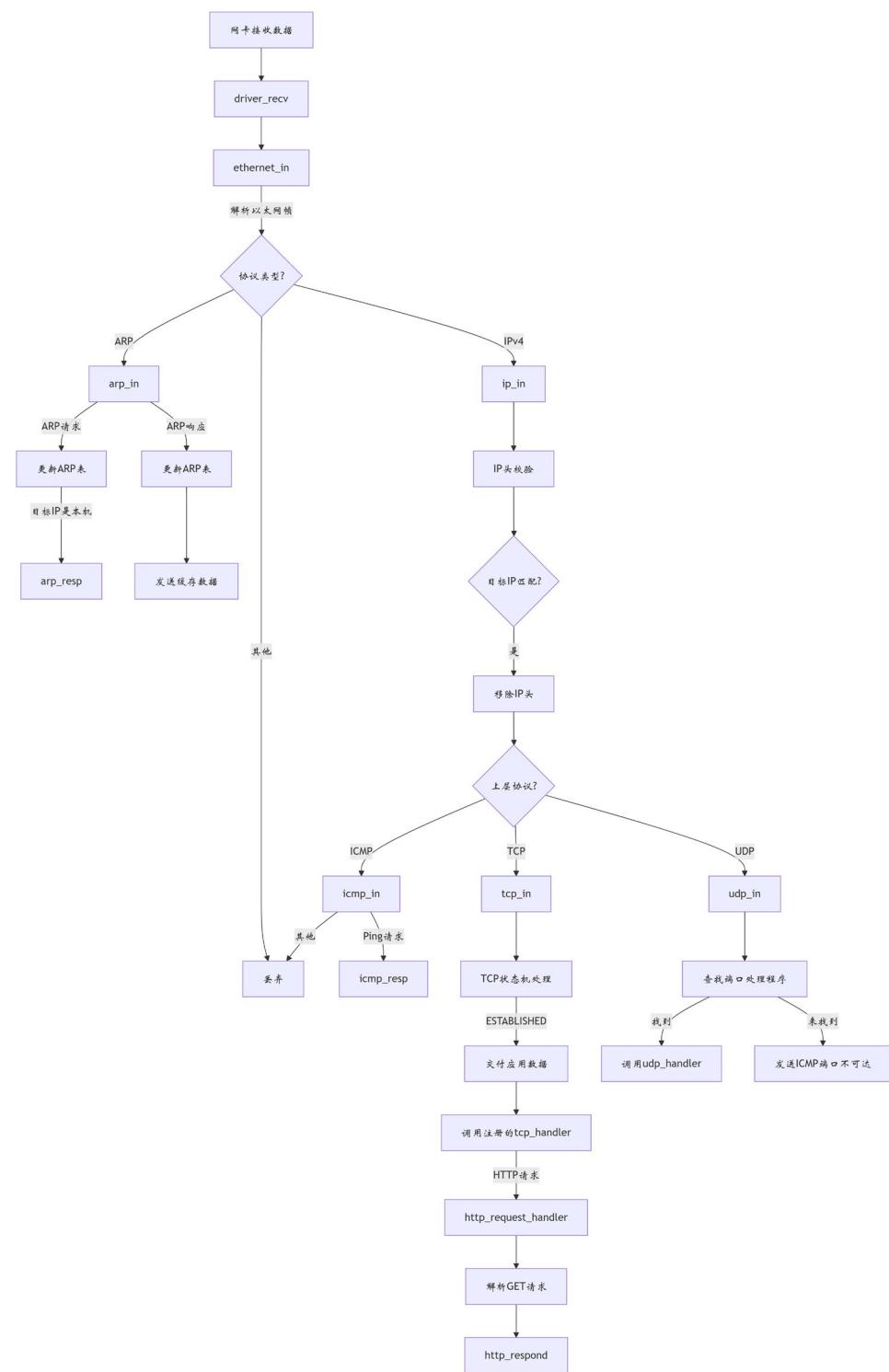
2025 年 3 月

# 一、协议实现详述

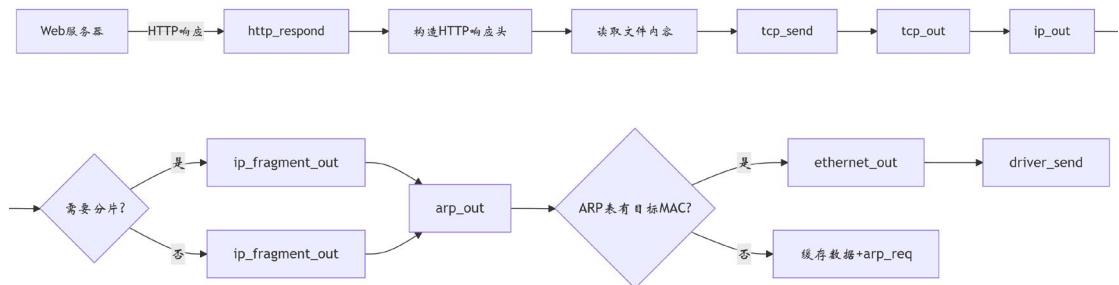
(注意不要完全照搬实验指导书上的内容,请根据你自己的设计方案来填写  
图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。)

## 1. 请给出协议栈实验的整体流程图

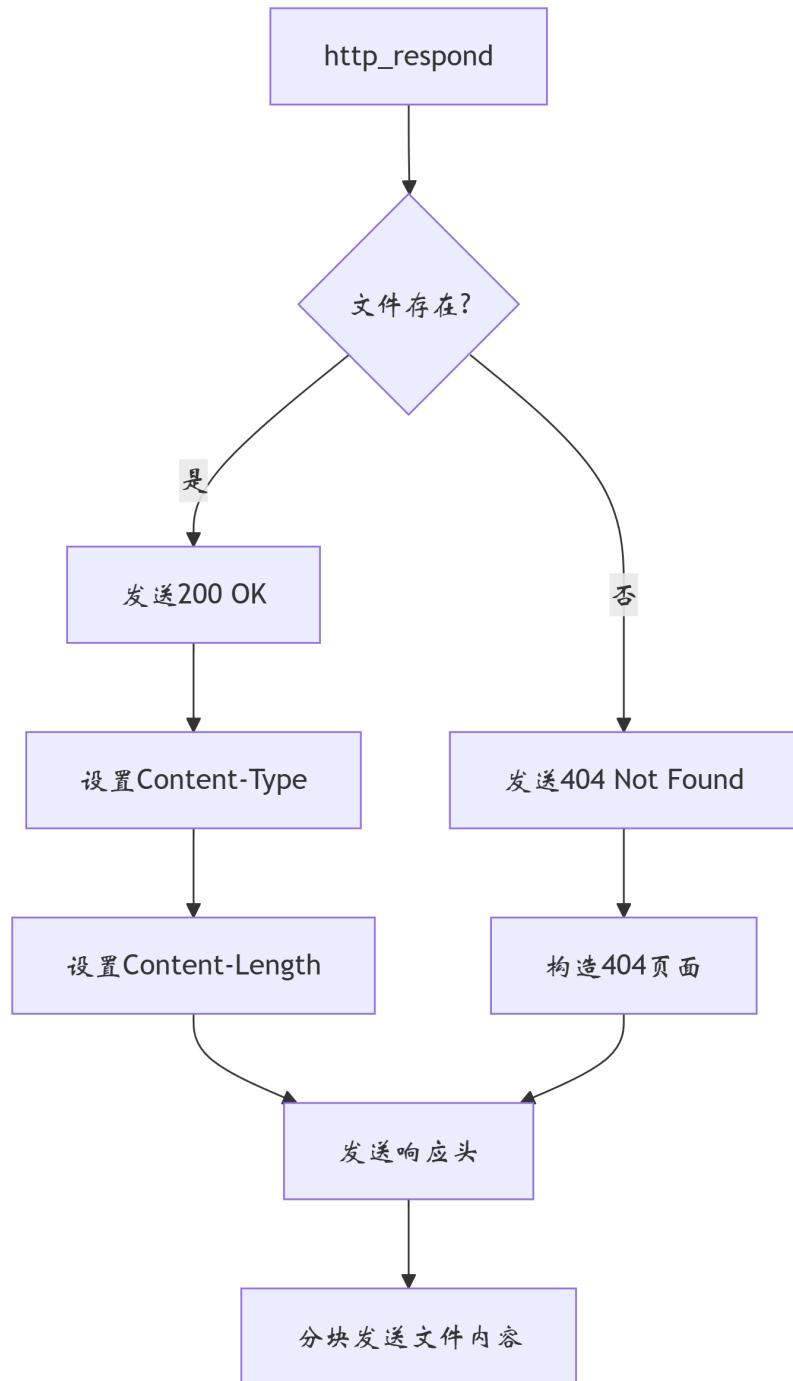
接收方向 (数据包从网卡到应用层)



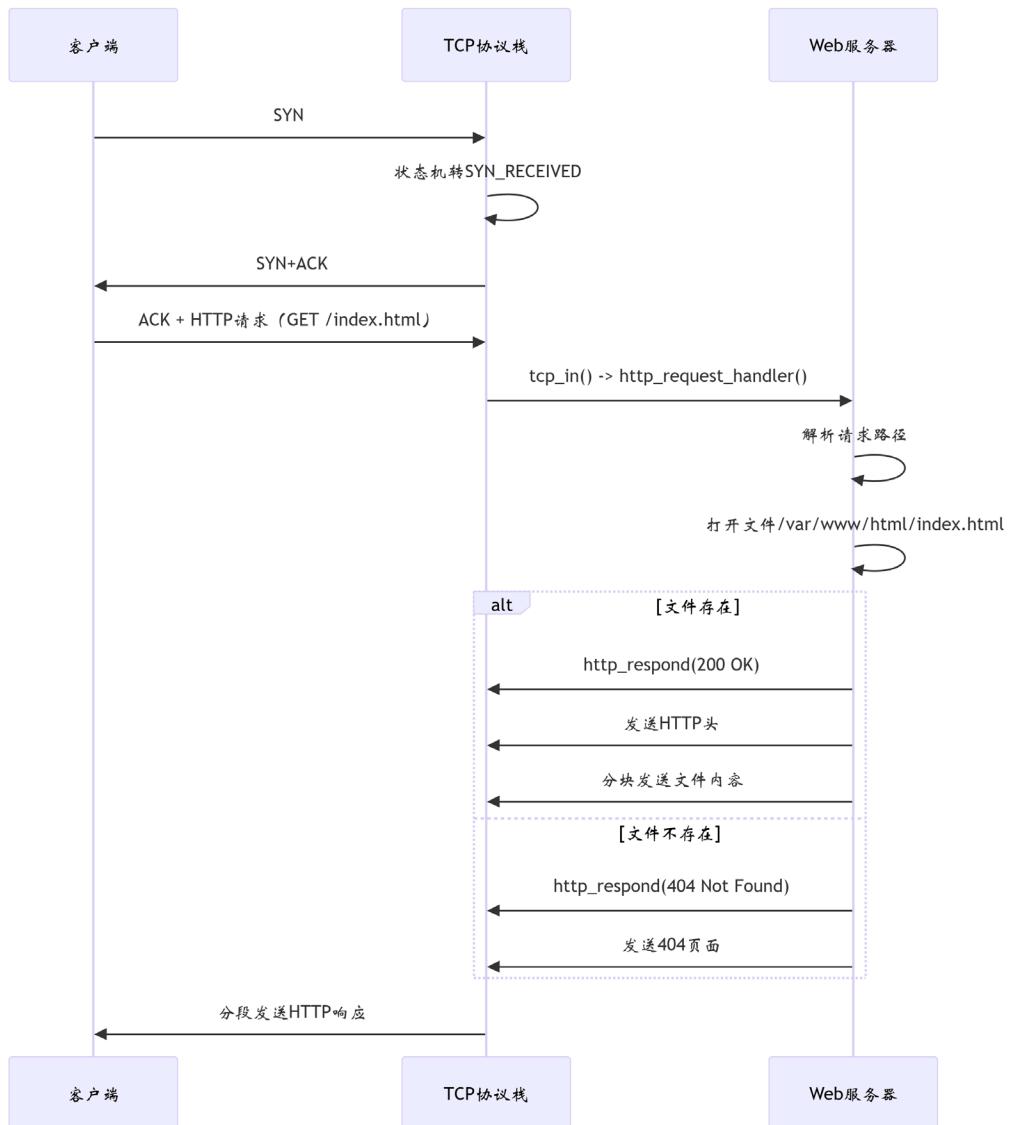
发送方向（从应用到网卡）



Web 服务器 HTTP 响应处理流程



## Web 服务器详细处理流程



## 关键函数调用关系

协议层	接收方向	发送方向	Web 服务器
驱动层	driver_recv()	driver_send()	-
链路层	ethernet_in()	ethernet_out()	-
ARP	arp_in()	arp_req()/arp_resp()	-
IP 层	ip_in()	ip_out()/ip_fragment_out()	-
ICMP	icmp_in()/icmp_resp()	icmp_unreachable()	-
TCP	tcp_in()	tcp_out()/tcp_send()	-
UDP	udp_in()	udp_out()/udp_send()	-
应用层	tcp_handler()	-	http_request_handler()
HTTP 服务	-	-	http_respond()
文件操作	-	-	fopen()/fread()/fclose()

## 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。)

数据封装（发送）

1. 长度检查：若数据长度 < 46 字节，填充 0 至最小长度

```
//Step1:若数据长度不足 46 字节, 需显式填充 0
if (buf->len < ETHERNET_MIN_TRANSPORT_UNIT){
    buf_add_padding(buf, len: ETHERNET_MIN_TRANSPORT_UNIT-buf->len);
}
```

2. 添加帧头：

```
//Step2:添加以太网包头
buf_add_header(buf, len: sizeof(ether_hdr_t));
ether_hdr_t *hdr = (ether_hdr_t *)buf->data;
//Step3:填写目的MAC地址
memcpy(hdr->dst, Src: mac, Size: NET_MAC_LEN);
//Step4:填写源MAC地址, 即本机的MAC地址。
memcpy(Dst: hdr->src, Src: net_if_mac, Size: NET_MAC_LEN);
//Step5:填写协议类型 protocol。
hdr->protocol16 = swap16(protocol);
```

3. 发送数据帧：

```
//Step6:发送数据帧
driver_send(buf);
```

数据解封装（接收）

1. 长度检查：丢弃长度 < 以太网头部的包

```
//Step1:数据长度检查
if (buf->len < sizeof(ether_hdr_t)){
    return;
}
```

2. 解析协议，移除帧头：

```
//Step2:移除以太网包头
ether_hdr_t *eth = (ether_hdr_t *)buf->data;
uint16_t protocol = swap16(eth->protocol16);
uint8_t* mac = eth->src;

buf_remove_header(buf, len: sizeof(ether_hdr_t));
```

3. 协议分发：

```
//Step3:向上层传递数据包
net_in(buf, protocol, src: mac);
```

### 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。)

ARP 请求处理

```
//Step1. 初始化缓冲区:  
buf_init(buf: &txbuf, len: sizeof(arp_pkt_t));  
//Step2. 填写ARP报头:  
arp_pkt_t* hdr = (arp_pkt_t*)txbuf.data;  
hdr->hw_type16 = swap16(ARP_HW_ETHER);  
hdr->proto_type16 = swap16(NET_PROTOCOL_IP);  
hdr->hw_len = NET_MAC_LEN;  
hdr->proto_len = NET_IP_LEN;  
memcpy(Dst: hdr->sender_mac, Src: net_if_mac, Size: NET_MAC_LEN);  
memcpy(Dst: hdr->sender_ip, Src: net_if_ip, Size: NET_IP_LEN);  
memcpy(Dst: hdr->target_ip, Src: target_ip, Size: NET_IP_LEN);  
//Step3. 设置操作类型(同时要注意进行大小端转换):  
hdr->opcode16 = swap16(ARP_REQUEST);  
//Step4. 发送 ARP 报文:  
ethernet_out(buf: &txbuf, mac: ether_broadcast_mac, protocol: NET_PROTOCOL_ARP);
```

关键点：设置操作码为 ARP\_REQUEST，广播发送

ARP 响应处理

```
//Step1. 初始化缓冲区:  
buf_init(buf: &txbuf, len: sizeof(arp_pkt_t));  
//Step2. 填写 ARP 报头首部:  
arp_pkt_t* hdr = (arp_pkt_t*)txbuf.data;  
hdr->hw_type16 = swap16(ARP_HW_ETHER);  
hdr->proto_type16 = swap16(NET_PROTOCOL_IP);  
hdr->hw_len = NET_MAC_LEN;  
hdr->proto_len = NET_IP_LEN;  
memcpy(Dst: hdr->sender_mac, Src: net_if_mac, Size: NET_MAC_LEN);  
memcpy(Dst: hdr->sender_ip, Src: net_if_ip, Size: NET_IP_LEN);  
memcpy(Dst: hdr->target_mac, Src: target_mac, Size: NET_MAC_LEN);  
memcpy(Dst: hdr->target_ip, Src: target_ip, Size: NET_IP_LEN);  
//Step3. 设置操作类型(同时要注意进行大小端转换):  
hdr->opcode16 = swap16(ARP_REPLY);  
//Step4. 发送 ARP 报文:  
ethernet_out(buf: &txbuf, mac: target_mac, protocol: NET_PROTOCOL_ARP);
```

关键点：设置操作码为 ARP\_REPLY，单点发送

ARP 表项更新机制

1. 动态更新：收到 ARP 响应时更新表项

```
110 void arp_in(buf_t *buf, uint8_t *src_mac) {  
126     //Step3. 更新 ARP 表项:  
127     map_set(map: &arp_table, key: hdr->sender_ip, value: hdr->sender_mac);
```

## 2. 超时机制：表项默认超时 5 分钟 (=60\*5=300 秒)

```
#define ARP_TIMEOUT_SEC (60 * 5) // arp表过期时间
#define ARP_MIN_INTERVAL (1 * 5) // arp请求的最小间隔
#define IP_DEFAULT_TTL 64
```

The screenshot shows a code editor with a search bar at the top containing the text "宏 ARP\_TIMEOUT\_SEC config.h". Below the search bar, there are two entries in the results list:

- arp.c 177 map\_init(&arp\_table, NET\_IP\_LEN, NET\_MAC\_LEN, 0, ARP\_TIMEOUT\_SEC, NULL, NULL);
- arp.c 39 map\_init(&arp\_table, NET\_IP\_LEN, NET\_MAC\_LEN, 0, ARP\_TIMEOUT\_SEC, NULL, NULL);

## 3. 缓存队列：未解析的包暂存 arp\_buf

```
153 void arp_out(buf_t *buf, uint8_t *ip) {
163     //Step3. 未找到对应 MAC 地址:
164     if (map_get(map: &arp_buf, key: ip)==NULL){//如果arp_buf里没有包
165         //将这个包缓存到 arp_buf 中
166         map_set(map: &arp_buf, key: ip, value: buf);
167         //发送一个请求目标IP地址对应的MAC地址的ARP请求报文
168         arp_req(target_ip: ip);
```

## 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。)

### 数据封装（发送）

#### 1. 分片决策：超过 MTU(1500B) 时自动分片

```
115 void ip_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol) {
133     //分片发送数据报
134     uint16_t current_offset = 0;
135     while (buf->len > data_max_len){
136         //首先调用 buf_init() 初始化一个 ip_buf。
137         buf_init(buf: ip_buf, len: data_max_len);
138         //将数据报截断，每个截断后的包长等于 IP 协议最大负载包长 (1500 字节 - IP 首部长度)
139         memcpy(Dst: ip_buf->data, Src: buf->data, Size: data_max_len);
140         buf_remove_header(buf, len: data_max_len);
141         //调用 ip_fragment_out() 函数发送出去
142         ip_fragment_out(buf: ip_buf, ip, protocol, id: ip_id, offset: current_offset/IP_
143         current_offset += data_max_len;
144     }
```

#### 2. 填充 IP 头：

考虑到 IP 报头字段有点多，这里只放了部分，作为示意即可。

```
77 void ip_fragment_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol, int id,
78     //Step1 增加头部缓存空间:
79     buf_add_header(buf, len: sizeof(ip_hdr_t));
80
81     //Step2 填写头部字段:
82     ip_hdr_t* hdr = (ip_hdr_t*)buf->data;
83
84     hdr->version      = IP_VERSION_4;
85     hdr->hdr_len      = sizeof(ip_hdr_t)/IP_HDR_LEN_PER_BYTE;
86     hdr->tos          = 0;
87     hdr->total_len16 = swap16(buf->len);
88     hdr->id16         = swap16(id);
```

### 3. 计算校验和并发送:

```
//Step3 计算并填写校验和:  
hdr->hdr_checksum16 = checksum16(data: (uint16_t*)hdr, len: sizeof(ip_hdr_t));  
  
//Step4 发送数据:  
//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);
```

数据解封装（接收）

#### 1. 头校验：检查版本/长度/校验和 长度与报头检测：

```
//Step1 检查数据包长度:  
if (buf->len < sizeof(ip_hdr_t)){//数据包长度小于最小首部长度，丢弃  
    return;  
}  
ip_hdr_t* hdr = (ip_hdr_t*)buf->data;  
  
//Step2 进行报头检测:  
if (hdr->version != IP_VERSION_4 || //版本号不为IPv4  
    swap16(hdr->total_len16) > buf->len || //总长度字段大于收到的数据包长度  
    hdr->hdr_len < sizeof(ip_hdr_t) / IP_HDR_LEN_PER_BYTE){//IP报头长度小于最小首部长度  
    return;  
}
```

校验和：

```
//Step3 校验头部校验和:  
//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为 0  
//是否需要大小端转换？——不需要，计算与验算均按网络字节顺序（见ip_fragment_out）  
uint16_t hdr_checksum = hdr->hdr_checksum16;  
hdr->hdr_checksum16 = 0;  
  
//调用 checksum16 函数来计算头部校验和与 IP 头部原本的头部校验和字段进行对比  
//若不一致，丢弃  
if (hdr_checksum != checksum16(data: (uint16_t *)hdr, len: hdr->hdr_len * IP_HDR_LEN_16)){  
    return;  
}  
//若一致，则再将该头部校验和字段恢复成原来的值  
hdr->hdr_checksum16 = hdr_checksum;
```

#### 2. 目标 IP 匹配

```
//Step4 对比目的 IP 地址:  
//若不是发送给本机的，将其丢弃  
if (memcmp(Buf1: hdr->dst_ip, Buf2: net_if_ip, Size: NET_IP_LEN) != 0){  
    return;  
}
```

#### 3. 协议分发

```

//Step6 去掉 IP 报头:
buf_remove_header(buf, len: hdr->hdr_len * IP_HDR_LEN_PER_BYTE);

//Step7 向上层传递数据包:
//调用 net_in() 函数向上层传递数据包。
if (net_in(buf, hdr->protocol, src: hdr->src_ip) != 0){
    //传递失败, 返回-1
    //重新加入 IP 报头
    buf_add_header(buf, len: hdr->hdr_len * IP_HDR_LEN_PER_BYTE);
    //调用 icmp_unreachable() 函数返回 ICMP 协议不可达信息
    icmp_unreachable(recv_buf: buf, hdr->src_ip, code: ICMP_CODE_PROTOCOL_UNREACH);
}

```

## 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。)

请求处理 (Ping)

```

void icmp_in(buf_t *buf, uint8_t *src_ip) {
    //Step3 回送回显应答: 调用 icmp_resp() 函数回送一个回显应答 (ping 应答)。
    if (hdr->type == ICMP_TYPE_ECHO_REQUEST &&
        hdr->code == 0){//本实验不支持0对应的代码
        icmp_resp(req_buf: buf, src_ip);
    }
}

```

响应生成

```

icmp_hdr_t *req_hdr = (icmp_hdr_t *)req_buf->data;
icmp_hdr_t *hdr = (icmp_hdr_t*)txbuf.data;

//封装 ICMP 报头
hdr->type = ICMP_TYPE_ECHO_REPLY;
hdr->code = 0;//本实验不支持0对应的代码
hdr->checksum16 = 0;
hdr->id16 = req_hdr->id16;
hdr->seq16 = req_hdr->seq16;

//Step2 填写校验和:
//ICMP报头的校验和涵盖了整个报文, 因此直接使用txbuf的参数而不是hdr
hdr->checksum16 = checksum16(data: (uint16_t*)txbuf.data, txbuf.len);

//Step3 发送数据报:
//调用 ip_out() 函数将封装好且填写了校验和的 ICMP 数据报发送出去。
ip_out(buf: &txbuf, ip: src_ip, protocol: NET_PROTOCOL_ICMP);

```

故障诊断 (ICMP 不可达)

```

icmp_hdr_t *hdr = (icmp_hdr_t*)txbuf.data;
hdr->type = ICMP_TYPE_UNREACH;
hdr->code = code;
hdr->checksum16 = 0;
hdr->id16 = 0;
hdr->seq16 = 0;

//Step3 计算校验和
//同icmp_resp()
hdr->checksum16 = checksum16( data: (uint16_t *)txbuf.data, txbuf.len);

//Step4 发送数据报
ip_out( buf: &txbuf, ip: src_ip, protocol: NET_PROTOCOL_ICMP);

```

不同 ICMP 代码对应场景:

```

typedef enum icmp_code {
    ICMP_CODE_PROTOCOL_UNREACH = 2, // 协议不可达
    ICMP_CODE_PORT_UNREACH = 3       // 端口不可达
} icmp_code_t;

```

## 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。)

数据封装 (发送)

1. 添加 UDP 头:

```

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

//Step2 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);

```

2. 计算校验和: 包含伪头部

```

//Step3 计算并填充校验和:
//先将校验和字段填充为 0
hdr->checksum16 = 0;
//然后调用 transport_checksum() 函数计算 UDP 数据报的校验和
hdr->checksum16 = transport_checksum(protocol: NET_PROTOCOL_UDP, buf, src_ip: net_if_ip,

```

伪首部结构如下所示:

定义在 udp.h 中。

```

typedef struct udp_peso_hdr
{
    uint8_t src_ip[4];      // 源IP地址
    uint8_t dst_ip[4];      // 目的IP地址
    uint8_t placeholder;   // 必须置0,用于填充对齐
    uint8_t protocol;      // 协议号
    uint16_t total_len16;   // 整个数据包的长度
} udp_peso_hdr_t;

```

增加伪首部:

```

//Step1 增加 UDP 伪头部:
size_t udp_peso_hdr_len = sizeof(udp_peso_hdr_t);
buf_add_header(buf, len: udp_peso_hdr_len);

//Step2 暂存 IP 头部:
uint8_t tmp[udp_peso_hdr_len];
//按字节拷贝后再按字节还原
memcpy(Dst: tmp, Src: buf->data, Size: udp_peso_hdr_len);

//Step3 填写 UDP 伪头部字段:
udp_peso_hdr_t* udp_peso_hdr = (udp_peso_hdr_t*)buf->data;
memcpy(Dst: udp_peso_hdr->src_ip, Src: src_ip, Size: NET_IP_LEN);
memcpy(Dst: udp_peso_hdr->dst_ip, Src: dst_ip, Size: NET_IP_LEN);
udp_peso_hdr->placeholder = 0;
udp_peso_hdr->protocol = protocol;
udp_peso_hdr->total_len16 = swap16(buf->len - udp_peso_hdr_len);

```

数据解封装（接收）

1. 校验检查:

```

//Step2 重新计算校验和:
//先把首部的校验和字段保存起来，然后将该字段填充为 0。
uint16_t checksum16 = hdr->checksum16;
hdr->checksum16 = 0;
//接着调用 transport_checksum() 函数重新计算校验和
//将计算得到的校验和值与接收到的 UDP 数据报的校验和进行比较
if (checksum16 != transport_checksum(protocol: NET_PROTOCOL_UDP, buf, src_ip, dst_ip: net_ip))
    //如果两者不一致，则说明数据报在传输过程中可能发生了错误，将其丢弃
    return;
}

```

2. 端口查找:

```

//Step3 查询处理函数:
//调用 map_get() 函数，在 udp_table 中查询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(hdr->dst_port16);
udp_handler_t *handler = map_get(map: &udp_table, key: &dst_port16);

```

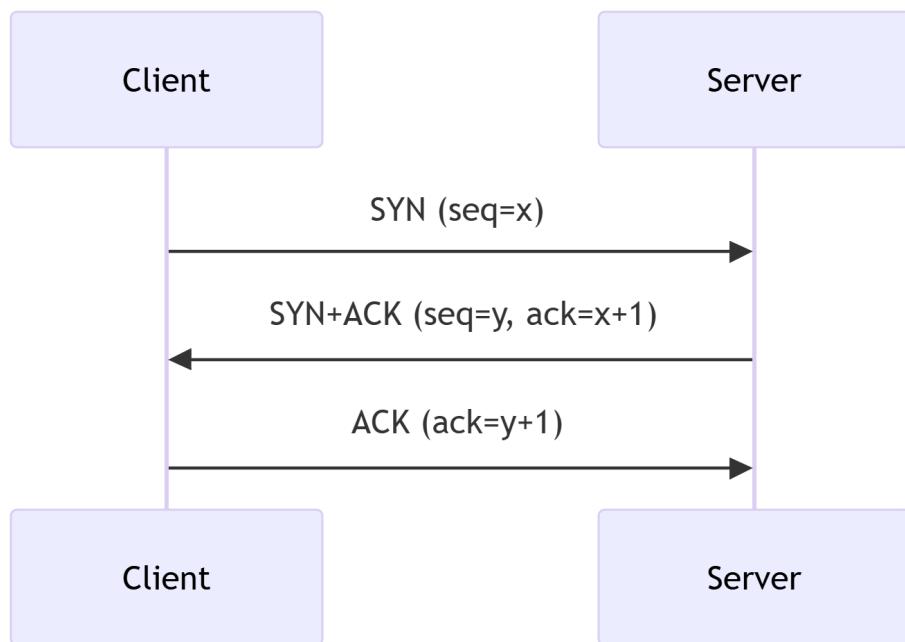
### 3. 无处理器时：回复 ICMP 端口不可达

```
//Step4 处理未找到处理函数的情况:
if (handler == NULL){
    //调用 buf_add_header() 函数增加 IPv4 数据报头部
    buf_add_header(buf, len: sizeof(ip_hdr_t));
    //调用 icmp_unreachable() 函数发送一个端口不可达的 ICMP 差错报文
    icmp_unreachable(recv_buf: buf, src_ip, code: ICMP_CODE_PORT_UNREACH);
    return;
}
```

## 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。)

连接建立（三次握手）过程如图所示：



代码实现：

Server 收到第一次握手时：

```
case TCP_STATE_LISTEN:
    // TODO: 仅在收到连接报文时 (SYN报文) 才做出处理, 否则直接返回
    if (!TCP_FLG_ISSET(recv_flags, TCP_FLG_SYN)) {
        return;
    }
    // TODO: 初始化 TCP 连接上下文 (tcp_conn结构体) 的seq字段
    tcp_conn->seq = tcp_generate_initial_seq();
    // TODO: 填写 TCP 连接上下文 (tcp_conn结构体) 的ack字段
    tcp_conn->ack = remote_seq + 1;
    // TODO: 填写回复标志 send_flags
    send_flags = TCP_FLG_SYN | TCP_FLG_ACK;
    // TODO: 进行状态转移
    tcp_conn->state = TCP_STATE_SYN_RECEIVED;
    break;
```

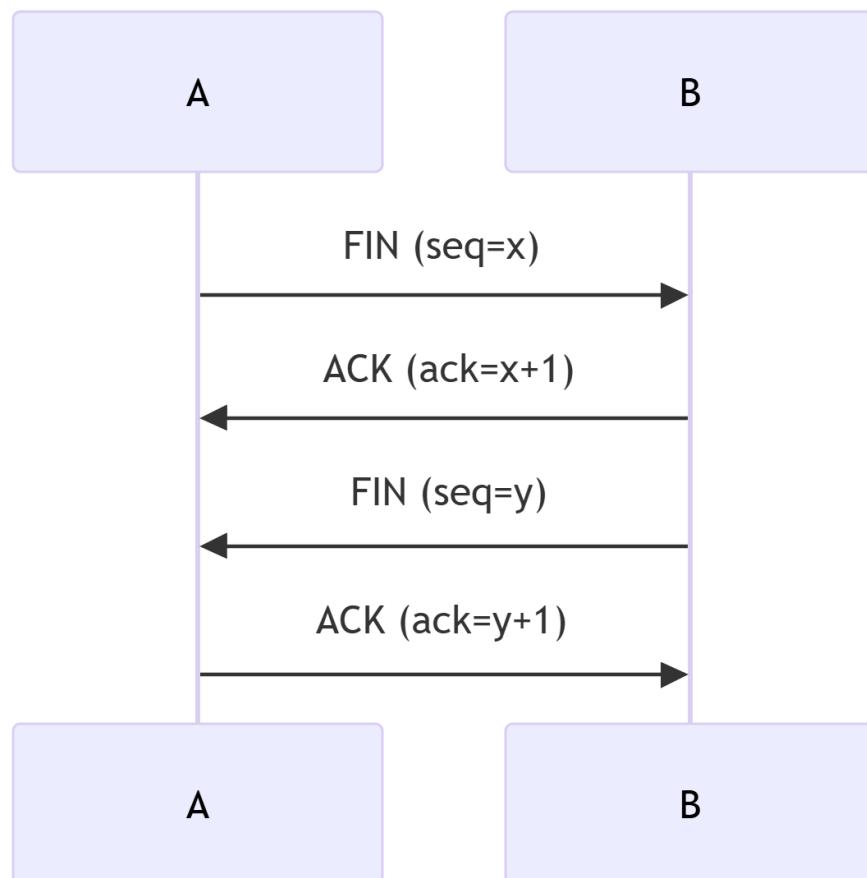
Server 收到第三次握手时建立连接:

```
case TCP_STATE_SYN_RECEIVED:
    // TODO: 仅在收到确认报文时 (ACK报文) 才做出处理, 否则直接返回
    if (!TCP_FLG_ISSET(recv_flags, TCP_FLG_ACK)) {
        return;
    }
    // TODO: 进行状态转移
    tcp_conn->state = TCP_STATE_ESTABLISHED;
    break;
```

Server 在连接建立后持续接收包:

```
// TODO: 计算接收到的数据长度, 更新 ACK
uint32_t data_size = buf->len - tcp_hdr_sz;
tcp_conn->ack += data_size;
// TODO: 如果接收报文携带数据, 则填写回复标志 send_flags 发送ACK
if (data_size){
    send_flags |= TCP_FLG_ACK;
}
```

连接关闭（四次挥手）过程如图所示:



代码实现：

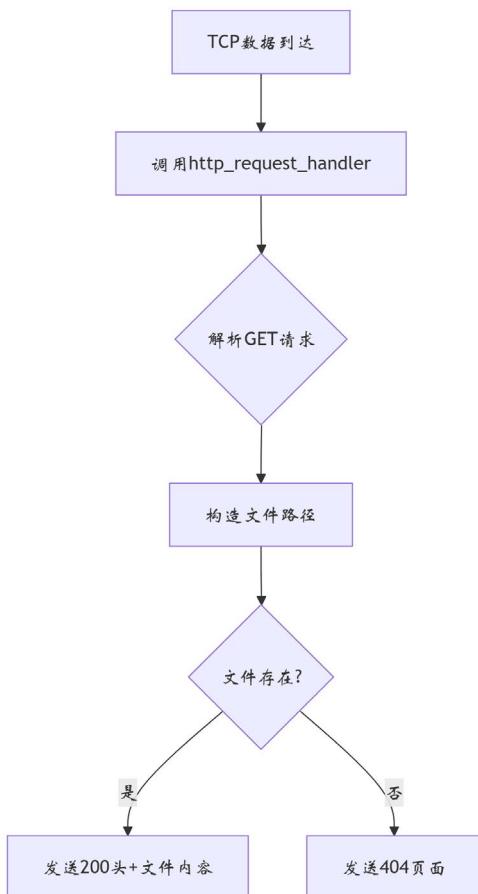
```
// TODO: 如果收到 FIN 报文, 则增加 send_flags 相应标志位, 并且进行状态转移
if (TCP_FLG_ISSET(recv_flags, TCP_FLG_FIN)){
    tcp_conn->ack++;
    send_flags |= TCP_FLG_ACK;
    tcp_conn->state = TCP_STATE_LAST_ACK;
}
break;

case TCP_STATE_LAST_ACK:
    // TODO: 仅在收到确认报文时 (ACK报文) 才做出处理, 否则直接返回
    if (!TCP_FLG_ISSET(recv_flags, TCP_FLG_ACK)){
        return;
    }
    // TODO: 关闭 TCP 连接
    tcp_close_connection(remote_ip, remote_port, host_port);
    break;
```

## 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。)

请求处理流程如下所示：



关键代码实现：

1. 请求解析：

```
while (data[idx] != ' ') {
    url_path[j++] = data[idx++];
}
url_path[j] = '\0';

// 发送响应
http_respond(tcp_conn, url_path, port: HTTP_LISTEN_PORT, dst_ip: src_ip, dst_port: src_

```

2. 响应生成

404 响应：

```
// 文件不存在时发送 404 响应
if (!file) {
    // HTTP 404 响应请求体
    char *not_found_body = "<HTML><TITLE>Not Found</TITLE>\r\n"
                           "The resource specified\r\n"
                           "is unavailable or nonexistent.\r\n"
                           "</BODY></HTML>\r\n";
    /* Step1：发送 HTTP 404 请求头 */
    // TODO: 发送 HTTP 状态行
    sprintf(stream: resp_buffer, format: "HTTP/1.1 404 Not Found\r\n");
    tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: strlen(Str: resp_buffer), src_

```

```
        // 发送 HTTP 连接信息
        sprintf(stream: resp_buffer, format: "Connection: Keep-Alive\r\n");
        tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: strlen(Str: resp_buffer), src_

```

```
        // TODO: 发送 HTTP 内容类型
        sprintf(stream: resp_buffer, format: "Content-Type: text/html\r\n");
        tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: strlen(Str: resp_buffer), src_

```

200 响应：

```
/* Step2：发送 HTTP 请求头 */
// TODO: 发送 HTTP 状态行
sprintf(stream: resp_buffer, format: "HTTP/1.1 200 OK\r\n");
tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: strlen(Str: resp_buffer), src_

```

```
        // 发送 HTTP 连接信息
        sprintf(stream: resp_buffer, format: "Connection: Keep-Alive\r\n");
        tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: strlen(Str: resp_buffer), src_

```

```
        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
        sprintf(stream: resp_buffer, format: "Content-Type: %s\r\n", content_type);
        tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: strlen(Str: resp_buffer), src_

```

```
        fseek(file, Offset: 0, Origin: SEEK_END);
        size_t content_length = ftell(file);
        fseek(file, Offset: 0, Origin: SEEK_SET);

```

## 特性支持

MIME 类型识别：根据文件扩展名返回 Content-Type

```
static inline const char *http_get_mime_type(const char *file_path) {
    if (strstr(Str::file_path, SubStr: ".html") || strstr(Str::file_path, SubStr: ".htm")) {
        return "text/html; charset=utf-8";
    } else if (strstr(Str::file_path, SubStr: ".css")) {
        return "text/css";
    } else if (strstr(Str::file_path, SubStr: ".jpg") || strstr(Str::file_path, SubStr: ".jpeg")) {
        return "image/jpeg";
    }
    return "application/octet-stream"; // 默认类型
}
```

Keep-Alive 连接：响应头包含 Connection: Keep-Alive（见上响应生成部分）

大文件分块传输：每次读取 1KB（char resp\_buffer[1024]）并发送

```
/* Step3 : 发送 HTTP 响应体 */
size_t bytes_read;
while ((bytes_read = fread(DstBuf: resp_buffer, ElementSize: 1, Count: sizeof(resp_buffer), file))
    // TODO: 每次发送读取的文件内容块
    tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: bytes_read, src_port: port, dst_ip, dst_
}
```

## 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。)*

注：为方便，以下实验的 log↔demo\_log 的对比不再叙述了，直接用实验自测截图代替。

### 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)*

实验自测：

```
Test project E:/net-lab/build_CLion
    Start 1: eth_in
1/1 Test #1: eth_in ..... Passed      0.26 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.27 sec
(base) PS E:\net-lab\build_CLion> ctest -R eth_out
Test project E:/net-lab/build_CLion
    Start 2: eth_out
1/1 Test #2: eth_out ..... Passed      0.40 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.42 sec
```

抓包分析：

eth\_in:

demo\_out.pcap 与 out.pcap 均为空。

毕竟 eth\_in 只测试了主机收到以太网帧的函数，没有涉及到发出数据帧。

eth\_out:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.163.103	10.0.2.2	TCP	60	22 → 64289 [ACK] Seq=1 Ack=1 Win=65535 Len=0
2	0.000000	192.168.163.103	10.0.2.2	SSH	186	Server: Encrypted packet (len=52)
3	0.000000	192.168.163.103	10.248.98.30	DNS	84	Standard query 0x96da A www.baidu.com OPT
4	0.000000	192.168.163.103	10.248.98.30	DNS	84	Standard query 0x9759 AAAA www.baidu.com OPT
5	0.000000	192.168.163.103	10.248.98.30	DNS	87	Standard query 0x5a54 AAAA www.a.shifen.com OPT
6	0.000000	192.168.163.103	183.232.231.172	ICMP	98	Echo (ping) request id=0x0001, seq=1/256, ttl=64 (no response found!)
7	0.000000	192.168.163.103	10.0.2.2	TCP	60	[TCP Previous_segment_not_captured] 22 → 64289 [ACK] Seq=261 Ack=261 Win=65535 Len=0
8	0.000000	192.168.163.103	10.0.2.2	SSH	186	Server: Encrypted packet (len=52)
9	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 10.0.2.3? Tell 192.168.163.103
10	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 10.0.2.4? Tell 192.168.163.103
11	0.000000	192.168.163.103	10.0.2.4	ICMP	98	Echo (ping) request id=0x0003, seq=1/256, ttl=64 (no response found!)
12	0.000000	192.168.163.103	10.248.98.30	DNS	100	Standard query 0x5d43 A connectivity-check.ubuntu.com OPT
13	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 10.0.2.5? Tell 192.168.163.103
14	0.000000	192.168.163.103	35.224.99.156	HTTP	141	GET / HTTP/1.1
15	0.000000	11:22:33:44:55:66	11:25:45:c2:dc:93		0x65f8	60 Ethernet II
16	0.000000	11:22:33:44:55:66	a5:8e:1e:83:1f:35		0x16c9	60 Ethernet II
17	0.000000	11:22:33:44:55:66	eF:8d:35:50:6c:c2		0x1712	60 Ethernet II

```

> Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: 00:0c:29:00:00:00 (00:0c:29:00:00:00), Dst: 52:54:00:12:35:02 (52:54:00:12:35:02)
> Destination: 52:54:00:12:35:02 (52:54:00:12:35:02)
> Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
Type: IPv4 (0x0800)
[Stream index: 0]
Padding: 0x0000000000000000
Internet Protocol Version 4, Src: 192.168.163.103, Dst: 10.0.2.2
Transmission Control Protocol, Src Port: 22, Dst Port: 64289, Seq: 1, Ack: 1

```

帧的结构和内容符合预期。目的 MAC 地址、源 MAC 地址、协议类型字段以及数据部分都是正确的。

```
#define NET_IF_IP \
{ \
    192, 168, 163, 103 \
} // 测试用网卡ip地址

#define NET_IF_MAC \
{ \
    0x11, 0x22, 0x33, 0x44, 0x55, 0x66 \
} // 测试用网卡mac地址
```

## 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。)

实验自测:

```
(base) PS E:\net-lab\build_CLion> ctest -R arp_test
Test project E:/net-lab/build_CLion
  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) = 0.27 sec
```

抓包分析:

No.	Time	Source	Destination	Protocol	Lengt	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 192.168.163.103? 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=54
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

数据包 2 对应的 info 是 who has 192.168.163.10 ? tell 192.168.163.103

也就是 arp 查询请求，查询 192.168.163.10。

发送方的 mac 地址是 11:22:33:44:55:66 (NET\_IF\_MAC)

No.	Time	Source	Destination	Protocol	Lengt	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 192.168.163.103? 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=54
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

数据包 2 对应的 info 是 192.168.163.103 is at 11:22:33:44:55:66。也就是 arp 响应报文

arp 响应中的目的 mac 地址是 1a:94:f0:3c:49:aa，IP 地址是 192.168.163.2

即这份响应报文回应的对应 arp 查询请求来自 192.168.163.2，其 mac 地址如上

### 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。)

实验自测：

ip\_test:

(base) PS E:\net-lab\build_CLion> ctest -R ip_test
Test project E:/net-lab/build_CLion
Start 4: ip_test
1/1 Test #4: ip_test ..... Passed 0.28 sec
100% tests passed, 0 tests failed out of 1
Total Test time (real) = 0.29 sec

## ip\_frag\_test:

```
(base) PS E:\net-lab\build_CLion> ctest -R ip_frag_test
Test project E:/net-lab/build_CLion
    Start 5: ip_frag_test
1/1 Test #5: ip_frag_test ..... Passed      0.23 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.25 sec
```

## 抓包分析：

No.	Time	Source	Destination	Protocol	Lengt	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		

> Frame 3: 84 bytes on wire (672 bits), 84 bytes captured (672 bits)	0000 21 32 43 54 65 06 11 22 33 44 55 66 08 00 45 00 !2CTe .." 3DUF..E
> Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 21:32:43:54:65 (00:0c:29:a8:3d:c0)	0010 00 46 00 00 00 40 11 b2 e4 c8 a8 a3 67 c0 a8 .F....@. ....g..
> Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10	0020 a3 0a ae 1b 00 35 00 32 79 68 96 da 01 00 00 01 ....5 2 yh..
0101 .... = Version: 4	0030 00 00 00 00 00 01 03 77 77 77 05 62 61 69 64 75 ....w ww baidu..
.... 0101 = Header Length: 20 bytes (5)	0040 03 63 6f 6d 00 00 01 00 01 00 00 29 20 00 00 00 ..com .....
> Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)	0050 00 00 00 00 .....
Total Length: 70	
Identification: 0x0000 (0)	
000. .... = Flags: 0x0	
... 0 0000 0000 0000 = Fragment Offset: 0	
Time to Live: 64	
Protocol: UDP (17)	
Header Checksum: 0xb2e4 [validation disabled]	
[Header checksum status: Unverified]	
Source Address: 192.168.163.103	
Destination Address: 192.168.163.10	
[Stream index: 0]	
> User Datagram Protocol, Src Port: 44571, Dst Port: 53	
> Domain Name System (query)	

IP 头部字段的正确。版本号=4、首部长度=5 (4\*5=20B)、总长度=70、标识=0x0000 (第一个使用 IP 封装的报文)、标志位=0x0 (DF=0, MF=0)、片偏移=0 (无分片)、TTL=64、协议类型=17 (DNS 使用 UDP)

没有差异	
内容仅在行分隔符中有差异	
testing\data\ip_frag_test\log	CRLF
✓ arp_out:	1 1
ip:192.168.163.103	2 2
buf: 45 00 05 dc 00 00 20 00 40 06	3 3
arp_out:	4 4
ip:192.168.163.103	5 5
buf: 45 00 05 dc 00 00 20 b9 40 06	6 6
arp_out:	7 7
ip:192.168.163.103	8 8
buf: 45 00 05 dc 00 00 21 72 40 06	9 9
arp_out:	10 10
ip:192.168.163.103	11 11
buf: 45 00 02 6c 00 00 02 2b 40 06	12 12

在 ip\_frag\_test.c 代码中调用 ip\_out() 函数时，目标 IP 地址设置为 net\_if\_ip (即本机网络接口的 IP): ip\_out(&buf, net\_if\_ip, NET\_PROTOCOL\_TCP);即目标 IP 是本机 IP 这意味着 IP 层会将数据包视为发往本机的数据 (环回流量)。在协议栈实现中，这类数据包通常直接交给上层协议处理，而不会经过物理网络接口。因此：

不会触发 ARP 请求，也不会调用底层网络发送函数

数据包不会被记录到 pcap\_out 文件 (该文件通常只记录物理接口的收发数据)

因此在这里采用 log 与 demo\_log 的内容比较来进行分片正确性验证

## 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。)

实验自测:

```
(base) PS E:\net-lab\build_CLion> ctest -R icmp_test
Test project E:/net-lab/build_CLion
    Start 6: icmp_test
1/1 Test #6: icmp_test ..... Passed 0.29 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.30 sec
```

抓包分析:

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	68 192.168.163.103 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	68 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)

Frame 7: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
Ethernet II, Src: Pixim 34:45:56 (11:22:33:44:55:66), Dst: Pixim 34:45:56 (11:22:33:44:55:66)
Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.110
Internet Control Message Protocol
Type: 0 (Echo (ping) reply)
Code: 0
Checksum: 0x436a [correct]
[Checksum Status: Good]
Identifier (BE): 1 (0x0001)
Identifier (LE): 256 (0x0100)
Sequence Number (BE): 1 (0x0001)
Sequence Number (LE): 256 (0x0100)
Data (56 bytes)

Frame 13: 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 21:32:43:54:65 (11:22:33:44:55:66)
Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10
Internet Control Message Protocol
Type: 3 (Destination unreachable)
Code: 2 (Protocol unreachable)
Checksum: 0x1846 [correct]
[Checksum Status: Good]
Unused: 00000000
Internet Protocol Version 4, Src: 192.168.163.10, Dst: 192.168.163.103
Transmission Control Protocol, Src Port: 80, Dst Port: 55428

Type=0, 代表 ICMP REPLY 报文, code=0, checksum 无误, 携带一段 ascii 从 0x10 到 0x37 的信息。均符合预期

Type 段与 Code 段参考如下:

```
typedef enum icmp_type {
    ICMP_TYPE_ECHO_REQUEST = 8, // 回显请求
    ICMP_TYPE_ECHO_REPLY = 0, // 回显响应
    ICMP_TYPE_UNREACH = 3, // 目的不可达
} icmp_type_t;

typedef enum icmp_code {
    ICMP_CODE_PROTOCOL_UNREACH = 2, // 协议不可达
    ICMP_CODE_PORT_UNREACH = 3 // 端口不可达
} icmp_code_t;
```

13 0.000000	192.168.163.103	192.168.163.10	ICMP	70 Destination unreachable (Protocol unreachable)
-------------	-----------------	----------------	------	---

Frame 13: 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 21:32:43:54:65 (11:22:33:44:55:66)
Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10
Internet Control Message Protocol
Type: 3 (Destination unreachable)
Code: 2 (Protocol unreachable)
Checksum: 0x1846 [correct]
[Checksum Status: Good]
Unused: 00000000
Internet Protocol Version 4, Src: 192.168.163.10, Dst: 192.168.163.103
Transmission Control Protocol, Src Port: 80, Dst Port: 55428

Type=3, 代表 ICMP UNREACH 报文, code=2, 说明端口不可达 (TCP 的 80 号端口对应 HTTP)。checksum 无误, 携带一段全 0 信息。均符合预期

## 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。)

首先是 udp\_test (单元测试):

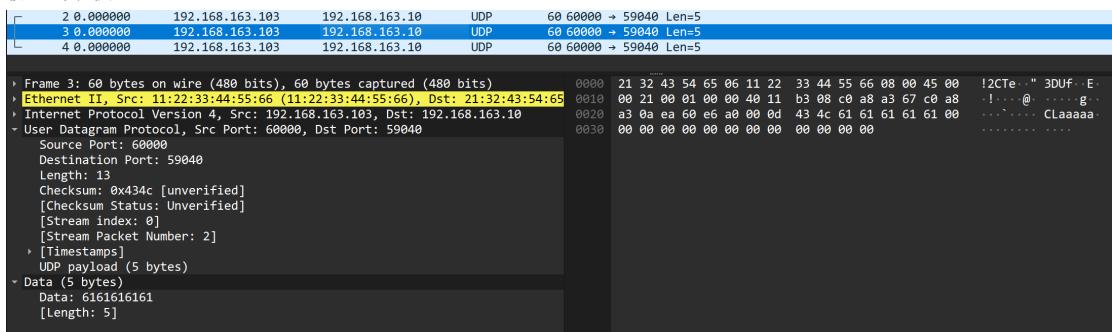
实验自测:

```
(base) PS E:\net-lab\build_CLion> ctest -R udp_test
Test project E:/net-lab/build_CLion
    Start 7: udp_test
1/1 Test #7: udp_test ..... Passed      0.26 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.28 sec
```

抓包分析:



源端口号是 60000, 目的端口号是 59040, 长度为 13, checksum 忽略, 符合预期。

然后是 udp\_server (集成测试):

物理机执行命令 ipconfig, 以太网适配器情况如下:

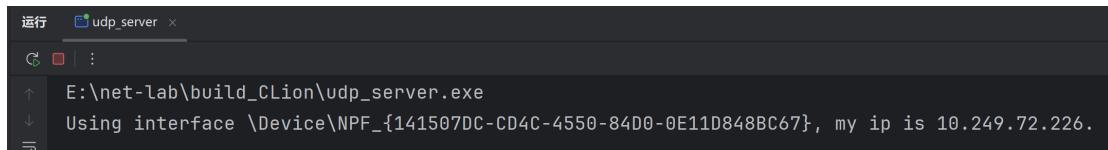
```
以太网适配器 以太网:
连接特定的 DNS 后缀 . . . . . :
IPv6 地址 . . . . . : 2001:250:3c0f:1010::c0a8
本地链接 IPv6 地址 . . . . . : fe80::6176:7645:33ec:3985%4
IPv4 地址 . . . . . : 10.249.72.228
子网掩码 . . . . . : 255.255.248.0
默认网关 . . . . . : fe80::200:5eff:fe00:103%4
                                         10.249.72.1
```

测试网卡设置:

```
#define NET_IF_IP          \
{                           \
|   10, 249, 72, 226 \     \
} // 自定义网卡ip地址

#define NET_IF_MAC          \
{                           \
|   0x00, 0x11, 0x22, 0x33, 0x44, 0x55 \     \
} // 自定义网卡mac地址
```

### 终端中启动协议栈

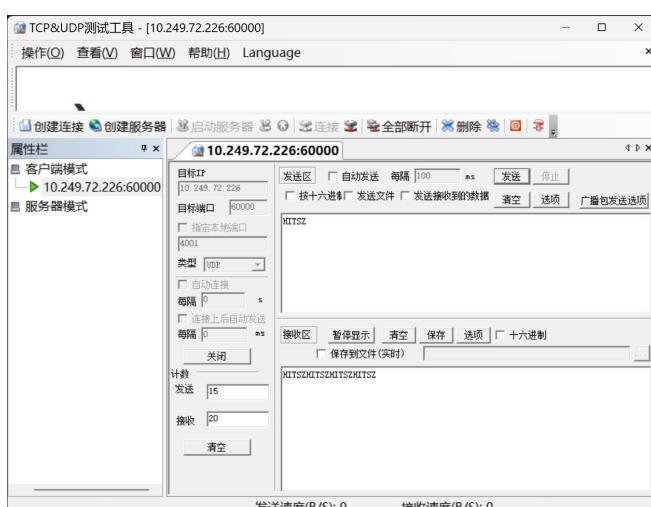


```
E:\net-lab\build_CLion\udp_server.exe
Using interface \Device\NPF_{141507DC-CD4C-4550-84D0-0E11D848BC67}, my ip is 10.249.72.226.
```

### 建立 UDP 连接



### 发送测试数据



```
E:\net-lab\build_CLion\udp_server.exe
Using interface \Device\NPF_{141507DC-CD4C-4550-84D0-0E11D848BC67}, my ip is 10.249.72.226.
recv udp packet from 10.249.72.228:49923 len=5
HITSZ
```

## Wireshark 抓包分析：

No.	Time	Source	Destination	Protocol	Len/Info
1466	132.990578	CIMSYS_33:44:55	Broadcast	ARP	60 ARP Announcement for 10.249.72.226
3029	241.445953	CIMSYS_33:44:55	LCFCElectron_e2:4f:...	ARP	60 10.249.72.226 is at 00:11:22:33:44:55
3184	271.287754	10.249.72.226	10.249.72.226	UDP	47 49923 → 60000 Len=5

Frame 1466: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 00:00 ff:ff:ff:ff:ff at 00:11:22:33:44:55 (Ethernet II, Src: CIMSYS\_33:44:55 (00:11:22:33:44:55), Dst: Broadcast (ff:ff:ff:ff:ff:ff))  
 Destination: Broadcast (ff:ff:ff:ff:ff:ff)  
 Source: CIMSYS\_33:44:55 (00:11:22:33:44:55)  
 Type: ARP (0x0806)  
 [Stream index: 6]  
 Padding: 00000000000000000000000000000000  
 Address Resolution Protocol (ARP Announcement)  
 Hardware type: Ethernet (1)  
 Protocol type: IPv4 (0x0800)  
 Hardware size: 6  
 Protocol size: 4  
 Opcode: request (1)  
 [Is gratuitous: True]  
 [Is announcement: True]  
 Sender MAC address: CIMSYS\_33:44:55 (00:11:22:33:44:55)  
 Sender IP address: 10.249.72.226  
 Target MAC address: 00:00:00:00:00:00 (00:00:00:00:00:00)  
 Target IP address: 10.249.72.226

ARP announcement（地址解析协议通告）是一种主动网络通告机制，主要用于更新局域网内其他设备的 ARP 缓存或防止 IP 地址冲突。当设备的 IP 地址或 MAC 地址发生变化时，通过发送特殊的 ARP 请求包，强制其他主机更新地址映射表，从而确保网络通信的正确性。这里对应我们启动协议栈。

这里是物理机试图向协议栈发送数据，因此先发送 ARP 请求报文查询 10.249.72.226 的 MAC 地址。

(这里有一条 10.249.72.226 is at 00:00:5e:00:01:03 的 ARP 响应报文，似乎与测试网卡不一致。这点在后文：三、实验中遇到的问题及解决方法中有提及。在这里先略过)

这是协议栈发送的 ARP 响应报文，告知物理机 MAC 地址 (00:11:22:33:44:55)。

```
Wireshark - Network traffic analysis tool

eth.addr == 00:11:22:33:44:55
No. Time Source Destination Protocol Len Info
1 1466 132.990570 CIMSYS_33:44:55 Broadcast ARP 60 ARP Announcement for 10.249.72.226
| 3029 241.445953 CIMSYS_33:44:55 LCFCElectron_e2:4f:3e ARP 60 10.249.72.226 is at 00:11:22:33:44:55
| 3184 271.207754 10.249.72.228 10.249.72.226 UDP 47 49923 > 60000 Len=5
|
Frame 3184: wire (376 bytes), 47 bytes captured (376 bits) on interface eth0, duration 0.000000 seconds (0.000000 B/sec), transmit rate 0.000 bits/sec
    Ethernet II, Src: LCFCElectron_e2:4f:3e (84:a9:38:e2:4f:3e), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55)
        Destination: CIMSYS_33:44:55 (00:11:22:33:44:55)
        Source: LCFCElectron_e2:4f:3e (84:a9:38:e2:4f:3e)
        Type: IPv4 (0x0800)
        [Stream index: 7]
    Internet Protocol Version 4, Src: 10.249.72.228, Dst: 10.249.72.226
    User Datagram Protocol, Src Port: 49923, Dst Port: 60000
        Source Port: 49923
        Destination Port: 60000
        Length: 13
        Checksum: 0xb41a [unverified]
        [Checksum Status: Unverified]
        [Stream index: 60]
        [Stream Packet Number: 2]
        [Timestamps]
        UDP payload (5 bytes)
    Data (5 bytes)
        Data: 484954535a
        [Length: 5]

```

物理机通过 udp 发送消息: HITSZ

## 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。)

首先是 `tcp_test` (单元测试):

### 实验自测：

```
(base) PS E:\net-lab\build_CLion> ctest -R tcp_test
Test project E:/net-lab/build_CLion
  Start 8: tcp_test
1/1 Test #8: tcp_test ..... Passed      0.26 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) = 0.27 sec
```

抓包分析：

源端口号、目的端口号在各个数据段的截图上有，就不再赘述了。

接下来主要分析一下三次握手四次挥手及我们实现的服务器连接状态转换。

### 三次握手：

## 第一次: Client -> Server (in.pcap)

```
Frame 2: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) 0000 11 22 33 44 55 66 21 32 43 54 65 06 08 09 04 05 00 "3DUF!2 CTe - E
Ethernet II, Src: 21:32:43:54:65:06 (21:32:43:54:65:06), Dst: 11:22:33:44:55 0010 00 34 bc fd 40 00 80 06 76 03 c0 a8 a3 0a c0 a8 4 @ v
Internet Protocol Version 4, Src: 192.168.163.10, Dst: 192.168.163.100 0020 a3 67 dc 7e ea 60 0e 07 3c cd 00 00 00 00 00 02 g ~ < . .
Transmission Control Protocol, Src Port: 56446, Dst Port: 60000, Seq: 0, Len 0030 fa f0 9a a8 00 00 02 04 05 b4 01 03 03 08 01 01 . .
Source Port: 56446 0040 04 02 . .
Destination Port: 60000 . .
[Stream index: 0] . .
[Stream Packet Number: 1] . .
[Conversation completeness: Incomplete (29)] . .
[TCP Segment Len: 0] . .
Sequence Number: 0 (relative sequence number) . .
Sequence Number (raw): 235355341 . .
[Next Sequence Number: 1 (relative sequence number)] . .
Acknowledgment Number: 0 . .
Acknowledgment number (raw): 0 . .
Acknowledgment number (raw): 0 . .
1000 .... = Header Length: 32 bytes (8) . .
Flags: 0x002 (SYN) . .
Window: 64240 . .
[Calculated window size: 64240] . .
Checksum: 0x9a8 [unverified] . .
[Checksum Status: Unverified] . .
Urgent Pointer: 0 . .
Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scal . .
[Timestamp] . .
```

SYN=1, ACK=0, seq=0。这是 Client 向 Server 的第一次握手。在收到这个报文段前, Server 一直处于 LISTEN 状态。

## 第二次： Server -> Client (out.pcap)

```

> Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) 0000 21 32 43 54 65 06 11 22 33 44 55 66 08 00 45 00 !2CTe .." 3DUF..E
> Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 21:32:43:54:65 0010 00 28 00 00 00 40 06 b3 0d c0 a8 a3 67 c0 a8 (@...@...g...
> Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10 0020 a3 0a ea 60 dc 7e 00 00 14 d4 0e 07 3c ce 50 12 ...P...
> Transmission Control Protocol, Src Port: 60000, Dst Port: 56446, Seq: 0, Ack 0030 ff ff c1 86 00 00 00 00 00 00 00 00 00 00 00 00
  Source Port: 60000
  Destination Port: 56446
  [Stream index: 0]
  [Stream Packet Number: 1]
  > [Conversation completeness: Incomplete (14)]
    [TCP Segment Len: 0]
    Sequence Number: 0 (relative sequence number)
    Sequence Number (raw): 5332
    [Next Sequence Number: 1 (relative sequence number)]
    Acknowledgment Number: 1 (relative ack number)
    Acknowledgment number (raw): 235355342
    0101 .... = Header Length: 20 bytes (5)
  > [Flags: 0x012 (SYN, ACK)]
  Window: 65535
  [Calculated window size: 65535]
  Checksum: 0xc186 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]

```

在收到第一次握手这条报文段后，Server 会发出第二次握手，其 SYN=1，ACK=1，ack\_seq=0+1=1，自身 seq 为随机起始值（如图所示。在此处由于测试程序的全局 seq 起始值为 0）。同时 Server 的状态转变为 SYN\_RCVD

## 第三次： Client -> Server (in.pcap)

```

> Frame 3: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) 0000 11 22 33 44 55 66 21 32 43 54 65 06 08 00 45 00 "3DUF!2 CTe ..E
> Ethernet II, Src: 21:32:43:54:65:06 (21:32:43:54:65:06), Dst: 11:22:33:44:55 0010 00 28 bc fe 40 00 80 06 76 0e c0 a8 a3 0a c0 a8 (@...v...
> Internet Protocol Version 4, Src: 192.168.163.10, Dst: 192.168.163.103 0020 a3 67 dc 7e ea 60 0e 07 3c ce 42 11 45 52 50 10 g ~...< B ERP...
> Transmission Control Protocol, Src Port: 56446, Dst Port: 60000, Seq: 1, Ack 0030 ff 70 4f 88 00 00 p0 ...
  Source Port: 56446
  Destination Port: 60000
  [Stream index: 0]
  [Stream Packet Number: 2]
  > [Conversation completeness: Incomplete (29)]
    [TCP Segment Len: 0]
    Sequence Number: 1 (relative sequence number)
    Sequence Number (raw): 235355342
    [Next Sequence Number: 1 (relative sequence number)]
    Acknowledgment Number: 1 (relative ack number)
    Acknowledgment number (raw): 1108428114
    0101 .... = Header Length: 20 bytes (5)
  > [Flags: 0x010 (ACK)]
  Window: 65392
  [Calculated window size: 16740352]
  [Window size scaling factor: 256]
  Checksum: 0x4f88 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]

```

SYN=0，ACK=1，seq=server.ack\_seq=1,ack\_seq = server.seq+1=1。这是 Client 向 Server 的第三次握手。在收到这个报文段前，Server 一直处于 SYN\_RCVD 状态。在收到这个报文段后，TCP 连接就被正式建立了。Server 转变为 ESTABLISHED 状态。

## 四次挥手：

### 第一次： Client -> Server (in.pcap)

```

> Frame 10: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) 0000 11 22 33 44 55 66 21 32 43 54 65 06 08 00 45 00 "3DUF!2 CTe ..E
> Ethernet II, Src: 21:32:43:54:65:06 (21:32:43:54:65:06), Dst: 11:22:33:44:55 0010 00 28 bd 05 40 00 80 06 76 07 c0 a8 a3 0a c0 a8 (@...v...
> Internet Protocol Version 4, Src: 192.168.163.10, Dst: 192.168.163.103 0020 a3 67 dc 7e ea 60 0e 07 3c d1 42 11 45 55 50 11 g ~...< B EUP...
> Transmission Control Protocol, Src Port: 56446, Dst Port: 60000, Seq: 4, Ack 0030 ff 6d 4f 84 00 00 m0 ...
  Source Port: 56446
  Destination Port: 60000
  [Stream index: 0]
  [Stream Packet Number: 9]
  > [Conversation completeness: Incomplete (29)]
    [TCP Segment Len: 0]
    Sequence Number: 4 (relative sequence number)
    Sequence Number (raw): 235355345
    [Next Sequence Number: 5 (relative sequence number)]
    Acknowledgment Number: 4 (relative ack number)
    Acknowledgment number (raw): 1108428117
    0101 .... = Header Length: 20 bytes (5)
  > [Flags: 0x011 (FIN, ACK)]
  Window: 65389
  [Calculated window size: 16739584]
  [Window size scaling factor: 256]
  Checksum: 0x4f84 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]

```

FIN=1, ACK=1, seq=4, ack\_seq=4, len=0。这是 Client 向 Server 的第一次挥手。在收到这个报文段前，Server 一直处于 ESTABLISHED 状态。

### 第二次及第三次挥手: Server -> Client (out.pcap)

```

Frame 6: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 21:32:43:54:65 (21:32:43:54:65)
Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10
Transmission Control Protocol, Src Port: 60000, Dst Port: 56446, Seq: 4, Ack: 5
Source Port: 60000
Destination Port: 56446
[Stream index: 0]
[Stream Packet Number: 5]
[Conversation completeness: Incomplete (30)]
[TCP Segment Len: 0]
Sequence Number: 4 (relative sequence number)
Sequence Number (raw): 29801
[Next Sequence Number: 5 (relative sequence number)]
Acknowledgment Number: 5 (relative ack number)
Acknowledgment number (raw): 235355346
0101 .... = Header Length: 20 bytes (5)
Flags: 0x811 (FIN, ACK)
Window: 65535
[Calculated window size: 65535]
[Window size scaling factor: -2 (no window scaling used)]
Checksum: 0x61ee [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
[Timestamps]

```

FIN=1,ACK=1,seq=4,ack\_seq=5

从抓包信息来看，第二次与第三次挥手合并了。事实上，我们实现的这个简易的用于服务端的 TCP 协议在代码层面也是没有第二次与第三次挥手间的 CLOSE\_WAIT 状态的。也就是说，在收到第一次挥手的这个报文后，Server 就不能再传输数据了，将直接从 ESTABLISHED 状态转变为 LAST\_ACK 状态。

代码层面: (in case TCP\_STATE\_ESTABLISHED:)

```

// TODO: 如果收到 FIN 报文，则增加 send_flags 相应标志位，并且进行状态转移
if (TCP_FLG_ISSET(recv_flags, TCP_FLG_FIN)){
    tcp_conn->ack++;
    send_flags |= TCP_FLG_ACK | TCP_FLG_FIN;
    tcp_conn->state = TCP_STATE_LAST_ACK;
}
break;

```

### 第四次: Client -> Server (in.pcap)

```

Frame 11: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
Ethernet II, Src: 21:32:43:54:65:06 (21:32:43:54:65:06), Dst: 11:22:33:44:55 (11:22:33:44:55)
Internet Protocol Version 4, Src: 192.168.163.10, Dst: 192.168.163.103
Transmission Control Protocol, Src Port: 56446, Dst Port: 60000, Seq: 5, Ack: 6
Source Port: 56446
Destination Port: 60000
[Stream index: 0]
[Stream Packet Number: 10]
[Conversation completeness: Incomplete (29)]
[TCP Segment Len: 0]
Sequence Number: 5 (relative sequence number)
Sequence Number (raw): 235355346
[Next Sequence Number: 5 (relative sequence number)]
Acknowledgment Number: 5 (relative ack number)
Acknowledgment number (raw): 1108428118
0101 .... = Header Length: 20 bytes (5)
Flags: 0x010 (ACK)
Window: 65389
[Calculated window size: 16739584]
[Window size scaling factor: 256]
Checksum: 0x4f83 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
[Timestamps]

```

ACK=1, seq=5, ack\_seq=5

Server 收到这条报文段前处于 LAST\_ACK 状态，收到这条报文段后进入 CLOSED 状态  
Client 端在发送这条报文段后等待 2MSL 后也会关闭连接。

**然后是 tcp\_server (集成测试):**

以太网适配器与测试网卡设置见 udp\_server 部分。

在终端中启动协议栈:

```
运行 E:\net-lab\build_CLion\tcp_server.exe
E:\net-lab\build_CLion\tcp_server.exe
Using interface \Device\NPF_{141507DC-CD4C-4550-84D0-0E11D848BC67}, my ip is 10.249.72.226.
```

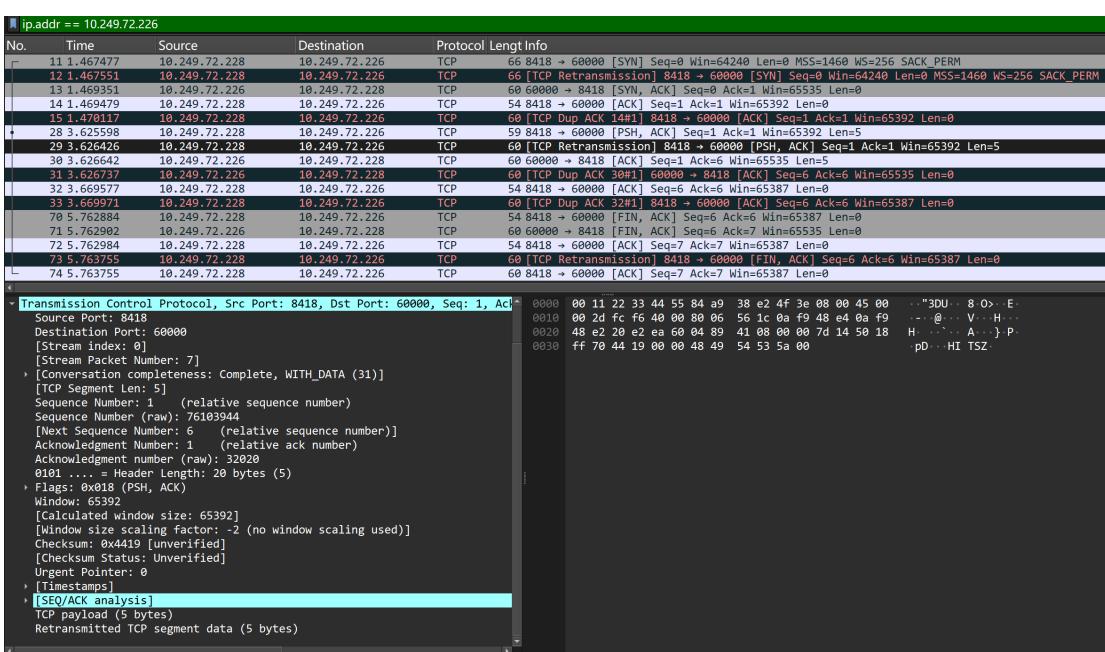
建立 TCP 连接并发送数据



此时可以在终端看到消息

```
运行 E:\net-lab\build_CLion\tcp_server.exe
E:\net-lab\build_CLion\tcp_server.exe
Using interface \Device\NPF_{141507DC-CD4C-4550-84D0-0E11D848BC67}, my ip is 10.249.72.226.
HITSZ
```

也可以在 Wireshark 中看到数据包的交互



具体分析类似 tcp\_test 中的三次握手四次挥手分析。不再赘述。

## 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。)

在终端启动 web 服务器

```
运行 web_server x
E:\net-lab\build_CLion\web_server.exe
Using interface \Device\NPF_{141507DC-CD4C-4550-84D0-0E11D848BC67}, my ip is 10.249.72.226.
```

访问 10.249.72.226



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](#)

访问第一个链接

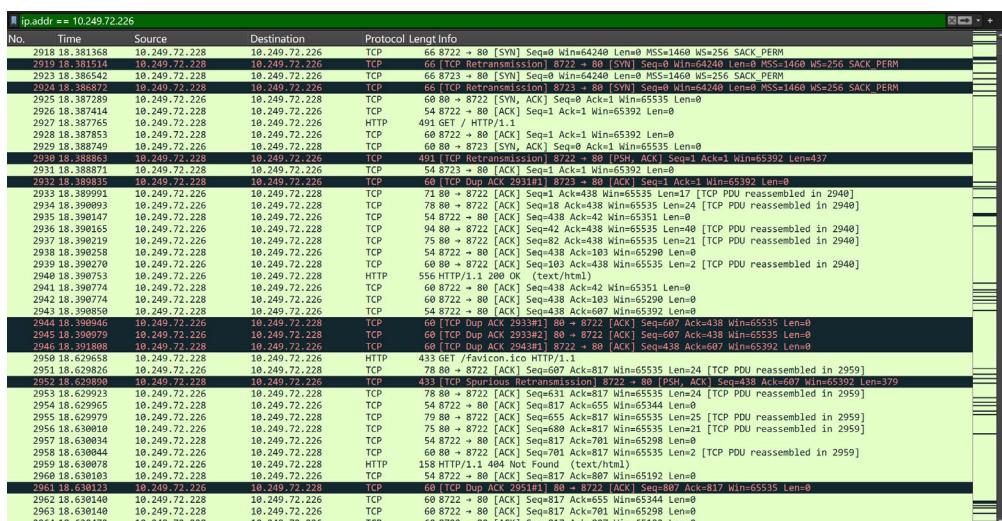


访问第二个链接



The resource specified is unavailable or nonexistent.

在 Wireshark 中也有数据包的交互:



## 请求报文:

```
+ 2927 18.387765      10.249.72.228    10.249.72.226   HTTP      491 GET / HTTP/1.1
1
Frame 2927: 491 bytes on wire (3928 bits), 491 bytes captured (3928 bits) on
Ethernet II, Src: LCFCElectron_e2:4f:3e (84:a9:38:e2:4f:3e), Dst: CMSYS_33: (00:11:22:33:44:55)
Internet Protocol Version 4, Src: 10.249.72.228, Dst: 10.249.72.226
Transmission Control Protocol, Src Port: 8722, Dst Port: 80, Seq: 1, Ack: 1,
Hypertext Transfer Protocol
    GET / HTTP/1.1\r\n
    Host: 10.249.72.226\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8\r\n
    Accept-Encoding: gzip,deflate\r\n
    Accept-Language: zh-CN,zh;q=0.9,en;q=0.8\r\n
\r\n
[Response in frame: 2940]
[Full request URL: http://10.249.72.226/]
```

请求由客户端发起，请求方法是 GET，请求 url 是 <http://10.249.72.226/>  
响应报文：

响应由服务端响应，响应状态码是 200 OK，响应体是 index.html 的文件内容

### 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。)

#### 1. 函数使用错误

倒不是逻辑上的用错了，而是函数名手滑敲错了。

```
//没有缓存的情况下，检查是否是请求自己MAC地址的ARP请求报文
if(hdr->opcode16 == swap16(ARP_REQUEST) &&
    memcmp(Dst:hdr->target_ip, Src:net_if_ip, Size:NET_IP_LEN)==0){
```

这个地方是比较自己的 IP 和 ARP 请求报文中的 Dst.IP 是否一致，但是我一开始把 memcmp 给打错成了 memcpy。导致我一度不能通过测试。

这种错误最难受的一点就是你很难从逻辑层次上来进行纠错。代码能跑通但是就是不对。从某种意义上来说我认为这种错误是比那些编译不能通过，跑不起来的错误还要致命的。因为要大范围进行排查的话会完全无从下手。

我这里是通过仔细对比 log 与 demo\_log，发现缓存区出现了问题，在这个地方再次经过仔细检查每一条语句，才发现了问题。

不过后来我想了想，这种大体逻辑上没错而在小细节上出的错误是可以通过 AI 来进行检查纠错的。至少这个错误 Deepseek 是能够检查出来的。

The screenshot shows a code editor with two sections: '问题:' (Problem:) and '修复:' (Fix:). In the '问题:' section, the code `if (memcpy(hdr->target\_ip, net\_if\_ip, NET\_IP\_LEN) == 0)` is highlighted with a red box, with a note: 'memcpy 返回值是目标地址指针，此处逻辑永远为假。应用 memcmp 比较IP是否相等。' (memcpy returns the target address pointer, the logic here is always false. Use memcmp to compare if the IPs are equal.) In the '修复:' section, the corrected code `if (memcmp(hdr->target\_ip, net\_if\_ip, NET\_IP\_LEN) == 0)` is shown.

#### 2. IP 复用问题

在我进行 udp\_server 的集成测试时，出现了一个问题。那就是每次我连接到 Server 后，第一次发送的数据“HITSZ”总是不能被正常接收，但是后面发送的都能正常接收。这引起了我的疑问。

打开 Wireshark 进行抓包分析，发现其实第一次是正常发送了的，但是问题是发送的目标 MAC 地址并不是自定义网卡的 00:11:22:33:44:55

Wireshark capture details for frame 3028:

- ip.addr == 10.249.72.226
- No. 3028 Time 241.445871 Source 10.249.72.228 Destination 10.249.72.226 Protocol UDP Length Info 47 49923 → 60000 Len=5
- 3184 271.207754 10.249.72.228 10.249.72.226 UDP 47 49923 → 60000 Len=5
- 3185 271.207848 10.249.72.228 10.249.72.226 UDP 60 49923 → 60000 Len=5
- 3192 273.856531 10.249.72.228 10.249.72.226 UDP 47 49923 → 60000 Len=5
- 3193 273.856599 10.249.72.228 10.249.72.226 UDP 60 49923 → 60000 Len=5
- 3186 271.211121 10.249.72.226 10.249.72.228 UDP 60 60000 → 49923 Len=5
- 3187 271.215380 10.249.72.226 10.249.72.228 UDP 60 60000 → 49923 Len=5
- 3194 273.862881 10.249.72.226 10.249.72.228 UDP 60 60000 → 49923 Len=5
- 3195 273.866823 10.249.72.226 10.249.72.228 UDP 60 60000 → 49923 Len=5

Frame details for frame 3028:

- Frame 3028: 47 bytes on wire (376 bits), 47 bytes captured (376 bits)
- Ethernet II, Src: LCFCElectron\_e2:4f:3e (84:a9:38:e2:4f:3e), Dst: IETF-VRRP-  
Destination: IETF-VRRP-VRID\_03 (00:00:5e:00:01:03)
- Source: LCFCElectron\_e2:4f:3e (84:a9:38:e2:4f:3e)
- Type: IPv4 (0x0800)
- [Stream index: 0]
- Internet Protocol Version 4, Src: 10.249.72.228, Dst: 10.249.72.226
- User Datagram Protocol, Src Port: 49923, Dst Port: 60000
- Data (5 bytes)

可以看到，后续几次都正常发送，并且收到了 Server 的回复。但是第一次的 MAC 地址与预期不符，为什么会出现这种情况呢？于是我查看了第一次发送附近的所有相关报文。

```
3026 241.444583 LCFCElectron_e2:4f:... Broadcast ARP 42 Who has 10.249.72.226? Tell 10.249.72.228
3027 241.445851 IETF-VRRP-VRID_03 LCFCElectron_e2:4f:... ARP 60 10.249.72.226 is at 00:00:5e:00:01:03
3028 241.445871 10.249.72.228 10.249.72.226 UDP 47 49923 → 60000 Len=5
3029 241.445953 CIMSYS_33:44:55 LCFCElectron_e2:4f:... ARP 60 10.249.72.226 is at 00:11:22:33:44:55
3030 242.669186 10.249.72.228 36.150.102.188 TLSv1.2 192 Application Data
3031 242.701792 36.150.102.188 10.249.72.228 TLSv1.2 186 Application Data
3032 242.717116 10.249.72.228 36.150.102.188 TCP 54 1618 → 7826 [ACK] Seq=1657 Ack=1586 Win=512 Len=0
...
> Frame 3027: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
- Ethernet II, Src: IETF-VRRP-VRID_03 (00:00:5e:00:01:03), Dst: LCFCElectron_e2:4f:3e (84:a9:38:e2:4f:3e)
  Destination: LCFCElectron_e2:4f:3e (84:a9:38:e2:4f:3e)
  Source: IETF-VRRP-VRID_03 (00:00:5e:00:01:03)
  Type: ARP (0x0806)
    [Stream index: 0]
    Padding: 0000000000000000000000000000000000000000000000000000000000000000
> Address Resolution Protocol (reply)
[Duplicate IP address detected for 10.249.72.226 (00:00:5e:00:01:03) - also in use by 00:11:22:33:44:55 (frame 1466)]
  ↳ [Frame showing earlier use of IP address: 1466]
    [Seconds since earlier frame seen: 108]
```

可以看到我们收到了两条 ARP 回应报文，其中第二条是我们设置的自定义网卡的。而第一条来自 MAC 地址 00:00:5e:00:01:03。

同时第一条报文还附带了一条消息：

[Duplicate IP address detected for 10.249.72.226 (00:00:5e:00:01:03) - also in use by 00:11:22:33:44:55 (frame 1466)]

这个错误消息明确表示网络中出现了 IP 地址冲突。

具体来说，IP 地址 10.249.72.226 被两个不同的设备同时使用了：

设备 A: MAC 地址 00:00:5e:00:01:03 (报告冲突的设备)

设备 B: MAC 地址 00:11:22:33:44:55 (被检测到的冲突设备)

这个时候，我反应了过来，我所连接的局域网是这一整个校园网，我被分配到的私网 IP 是 10.249.72.228，于是我按实验要求设置了自定义网卡的 IP 地址是 10.249.72.226。但是这个地址是我手动配置的。而我并不能确认这个 IP 是否是被局域网内的 DHCP 主机提供给了另一个连接到局域网的设备。因此产生了冲突。

在这次实验中，第一次通过 UDP 发送数据时，MAC 地址 00:11:22:33:44:55 的 ARP 响应报文还未到达，因此按照 ARP 表中的数据，发给了 MAC 地址 00:00:5e:00:01:03。

之后自定义网卡发送的 ARP 响应报文到达，ARP 表把 IP=10.249.72.226 对应的 MAC 地址修改为了 00:11:22:33:44:55。此后几次发送时，ARP 表已有对应的项，因此直接读取 MAC 地址进行发送，流程正确。

在后续的 tcp server 测试中，我就在启动协议栈前先试着 ping 了 10.249.72.226

```
正在 Ping 10.249.72.226 具有 32 字节的数据：  
请求超时。  
请求超时。  
请求超时。  
请求超时。  
  
10.249.72.226 的 Ping 统计信息：  
数据包：已发送 = 4, 已接收 = 0, 丢失 = 4 (100% 丢失)
```

于是我开始进行测试，测试流程一切正常，没有再发生 udp server 测试时的问题

## 四、实验收获和建议

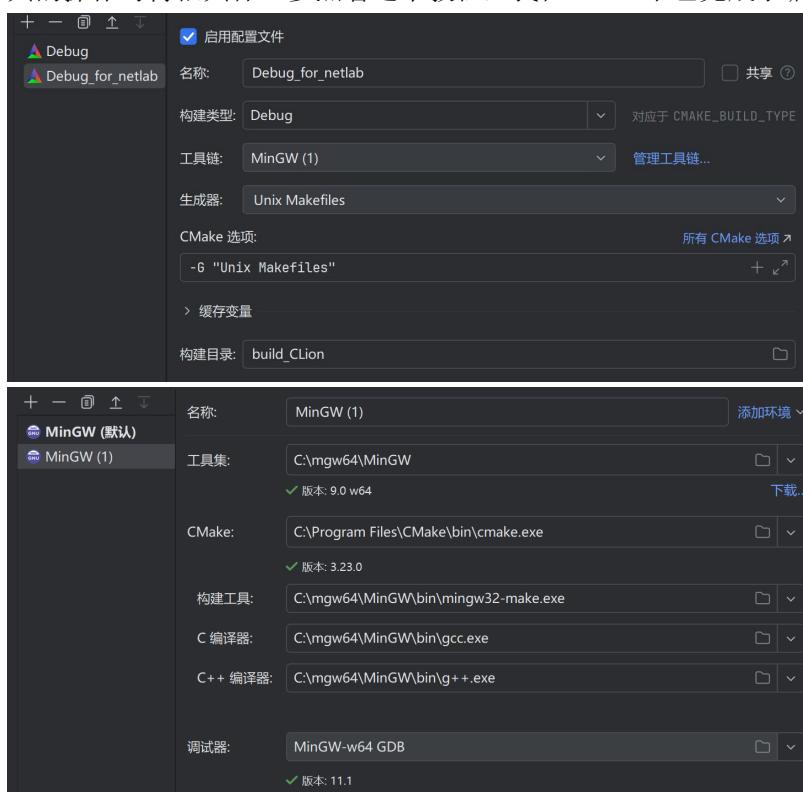
(总结配置实验及协议栈实验过程中的实践收获,结合实操体验针对性提出实验流程优化及环境完善建议,为后续实验教学与研究的迭代改进提供参考依据。)

实践收获还挺大的。

首先,我其实还是第一次用 C 语言写这种工程项目类的代码。在此之前,我写过的工程项目型的代码都是基于 Java 或者 Python 的。对我来说,C/C++更多的是在写单个功能的程序,也就是更多的是面向算法题和竞赛的。这次使用 C 语言来写这么一个协议栈项目的代码,感觉最大的收获就是切身体会了面向工程和面向竞赛的区别。对于竞赛来说,效率和结果至上。只要能够 AC,一切都好说。全局数组随便开,码风狂乱也无所谓。但是在写工程项目时,不但要考虑时间效率,还要考虑空间效率。同时,良好的注释习惯不但能让代码的可读性更好,在自己进行 debug 的时候也能更好的明白自己之前想的什么,写的又是什么。

其次,这个实验也充分加深了我对 TCP/IP 协议栈的理解。在进行理论学习时,对 Eth, ARP, IP 等帧 (/分组/数据段) 的首部结构都是以纯记忆为主,而各种协议在面对不同报文时的反应也是以记忆为主,最多了解一个大概的流程,但是底层具体怎么实现的还是有层隔离感,会觉得还是有些晦涩。而当自己真正上手实现过一次对应的协议操作后,对不同协议工作时干了什么会有更深刻的理解。

此外,这次实验其实还有很多工具使用上的收获。众所周知,对于写代码来说,尤其是对于新手来说,配置环境这个操作是真的又复杂又难。我相信有很多人刚读大学时听别人说 VSCode 好用,开源,轻量化,适配性好。但是在安装完 VSCode 后首先就要面临的一座大山就是配置环境。即使是在网上搜索各种教程,还是会有很多地方不懂,以至于有同学会在 VSCode 里编辑完后复制粘贴到 Dev-Cpp 中进行编译运行,就是因为不会配环境。本门课中指导书里附录 B: Windows 开发环境搭建这一章我认为对这些同学来说非常友好。对配置编译工具的操作写得很具体。参照着这个教程,我在 Clion 中也完成了编译运行的配置。



在以前安装 MinGW 时，我是纯粹按照网上的教程依葫芦画瓢安装然后进行环境配置的，具体该怎么使用我是没有懂的，这次是我第一次自己完成了这样的配置设置。然后是 Wireshark 这个工具，我也认为非常好用。

提到工具，在这里我就提一个小小的建议。

在我看来，这一整个协议栈的实验可以看作一个项目的迭代开发，那么就会涉及到版本控制（虽然这个工程项目相对来说很小，也不需要多人协作）。这个实验我认为其实很适合拿来当做入门 git 与 github 的素材。我觉得可以把版本控制的相关内容也加入到这门课程里。这样不仅能让同学们学到更多实用的知识，甚至在提交作业的时候可以直接提交一个仓库链接就可以了，非常方便。具体来说也不需要教的面面俱到，指导书里加一个附录，教一下最基本的仓库创建，关联远程仓库，.gitignore 的配置，add，commit (-m)，push 等实用操作就可以了，对于感兴趣的同学来说，这样的基础知识也能为他们学习进阶运用打好基础。

至少对我本人来说，这个实验就是熟悉 git 与 github 的第一个案例：

