



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: 计科十班

学生学号: 220111012

学生姓名: 王靳

评阅教师: _____

报告成绩: _____

实验与创新实践教育中心制

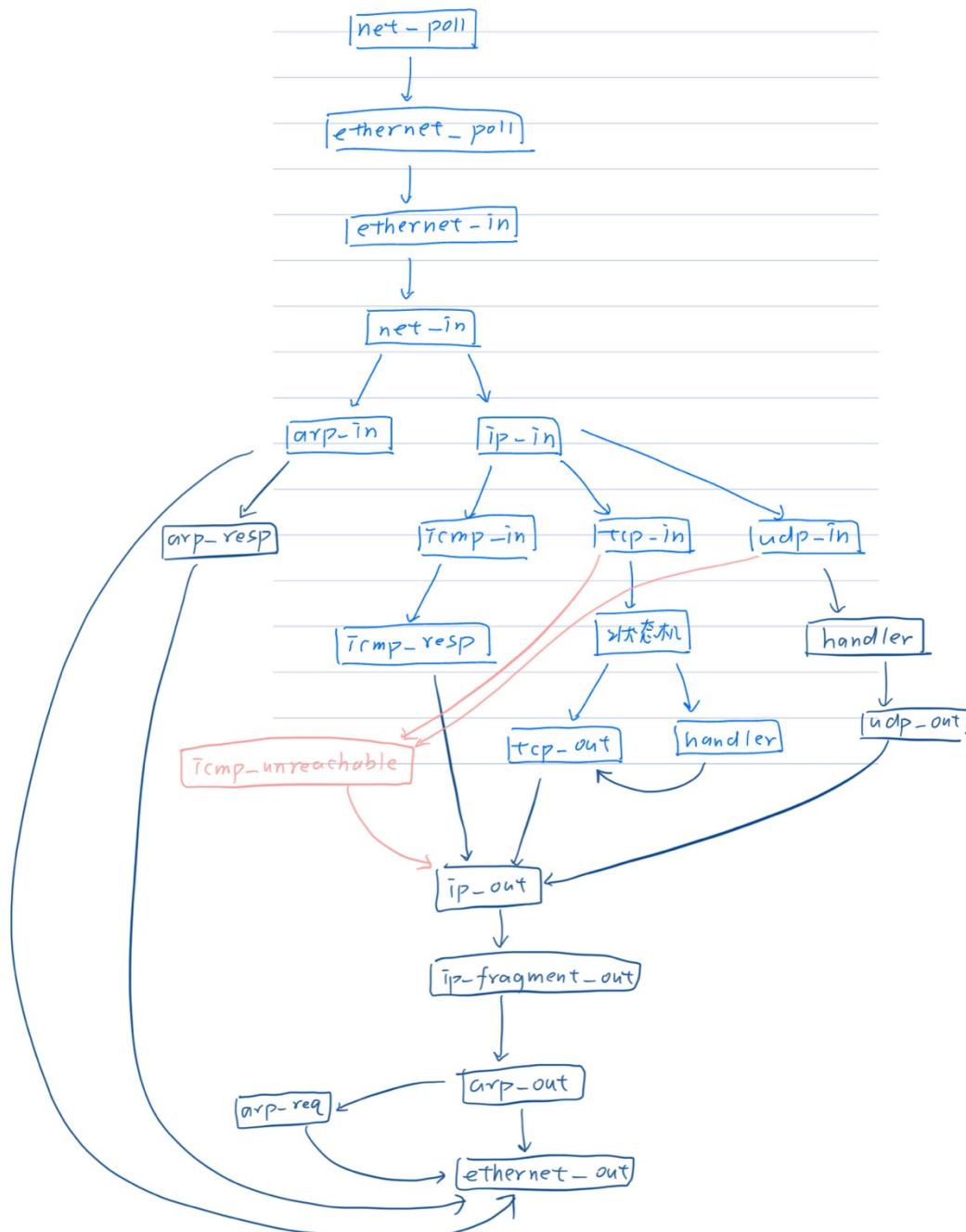
2025 年 3 月

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

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



2. Eth 协议详细设计

(描述以太网 (Eth) 协议的数据封装与解封装过程等。)

```

15  /**
16   * @brief 处理一个收到的数据包
17   *
18   * @param buf 要处理的数据包
19   */
20 void ethernet_in(buf_t *buf) {
21     ether_hdr_t *ether_hdr = (ether_hdr_t *)buf->data;
22     uint8_t src_mac[6];
23     memcpy(dest: src_mac, src: ether_hdr->src, n: NET_MAC_LEN);
24     uint16_t protocol = ntohs(netshort: ether_hdr->protocol16);
25     buf_remove_header(buf, len: sizeof(ether_hdr_t));
26     net_in(buf, protocol, src: src_mac);
27 }
28 /...

```

去掉以太网帧的头部信息，交给上一层协议栈处理

```

28 /**
29  * @brief 处理一个要发送的数据包
30  *
31  * @param buf 要处理的数据包
32  * @param mac 目标MAC地址
33  * @param protocol 上层协议
34  */
35 void ethernet_out(buf_t *buf, const uint8_t *mac, net_protocol_t protocol) {
36     if (buf->len < ETHERNET_MIN_TRANSPORT_UNIT) {
37         int pad_len = ETHERNET_MIN_TRANSPORT_UNIT - buf->len;
38         buf_add_padding(buf, len: pad_len);
39     }
40     buf_add_header(buf, len: sizeof(ether_hdr_t));
41     ether_hdr_t *ether_hdr = (ether_hdr_t *)buf->data;
42     memcpy(dest: ether_hdr->dst, src: mac, n: NET_MAC_LEN);
43     memcpy(dest: ether_hdr->src, src: net_if_mac, n: NET_MAC_LEN);
44     ether_hdr->protocol16 = htons(hostshort: protocol);
45 #if 0
46     putchar('\n');
47     for (int i = 0; i < buf->len; i++) {
48         printf("%02x ", buf->data[i]);
49     }
50     putchar('\n');
51 #endif
52     driver_send(buf);
53 }
54 /...

```

给上层传下来的 pdu 加上以太网帧头部，头部信息包括：目的 mac 地址，源 mac 地址，上层使用的协议类型

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。)

```

89  /**
90  * @brief 处理一个收到的数据包
91  *
92  * @param buf 要处理的数据包
93  * @param src_mac 源mac地址
94  */
95 void arp_in(buf_t *buf, const uint8_t *src_mac) {
96     // TO-DO
97     // if (buf->len < 2 + 2 + 1 + 1 + 2) {
98     //     // DROP
99     //     return;
100    //}
101    arp_pkt_t *pkt = (arp_pkt_t *)buf->data;
102    // some check
103    if (pkt->hw_type16 != htons(hostshort: ARP_HW_ETHER)) {
104        return;
105    }
106    if (pkt->pro_type16 != htons(hostshort: NET_PROTOCOL_IP)) {
107        return;
108    }
109    if (pkt->hw_len != NET_MAC_LEN) {
110        return;
111    }
112    if (pkt->pro_len != NET_IP_LEN) {
113        return;
114    }
115    if (pkt->opcode16 != htons(hostshort: ARP_REQUEST) && pkt->opcode16 != htons(hostshort: ARP_REPLY)) {
116        return;
117    }
118    map_set(map: &arp_table, key: pkt->sender_ip, value: pkt->sender_mac);
119    if (memcmp(s1: pkt->sender_mac, s2: src_mac, n: NET_MAC_LEN) != 0) {
120        return;
121    }
122    // refresh the arp_buf
123    buf_t *cache = map_get(map: &arp_buf, key: pkt->sender_ip);
124    if (cache) {
125        ethernet_out(buf: cache, mac: pkt->sender_mac, protocol: NET_PROTOCOL_IP);
126        map_delete(map: &arp_buf, key: pkt->sender_ip);
127    } else {
128        if (pkt->opcode16 == htons(hostshort: ARP_REQUEST) && memcmp(s1: pkt->target_ip, s2: net_if_ip, n: NET_IP_LEN) == 0) {
129            arp_resp(target_ip: pkt->sender_ip, target_mac: pkt->sender_mac);
130        }
131    }
132}

```

收到 arp 信息，有两种可能：一种是对方发来的 arp 请求，一种是对方发来的 arp 相应。

- 1) 如果是 arp 响应，说明是对上一次自己发送的 arp 查询请求的响应，就有两个动作：
 1. 检查 arp_buf，发送上次因为 mac 地址缺失，未发出去的包，2. 更新自己的 arp_table(ip -> mac)
- 2) 如果是 arp 请求，那么就响应 arp 请求。

```

134  /**
135  * @brief 处理一个要发送的数据包
136  *
137  * @param buf 要处理的数据包
138  * @param ip 目标ip地址
139  * @param protocol 上层协议
140  */
141 void arp_out(buf_t *buf, const uint8_t *ip) {
142     // T0-D0
143     uint8_t *mac = map_get(map: &arp_table, key: ip);
144     if (mac) {
145         ethernet_out(buf, mac, protocol: NET_PROTOCOL_IP);
146     } else {
147         buf_t *cache = map_get(map: &arp_buf, key: ip);
148         if (!cache) {
149             map_set(map: &arp_buf, key: ip, value: buf);
150             arp_req(target_ip: ip);
151         }
152     }
153 }

```

所有的 ip_out 都必须经过 arp_out，因为要封装以太网帧，以太网帧要求填写目的 mac 地址字段。如果 arp_table 中能找到目的 ip 对应的 mac 地址，那么可以直接发送；如果 arp_table 中找不到目的 ip 对应的 mac 地址，那么就需要发起 arp 查询请求，等收到 arp 响应报文的时候再发送（请看上面关于 arp_in 的分析）

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。)

```

15  /**
16   * @brief 处理一个收到的数据包
17   *
18   * @param buf 要处理的数据包
19   * @param src_mac 源mac地址
20   */
21 void ip_in(buf_t *buf, const uint8_t *src_mac) {
22     // TO-DO
23     ip_hdr_t *ip_hdr = (ip_hdr_t *)buf->data;
24     uint16_t ip_hdr_len = ip_hdr->len * IP_HDR_LEN_PER_BYTE;
25     uint16_t total_len = ntohs(netshort: ip_hdr->total_len16);
26     if (total_len < ip_hdr_len) {
27         return; // drop, packet shorter than header
28     }
29     if (ip_hdr->version != IP_VERSION_4) {
30         return; // drop
31     }
32     if (buf->len < total_len) {
33         return; // drop, buffer shorter than packet length
34     }
35     // check checksum
36     uint16_t checksum = ip_hdr->hdr_checksum16;
37     ip_hdr->hdr_checksum16 = 0;
38     if (checksum16(data: (uint16_t *)ip_hdr, len: ip_hdr_len) != checksum) {
39         return; // drop, checksum error
40     }
41     ip_hdr->hdr_checksum16 = checksum;
42     // check destination ip
43     if (memcmp(s1: ip_hdr->dst_ip, s2: net_if_ip, n: 4) != 0) {
44         return; // drop, packet not for me
45     }
46     // remove padding
47     int pad_len = buf->len - total_len;
48     if (pad_len > 0) {
49         buf_remove_padding(buf, len: pad_len);
50     }
51     buf_remove_header(buf, len: ip_hdr_len);
52     // call protocol handler
53     uint8_t protocol = ip_hdr->protocol;
54     int ret = net_in(buf, protocol, src: ip_hdr->src_ip);
55     if (ret == -1) {
56         buf_add_header(buf, len: ip_hdr_len); // restore header
57         icmp_unreachable(recv_buf: buf, src_ip: ip_hdr->src_ip, code: ICMP_CODE_PROTOCOL_UNREACH);
58     }
59 }
```

先会对收到的 ip 数据包做合法性检查（长度、校验和、版本）。我这里没有做 ip 重组，因此我这里默认 ip datagram 就是 ip packet，因此 ip header 的 buf_len==ip_hdr.total_len + padding_len。Padding 可能有，可能没有，如果 buf_len==ip_hdr.total_len，那么就没有 padding，反之则有。Padding 移除是必须的，因为我们这里做的 padding 对上层是透明的。去除 ip_padding 和 ip_hdr 后，交给上层处理。如果上层没有可交付的协议栈，那么就返回 icmp_unreachable 信息。

```

91  /**
92  * @brief 处理一个要发送的ip数据包
93  *
94  * @param buf 要处理的包
95  * @param ip 目标ip地址
96  * @param protocol 上层协议
97  */
98 void ip_out(buf_t *buf, const uint8_t *ip, net_protocol_t protocol) {
99    // TO-DO
100   static int id = 0;
101 #ifndef IP_FRAG_TEST
102   id = random();
103 #endif
104   // id++;           // counter for ip id
105   int offset = 0; // accumulate offset
106   int buf_len = buf->len;
107   for (; buf_len > IP_MAX_PAYLOAD; buf_len -= IP_MAX_PAYLOAD, offset += IP_MAX_PAYLOAD) {
108     buf_t frag_buf;
109     buf_init(buf: &frag_buf, len: IP_MAX_PAYLOAD);
110     memcpy(dest: frag_buf.data, src: buf->data + offset, n: IP_MAX_PAYLOAD);
111     ip_fragment_out(buf: &frag_buf, ip, protocol, id, offset, mf: 1);
112   }
113   buf_remove_header(buf, len: offset);
114   ip_fragment_out(buf, ip, protocol, id, offset, mf: 0);
115 }

```

Ip_out 这里处理主要是分片处理，用了一个 for 循环来处理的。(个人感觉实现较为简洁)

```

61 /**
62 * @brief 处理一个要发送的ip分片
63 *
64 * @param buf 要发送的分片
65 * @param ip 目标ip地址
66 * @param protocol 上层协议
67 * @param id 数据包id
68 * @param offset 分片offset, 必须被8整除
69 * @param mf 分片mf标志, 是否有下一个分片
70 */
71 void ip_fragment_out(buf_t *buf, const uint8_t *ip, net_protocol_t protocol, int id, uint16_t offset, int mf) {
72    // TO-DO
73    buf_add_header(buf, len: sizeof(ip_hdr_t));
74    ip_hdr_t *ip_hdr = (ip_hdr_t *)buf->data;
75    ip_hdr->version = IP_VERSION_4;
76    ip_hdr->hdr_len = sizeof(ip_hdr_t) / IP_HDR_LEN_PER_BYTE;
77    ip_hdr->tos = 0;
78    ip_hdr->total_len16 = htons(hostshort: buf->len);
79    ip_hdr->id16 = htons(hostshort: id);
80    ip_hdr->flags_fragment16 = htons(hostshort: (offset >> 3) | (mf ? IP_MORE_FRAGMENT : 0)); // FIXME
81    ip_hdr->ttl = 64;
82    ip_hdr->protocol = protocol;
83    memcpy(dest: ip_hdr->src_ip, src: net_if_ip, n: 4);
84    memcpy(dest: ip_hdr->dst_ip, src: ip, n: 4);
85    ip_hdr->hdr_checksum16 = 0;
86    int checksum = checksum16(data: (uint16_t *)ip_hdr, len: sizeof(ip_hdr_t));
87    ip_hdr->hdr_checksum16 = checksum;
88    arp_out(buf, ip);
89 }

```

Ip_fragment_out 就是实际上的发送。

```
70  /**
71  * @brief 计算16位校验和
72  *
73  * @param buf 要计算的数据包
74  * @param size 要计算的长度
75  * @return uint16_t 校验和
76  */
77 uint16_t checksum16(uint16_t *data, size_t len) {
78     uint32_t sum = 0;
79     while (len > 1) {
80         sum += *data++;
81         len -= 2;
82         while (sum >> 16)
83             sum = (sum >> 16) + (sum & 0xffff);
84     }
85     if (len)
86         sum += *(uint8_t *)data;
87     while (sum >> 16)
88         sum = (sum >> 16) + (sum & 0xffff);
89     sum = (uint16_t)sum;
90     sum = ~sum;
91     return (uint16_t)sum;
92 }
```

校验和计算，这里传参是值得注意的，虽说传入的是 `len`（数组元素的个数），但传入的是数组占用的内存大小。

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。)

```

31  /**
32   * @brief 处理一个收到的数据包
33   *
34   * @param buf 要处理的数据包
35   * @param src_ip 源ip地址
36   */
37  void icmp_in(buf_t *buf, const uint8_t *src_ip) {
38      icmp_hdr_t *icmp_hdr = (icmp_hdr_t *)buf->data;
39      if (buf->len < sizeof(icmp_hdr_t)) {
40          return; // drop, non integrity
41      }
42      putchar(c: '\n');
43      // printf("icmp_in: type = %d, code = %d\n", icmp_hdr->type, icmp_hdr->code);
44      if (icmp_hdr->type == 0x8 && icmp_hdr->code == 0x0) {
45          icmp_resp(req_buf: buf, src_ip);
46      }
47 }

```

Icmp_in，只会处理 icmp echo 请求。然后调用 icmp_resp

```

11  /**
12   * @brief 发送icmp响应
13   *
14   * @param req_buf 收到的icmp请求包
15   * @param src_ip 源ip地址
16   */
17  static void icmp_resp(buf_t *req_buf, const uint8_t *src_ip) {
18      // TO-DO
19      icmp_hdr_t *req_hdr = (icmp_hdr_t *)req_buf->data;
20      int len = req_buf->len;
21      buf_init(buf: &txbuf, len);
22      icmp_hdr_t *icmp_hdr = (icmp_hdr_t *)txbuf.data;
23      memcpy(dest: icmp_hdr, src: req_hdr, n: len);
24      icmp_hdr->type = 0x0; // echo reply
25      icmp_hdr->code = 0x0;
26      icmp_hdr->checksum16 = 0;
27      icmp_hdr->checksum16 = checksum16(data: (uint16_t *)icmp_hdr, len);
28      ip_out(buf: &txbuf, ip: src_ip, protocol: NET_PROTOCOL_ICMP);
29 }

```

Icmp_resp 同样只会发送 icmp echo 响应。

```

49  /**
50  * @brief 发送icmp不可达
51  *
52  * @param recv_buf 收到的ip数据包
53  * @param src_ip 源ip地址
54  * @param code icmp code, 协议不可达或端口不可达
55  */
56 void icmp_unreachable(buf_t *recv_buf, const uint8_t *src_ip, icmp_code_t code) {
57     // T0-D0
58     buf_init(buf: &txbuf, len: sizeof(icmp_hdr_t) + sizeof(ip_hdr_t) + 8);
59     icmp_hdr_t *icmp_hdr = (icmp_hdr_t *)txbuf.data;
60     icmp_hdr->type = 0x3; // 协议不可达, 端口不可达
61     icmp_hdr->code = code;
62     icmp_hdr->checksum16 = 0;
63     icmp_hdr->id16 = 0;
64     icmp_hdr->seq16 = 0;
65     memcpy(dest: icmp_hdr + 1, src: recv_buf, n: sizeof(ip_hdr_t) + 8);
66     uint16_t len = sizeof(icmp_hdr_t) + sizeof(ip_hdr_t) + 8;
67     icmp_hdr->checksum16 = checksum16(data: (uint16_t *)icmp_hdr, len);
68     ip_out(buf: &txbuf, ip: src_ip, protocol: NET_PROTOCOL_ICMP);
69 }

```

Icmp 还会发送 unreachable, 当发生: 没有可交付处理的协议时, 会发生 unreachable; 当 udp, tcp 没有响应的监听进程的时候, 会发生 icmp unreachable

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。)

```

20  /**
21  * @brief 处理一个收到的udp数据包
22  *
23  * @param buf 要处理的包
24  * @param src_ip 源ip地址
25  */
26 void udp_in(buf_t *buf, const uint8_t *src_ip) {
27     // T0-D0
28     if (buf->len < sizeof(udp_hdr_t)) {
29         return; // drop
30     }
31     udp_hdr_t *udp_hdr = (udp_hdr_t *)buf->data;
32     uint16_t src_port = ntohs(netshort: udp_hdr->src_port16);
33     uint16_t dst_port = ntohs(netshort: udp_hdr->dst_port16);
34     uint16_t total_len = ntohs(netshort: udp_hdr->total_len16);
35     if (buf->len < total_len) {
36         return; // drop
37     }
38     if (udp_hdr->checksum16 != 0) {
39         uint16_t checksum16 = udp_hdr->checksum16;
40         udp_hdr->checksum16 = 0;
41         uint16_t cal_checksum = transport_checksum(protocol: NET_PROTOCOL_UDP, buf, src_ip, dst_ip: net_if_ip);
42         if (checksum16 != cal_checksum) {
43             return; // drop
44         }
45     }
46     udp_handler_t *handler = map_get(map: &udp_table, key: &dst_port);
47     if (handler) {
48         uint8_t *data = buf->data + sizeof(udp_hdr_t);
49         size_t len = buf->len - sizeof(udp_hdr_t);
50         (*handler)(data, len, src_ip, src_port);
51     } else {
52         buf_add_header(buf, len: sizeof(ip_hdr_t));
53         icmp_unreachable(recv_buf: buf, src_ip, code: 3);
54     }
55 }
```

Udp 协议栈，收到要处理的数据包，会先进行合法性检查（长度、校验和如果不为 0），再根据目的端口，从 udp_table 中找到监听此端口的进程（其实也就是 handler），去掉头部信息并交付给相应的 handler；如果找不到 handler，发送 icmp_unreachable

```

57 /**
58  * @brief 处理一个要发送的数据包
59  *
60  * @param buf 要处理的包
61  * @param src_port 源端口号
62  * @param dst_ip 目的ip地址
63  * @param dst_port 目的端口号
64  */
65 void udp_out(buf_t *buf, uint16_t src_port, const uint8_t *dst_ip, uint16_t dst_port) {
66     // T0-D0
67     buf_add_header(buf, len: sizeof(udp_hdr_t));
68     udp_hdr_t *udp_hdr = (udp_hdr_t *)buf->data;
69     udp_hdr->src_port16 = src_port;
70     udp_hdr->dst_port16 = dst_port;
71     udp_hdr->total_len16 = htons(hostshort: buf->len + sizeof(udp_hdr_t));
72     udp_hdr->checksum16 = 0;
73     uint16_t checksum = transport_checksum(protocol: NET_PROTOCOL_UDP, buf, src_ip: net_if_ip, dst_ip);
74     udp_hdr->checksum16 = checksum;
75     ip_out(buf, ip: dst_ip, protocol: NET_PROTOCOL_UDP);
76 }
```

朴实无华的 udp_out

```

104  /**
105   * @brief 计算传输层协议（如TCP/UDP）的校验和
106   *
107  * @param protocol 传输层协议号（如NET_PROTOCOL_UDP、NET_PROTOCOL_TCP）
108  * @param buf 待计算的数据包缓冲区
109  * @param src_ip 源IP地址
110  * @param dst_ip 目的IP地址
111  * @return uint16_t 计算得到的16位校验和
112  */
113 uint16_t transport_checksum(uint8_t protocol, buf_t *buf, const uint8_t *src_ip, const uint8_t *dst_ip) {
114     // TO-DO
115     size_t udp_len = buf->len;
116     // saved the old ip_hdr
117     buf_add_header(buf, len: sizeof(peso_hdr_t));
118     peso_hdr_t *udp_peso_hdr = (peso_hdr_t *)buf->data;
119
120     // 暂存IP首部
121     peso_hdr_t ip_hdr; // partial ip_hdr
122     memcpy(dest: &ip_hdr, src: udp_peso_hdr, n: sizeof(peso_hdr_t));
123
124     // 填充伪首部
125     memcpy(dest: udp_peso_hdr->src_ip, src: src_ip, n: NET_IP_LEN);
126     memcpy(dest: udp_peso_hdr->dst_ip, src: dst_ip, n: NET_IP_LEN);
127     udp_peso_hdr->placeholder = 0;
128     udp_peso_hdr->protocol = protocol;
129     udp_peso_hdr->total_len16 = htons(hostshort: udp_len);
130
131     int paddled = 0;
132     if (buf->len % 2) {
133         buf_add_padding(buf, len: 1);
134         paddled = 1;
135     }
136
137     // 计算校验和
138     uint16_t checksum = checksum16(data: (uint16_t *)buf->data, len: buf->len);
139
140     // 恢复IP数据
141     memcpy(dest: udp_peso_hdr, src: &ip_hdr, n: sizeof(peso_hdr_t));
142     buf_remove_header(buf, len: sizeof(peso_hdr_t));
143
144     if (paddled)
145         buf_remove_padding(buf, len: 1);
146
147     return checksum;
148 }

```

上面是校验和计算，会先将可能会因为伪首部计算而被破坏的信息暂存到栈上。在校验和计算完成以后恢复。同时还会判断计算校验和的数据的内存大小是否为偶数，因为是16bit校验和，所以如果是奇数的内存大小，那么就会在最末尾处补0。最后传入校验和计算函数，这个校验和计算函数只是单纯的做计算。

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。)

三次握手：作为服务端，我们并不会主动发起 tcp 连接。因此，对于服务端来说，他的状态转移是：初始状态是 LISTEN，当收到第一个 seq=x 的 SYN 数据包时，进入 SYN_RCVD 状态，并发送 ack=x+1, seq=y 的 ACK, SYN 数据包；当收到 seq=x+1, ack=y+1 的 ACK 数据包时，进入 ESTABLISHED 状态。

```

192 // 根据当前 TCP 连接的状态进行不同的处理
193 switch (tcp_conn->state) {
194     case TCP_STATE_LISTEN:
195         // TODO: 仅在收到连接报文时 (SYN报文) 才做出处理, 否则直接返回
196         if (!TCP_FLG_ISSET(recv_flags, TCP_FLG_SYN)) {
197             return;
198         }
199     // TODO: 初始化 TCP 连接上下文 (tcp_conn结构体) 的seq字段
200 #if TEST
201         tcp_conn->seq = 0; // 方便与参考答案对比
202 #else
203         tcp_conn->seq = tcp_generate_initial_seq();
204 #endif
205         // TODO: 填写回复标志 send_flags
206         send_flags = TCP_FLG_SYN | TCP_FLG_ACK;
207         // TODO: 填写 TCP 连接上下文 (tcp_conn结构体) 的ack字段
208         tcp_conn->ack = bytes_in_flight(len: remote_seq, flags: send_flags);
209         // TODO: 进行状态转移
210         tcp_conn->state = TCP_STATE_SYN_RECEIVED;
211         break;
212
213     case TCP_STATE_SYN_RECEIVED:
214         // TODO: 仅在收到确认报文时 (ACK报文) 才做出处理, 否则直接返回
215         if (!TCP_FLG_ISSET(recv_flags, TCP_FLG_ACK)) {
216             return;
217         }
218         tcp_conn->seq += 1;
219         // TODO: 进行状态转移
220         tcp_conn->state = TCP_STATE_ESTABLISHED;
221         break;
222
223     case TCP_STATE_ESTABLISHED:
224         // 未收到顺序包, 丢弃并发送重复 ACK
225         if (remote_seq != tcp_conn->ack) {
226             buf_init(buf: &txbuf, len: 0);
227             tcp_out(tcp_conn, buf: &txbuf, src_port: host_port, dst_ip: remote_ip, dst_port: remote_port, flags: TCP_FLG_ACK);
228             return;
229         }
230         // TODO: 计算接收到的数据长度, 更新 ACK
231         data_len = buf->len - tcp_hdr_sz;
232         tcp_conn->ack += bytes_in_flight(len: data_len, flags: recv_flags); // include FIN
233         // TODO: 如果接收到携带数据, 则填写回复标志 send_flags 发送ACK
234         if (data_len > 0) {
235             send_flags = TCP_FLG_ACK;
236         }
237         // 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
238         if (TCP_FLG_ISSET(recv_flags, TCP_FLG_FIN)) {
239             send_flags = TCP_FLG_ACK;
240             tcp_conn->state = TCP_STATE_LAST_ACK;
241         }
242         break;

```

四次挥手：对于本次实验，我们的服务器并不会主动的断开连接，总是客户端先发起断开连接。当服务器收到 seq=u 的 FIN 数据包时，进入 last_ack 状态，并发送 ack=u+1, seq=v 的 FIN, ACK 数据包。我们这里是一个简化版的 tcp 协议栈，我们将 close_wait、last_ack 合并成同一个状态（完整版应该是先进入 close_wait 状态，然后作为服务端我们还能再发送一些数据，发送完成再进入 last_ack 状态）；当收到 seq=u+1, ack+v+1 的 ACK 数据包时，断开连接，或者说是进入了 CLOSED 状态。

```

223     case TCP_STATE_ESTABLISHED:
224         // 未收到顺序包，丢弃并发送重置 ACK
225         if (remote_seq != tcp_conn->ack) {
226             buf_init(buf: &txbuf, len: 0);
227             tcp_out(tcp_conn, buf: &txbuf, src_port: host_port, dst_ip: remote_ip, dst_port: remote_port, flags: TCP_FLG_ACK);
228             return;
229         }
230         // TODO: 计算接收到的数据长度，更新 ACK
231         data_len = buf->len - tcp_hdr_sz;
232         tcp_conn->ack += bytes_in_flight(len: data_len, flags: recv_flags); // include FIN
233         // TODO: 如果接收报文携带数据，则填写回复标志 send_flags 发送ACK
234         if (data_len > 0) {
235             send_flags = TCP_FLG_ACK;
236         }
237         // 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
238         if (TCP_FLG_ISSET(recv_flags, TCP_FLG_FIN)) {
239             send_flags = TCP_FLG_ACK;
240             tcp_conn->state = TCP_STATE_LAST_ACK;
241             break;
242         }
243
244     case TCP_STATE_LAST_ACK:
245         // TODO: 仅在收到确认报文时 (ACK报文) 才做出处理，否则直接返回
246         if (!TCP_FLG_ISSET(recv_flags, TCP_FLG_ACK)) {
247             return;
248         }
249         // TODO: 关闭 TCP 连接
250         tcp_close_connection(remote_ip, remote_port, host_port);
251         break;
252
253     default:
254         printf(format: "do not support state %d\n", tcp_conn->state);
255         break;
256     }

```

连接状态是用一个状态机实现的。确认号 $Ack = \text{len}(\text{received pdu}) + (\text{SYN or FIN} ? 1 : 0)$ 。并且这里确认采用的是捎带确认的方式，指的是：不会发送一个 $\text{len}(\text{pdu})=0$ 但只有 ack 的数据包，这个确认号会连同下一次发包一起发送出去，但也有例外：收到 SYN/FIN 包。

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。)

服务器调用 `tcp_open`, 然后在 `tcp_handler_table` 建立一个 port 到 handler 之间的映射。然后轮询网卡, 看是否有收到新的消息, 然后就会进入本报告最上面展示的调用关系图。如果是一个 `tcp`, 并且有用户数据, 那么就会查找 `tcp_handler_table`, 找到监听此 port 的 `handler`, 并交给对应的 `handler` 处理, 对于 `web_server` 来说, `handler` 就是 `http_request_handler`。我们这里的 `web_server`, 只支持 `GET` 方法, 对于一个 `GET` 方法, 我们就会 `http_repond`, 我们会打开 `url` 中对应的文件, 并按照 `http` 协议的规定, 封装数据报, 然后将打开文件的内容写入 `http` 报文中, 最后调用 `tcp_out` 响应。对于没有资源相对应的文件, 我们就会返回 `404`

```

152 int main(int argc, char const *argv[]) {
153     printf(format: "Starting web server...\n");
154
155     if (net_init() == -1) { // 初始化协议栈
156         printf(format: "net init failed.\n");
157         return -1;
158     }
159     printf(format: "Network stack initialized successfully.\n");
160
161     printf(format: "Opening TCP port %d...\n", HTTP_LISTEN_PORT);
162     if (tcp_open(port: HTTP_LISTEN_PORT, handler: http_request_handler) == -1) {
163         printf(format: "Failed to open TCP port %d.\n", HTTP_LISTEN_PORT);
164         return -1;
165     }
166     printf(format: "TCP port %d opened successfully.\n", HTTP_LISTEN_PORT);
167
168     printf(format: "Web server is now running...\n");
169     while (1) {
170         net_poll(); // 一次主循环
171     }
172
173     return 0;
174 }
```

(启动 `web_server`)

```

91     size_t offset = 0;
92
93     const char *content_type = http_get_mime_type(file_path);
94     fseek(stream: file, off: 0, whence: SEEK_END);
95     size_t content_length = ftell(stream: file);
96     fseek(stream: file, off: 0, whence: SEEK_SET);
97
98 #define CRLF "\r\n"
99
100    /* Step2 : 发送 HTTP 请求头 */
101    // TODO: 发送 HTTP 状态行
102    offset += sprintf(s: resp_buffer + offset, format: "HTTP/1.1 200 OK" CRLF);
103    offset += sprintf(s: resp_buffer + offset, format: "Connection: Keep-Alive" CRLF);
104    offset += sprintf(s: resp_buffer + offset, format: "Content-Type: %s" CRLF, content_type);
105    offset += sprintf(s: resp_buffer + offset, format: "Content-Length: %zu" CRLF, content_length);
106    offset += sprintf(s: resp_buffer + offset, format: CRLF);
107    tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: offset, src_port: port, dst_ip, dst_port);
108
109    // 4. 打印调试信息, 用十六进制显示确保换行符正确
110    printf(format: "==== HTTP Headers (hex) ====\n");
111    for (size_t i = 0; i < offset; i++) {
112        printf(format: "%02x ", (unsigned char)resp_buffer[i]);
113        if ((i + 1) % 16 == 0)
114            printf(format: "\n");
115    }
116    printf(format: "\n==== HTTP Headers (text) ====\n%s", resp_buffer);
117    printf(format: "http_repond length: %zu\n", offset);
118
119    /* Step3 : 发送 HTTP 响应体 */
120    size_t bytes_read;
121    while ((bytes_read = fread(ptr: resp_buffer, size: 1, n: sizeof(resp_buffer), stream: file)) > 0) {
122        tcp_send(tcp_conn, data: (uint8_t *)resp_buffer, len: bytes_read, src_port: port, dst_ip, dst_port);
123    }
124

```

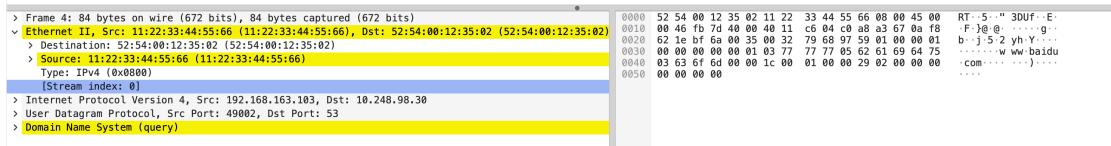
(访问资源, 组成 tcp 报文)

二、实验结果截图及分析

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

1. Eth 协议实验结果及分析

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



我分析的是 testing/eth_out 对应的 pcap.out，我注意到 testing/eth_out 的行为是：读取 pcap.in，然后将 frame 的首部清空，然后传给 ethernet_out，目的 mac 地址、上层协议保持不变

```
int i = 1;
PRINT_INFO("Feeding input %02d", i);
while ((ret = driver_recv(buf: &buf)) > 0) {
    printf(format: "\b\b%02d", i);
    fprintf(stream: control_flow, format: "\nRound %02d -----\n", i++);
    buf_copy(pdst: &buf2, psrc: &buf, len: 0);
    memset(s: buf.data, c: 0, n: sizeof(ether_hdr_t));
    buf_remove_header(buf: &buf, len: sizeof(ether_hdr_t));
    int proto = buf2.data[12];
    proto <= 8;
    proto |= buf2.data[13];
    ethernet_out(buf: &buf, mac: buf2.data, protocol: proto);
}
```

这是一条 dns 查询，dns（应用层）-> udp（传输层）-> ip（网络层）-> ethernet（链路层）这里的 ethernet 的源 mac 地址是 11:22:33:44:55:66 符合预期。目的地址、上层协议类型与输入保持不变，符合预期。

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。)

这个数据包对应的 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

```
> 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: 1a:94:f0:3c:49:aa (1a:94:f0:3c:49:aa)
    > Destination: 1a:94:f0:3c:49:aa (1a:94:f0:3c:49:aa)
    > Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
      Type: ARP (0x0806)
      Stream index: 2
      Padding: 
    <--> Address Resolution Protocol (reply)
      Hardware type: Ethernet (1)
      Protocol type: IPv4 (0x0800)
      Hardware size: 6
      Protocol size: 4
      Opcode: reply (2)
      Sender MAC address: 11:22:33:44:55:66 (11:22:33:44:55:66)
      Sender IP address: 192.168.163.103
      Target MAC address: 1a:94:f0:3c:49:aa (1a:94:f0:3c:49:aa)
      Target IP address: 192.168.163.2
```

0000	1a	94	f0	3c	49	aa	11	22	33	44	55	66	08	06	01	...<1...	3DUF	g	
0010	08	00	06	04	00	02	11	22	33	44	55	66	E0	a8	a3	67	...<1...	3DUF	g
0020	1a	94	f0	3c	49	aa	00	a8	a3	02	00	00	00	00	00	00	...<1...	3DUF	g
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	...<1...	3DUF	g

这个是 arp reply，arp 响应中的目的 mac 地址是 1a:94:f0:3c:49:aa

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。)

我将一段长文本使用 udp 进行发送。Udp 数据部分长度 5040，udp 首部长度 8，因此 ip 的 payload 长度是 5048

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.163.103	Broadcast	ARP	60	ARP Announcement for 192.168.163.103
2	0.000000	192.168.163.103	198.103.48.156	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=4567) [Reassembled in #5]
3	0.000000	192.168.163.103	198.103.48.156	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=4567) [Reassembled in #5]
4	0.000000	192.168.163.103	198.103.48.156	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=4567) [Reassembled in #5]
5	0.000000	192.168.163.103	198.103.48.156	UDP	642	642 36895 - 36895 [BAD UDP LENGTH 5056 > IP PAYLOAD LENGTH] Len=5048

5048 的数据，被分成了 4 个部分 $5480 = 1480 + 1480 + 1480 + 608$ ，可以看到，这里有正确的分片。

> Frame 2: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
> Destination: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
> Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
Type: IPv4 (0x0800)
[Stream index: 1]
Internet Protocol Version 4, Src: 192.168.163.103, Dst: 190.103.48.156
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
> Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 1500
Identification: 0x4567 (17767)
> 001. = Flags: 0x1, More fragments
...0 0000 0000 0000 = Fragment Offset: 0
Time to Live: 64
Protocol: UDP (17)
Header Checksum: 0xbc96 [validation disabled]
[Header checksum status: Unverified]
Source Address: 192.168.163.103
Destination Address: 190.103.48.156
[Reassembled IPv4 in frame: 5]
[Stream index: 0]
Data (1480 bytes)
Data [...] : 901f901f13c007fa416c6963652077617320626567696e6e696e6720746f20676574207665727920746972
[Length: 1480]

#1 MF 置位，offset = 0。版本号为 ipv4，首部长度为 20，总长度为 1500，标识为 17767（随机生成），TTL=64（顶格），协议字段为 17，因为上层是 udp。各字段均符合预期

```

> Frame 3: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
< Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
  > Destination: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
  > Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
    Type: IPv4 (0x0800)
    [Stream index: 1]
< Internet Protocol Version 4, Src: 192.168.163.103, Dst: 190.103.48.156
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 1500
    Identification: 0x4567 (17767)
  > 001. .... = Flags: 0x1, More fragments
    ...0 0000 1011 1001 = Fragment Offset: 1480
    Time to Live: 64
    Protocol: UDP (17)
    Header Checksum: 0xbdd [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.163.103
    Destination Address: 190.103.48.156
    [Reassembled IPv4 in frame: 5]
    [Stream index: 0]
< Data (1480 bytes)
  Data [...]: 207374726169676874206f6e206c696b6520612074756e6e656c20666f7220736f6d65207761792c20616e
  [Length: 1480]

```

#2 MF 被置位, offset 为 1480, 标识为 17767

```

> Frame 4: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
< Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
  > Destination: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
  > Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
    Type: IPv4 (0x0800)
    [Stream index: 1]
< Internet Protocol Version 4, Src: 192.168.163.103, Dst: 190.103.48.156
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 1500
    Identification: 0x4567 (17767)
  > 001. .... = Flags: 0x1, More fragments
    ...0 0001 0111 0010 = Fragment Offset: 2960
    Time to Live: 64
    Protocol: UDP (17)
    Header Checksum: 0xbb24 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.163.103
    Destination Address: 190.103.48.156
    [Reassembled IPv4 in frame: 5]
    [Stream index: 0]
< Data (1480 bytes)
  Data [...]: 6f727420696e20686572206c6573736f6e7320696e20746865207363686f6f6c726f6f6d2c20616e642074
  [Length: 1480]

```

#3 MF 被置位, offset 为 $2960 = 1480 + 1480$, 标识为 17767

```

> Frame 5: 642 bytes on wire (5136 bits), 642 bytes captured (5136 bits)
> Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
  Internet Protocol Version 4, Src: 192.168.163.103, Dst: 190.103.48.156
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCH: CS0, ECN: Not-ECT)
    Total Length: 628
    Identification: 0x4567 (17767)
  > 000. .... = Flags: 0x0
    ...0 0010 0010 1011 = Fragment Offset: 4440
    Time to Live: 64
    Protocol: UDP (17)
    Header Checksum: 0xdddd [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.163.103
    Destination Address: 190.103.48.156
  > [4 IPv4 Fragments (5048 bytes): #2(1480), #3(1480), #4(1480), #5(608)]
    [Stream index: 0]
  < User Datagram Protocol, Src Port: 36895, Dst Port: 36895
    Source Port: 36895
    Destination Port: 36895
    > Length: 5056 (bogus, payload length 5048)
      Checksum: 0x07fa [unverified]
      [Checksum Status: Unverified]
      [Stream index: 0]
      [Stream Packet Number: 1]
      [Timestamps]
      UDP payload (5040 bytes)
    < Data (5040 bytes)
      Data [...] 416c6963652077617320626567696e6e696e6720746f206765742076657279207469726564206f66207369
      [Length: 5040]

```

MF=0, offset 为 $4440 = 1480 + 1480 + 1480$, 标识为 17767

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。)

Type=0, echo reply, code=0, checksum 无误, 携带一段 ascii 从 0x10 到 0x37 的信息。均符合预期

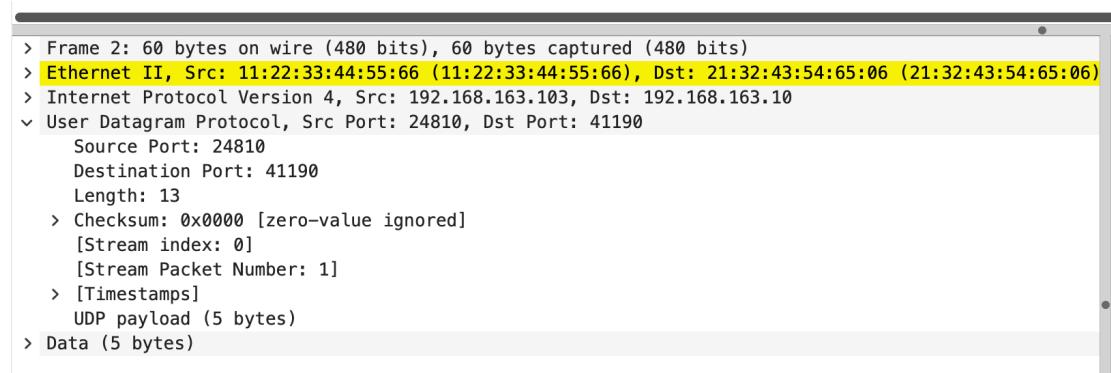
```
> Frame 7: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
> Ethernet II, Src: Pixim_34:45:56 (01:12:23:34:45:56), Dst: Pixim_34:45:56 (01:12:23:34:45:56)
> Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.110
< Internet Control Message Protocol
  Type: 0 (Echo (ping) 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)
  Data: c8e4865f00000000ae7c00000000000101112131415161718191a1b1c1d1e1f202122232425262728292a...
  [Length: 56]
```

Type=3, destination unreachable, code=2, checksum 无误, 携带一段全 0 信息。均符合预期。

```
> 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: 21:32:43:54:65:06 (21:32:43:54:65:06)
> 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: 0xce97 [correct]
    [Checksum Status: Good]
    Unused: 00000000
< Internet Protocol, bogus version (2)
  < 0010 ... - Version: 2
    > [Expert Info (Error/Protocol): Bogus IP version]
```

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。)



The screenshot shows a network traffic analysis tool interface. A single UDP frame is selected. The details pane displays the following information:

- > Frame 2: 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: 21:32:43:54:65:06 (21:32:43:54:65:06)
- > Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.10
- User Datagram Protocol, Src Port: 24810, Dst Port: 41190
 - Source Port: 24810
 - Destination Port: 41190
 - Length: 13
 - Checksum: 0x0000 [zero-value ignored]
 - [Stream index: 0]
 - [Stream Packet Number: 1]
 - [Timestamps]
 - UDP payload (5 bytes)
- > Data (5 bytes)

源端口号是 24810, 目的端口号是 41190, 长度为 13, checksum 忽略, 符合预期。

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。)
源端口号、目的端口号比较 trivial, 图上也有。下面仅分析一些关键步骤。

1) 三次握手

- Client(curl) -> server(web_server) SYN, seq=0(relative), server 上一个状态是 listen, client 下一个状态是 syn-sent

```

> Frame 7: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
> Ethernet II, Src: be:67:30:9c:b6:46 (be:67:30:9c:b6:46), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55)
> Internet Protocol Version 4, Src: 198.19.249.147, Dst: 198.19.249.148
> Transmission Control Protocol, Src Port: 46370, Dst Port: 80, Seq: 0, Len: 0
    Source Port: 46370
    Destination Port: 80
    [Stream index: 0]
    [Stream Packet Number: 1]
    > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0      (relative sequence number)
    Sequence Number (raw): 2068663036
    [Next Sequence Number: 1      (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
    > Flags: 0x002 (SYN)
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0x7f7e [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    > Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
    > [Timestamps]

```

- Server -> client SYN, ACK, seq=0(relative), ack=1, client 上一个状态是 syn-sent, server 下一个状态是 syn-rcvd

```

> Frame 8: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: CIMSYS_33:44:55 (00:11:22:33:44:55), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
> Internet Protocol Version 4, Src: 198.19.249.148, Dst: 198.19.249.147
> Transmission Control Protocol, Src Port: 80, Dst Port: 46370, Seq: 0, Ack: 1, Len: 0
    Source Port: 80
    Destination Port: 46370
    [Stream index: 0]
    [Stream Packet Number: 2]
    > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0      (relative sequence number)
    Sequence Number (raw): 1473520563
    [Next Sequence Number: 1      (relative sequence number)]
    Acknowledgment Number: 1      (relative ack number)
    Acknowledgment number (raw): 2068663037
    0101 .... = Header Length: 20 bytes (5)
    > Flags: 0x012 (SYN, ACK)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0x393e [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    > [Timestamps]
    > [SEQ/ACK analysis]

```

- Client -> server ACK, seq=1, ack=1, server 上一个状态是 syn-rcvd, client 下一个状态是 established。Server 下一个状态是 established

```

> Frame 10: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
> Ethernet II, Src: be:67:30:9c:b6:46 (be:67:30:9c:b6:46), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55)
> Internet Protocol Version 4, Src: 198.19.249.147, Dst: 198.19.249.148
< Transmission Control Protocol, Src Port: 46370, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
    Source Port: 46370
    Destination Port: 80
    [Stream index: 0]
    [Stream Packet Number: 4]
    > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1      (relative sequence number)
    Sequence Number (raw): 2068663037
    [Next Sequence Number: 1      (relative sequence number)]
    Acknowledgment Number: 1      (relative ack number)
    Acknowledgment number (raw): 1473520564
    0101 .... = Header Length: 20 bytes (5)
    > Flags: 0x010 (ACK)
    Window: 64240
    [Calculated window size: 8222720]
    [Window size scaling factor: 128]
    Checksum: 0x7f6a [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    > [Timestamps]
    > [SEQ/ACK analysis]

```

2) Established

- Client -> server 发起 http 的 get 方法, seq=1, ack=1

```

> Frame 12: 132 bytes on wire (1056 bits), 132 bytes captured (1056 bits)
> Ethernet II, Src: be:67:30:9c:b6:46 (be:67:30:9c:b6:46), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55)
> Internet Protocol Version 4, Src: 198.19.249.147, Dst: 198.19.249.148
< Transmission Control Protocol, Src Port: 46370, Dst Port: 80, Seq: 1, Ack: 1, Len: 78
    Source Port: 46370
    Destination Port: 80
    [Stream index: 0]
    [Stream Packet Number: 6]
    > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 78]
    Sequence Number: 1      (relative sequence number)
    Sequence Number (raw): 2068663037
    [Next Sequence Number: 79      (relative sequence number)]
    Acknowledgment Number: 1      (relative ack number)
    Acknowledgment number (raw): 1473520564
    0101 .... = Header Length: 20 bytes (5)
    > Flags: 0x018 (PSH, ACK)
    Window: 64240
    [Calculated window size: 8222720]
    [Window size scaling factor: 128]
    Checksum: 0x7fb8 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    > [Timestamps]
    > [SEQ/ACK analysis]
    TCP payload (78 bytes)
    > Hypertext Transfer Protocol

```

- Server -> client http 响应, 采用捎带确认, seq=1, ack=79(1+78), 上一个 tcp 的 pdu 长度是 78

```

  20 16.186396 198.19.249.148 198.19.249.147 TCP 60 80 → 46370
> Frame 13: 158 bytes on wire (1264 bits), 158 bytes captured (1264 bits)
> Ethernet II, Src: CIMSYS_33:44:55 (00:11:22:33:44:55), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
> Internet Protocol Version 4, Src: 198.19.249.148, Dst: 198.19.249.147
< Transmission Control Protocol, Src Port: 80, Dst Port: 46370, Seq: 1, Ack: 79, Len: 104
  Source Port: 80
  Destination Port: 46370
  [Stream index: 0]
  [Stream Packet Number: 7]
> [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 104]
  Sequence Number: 1      (relative sequence number)
  Sequence Number (raw): 1473520564
  [Next Sequence Number: 105      (relative sequence number)]
  Acknowledgment Number: 79      (relative ack number)
  Acknowledgment number (raw): 2068663115
  0101 .... = Header Length: 20 bytes (5)
> Flags: 0x010 (ACK)
  Window: 65535
  [Calculated window size: 65535]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0xfc7b [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
> [Timestamps]
> [SEQ/ACK analysis]
  TCP payload (104 bytes)
  [Reassembled PDU in frame: 14]
  TCP segment data (104 bytes)

```

3) 四次挥手

1. client -> server, 第一次挥手 FIN, ACK, seq=79, ack=607, len=0, 服务器的上一个状态是 established, 客户端的下一个状态是 fin_wait

```

> Frame 19: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
> Ethernet II, Src: be:67:30:9c:b6:46 (be:67:30:9c:b6:46), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55)
> Internet Protocol Version 4, Src: 198.19.249.147, Dst: 198.19.249.148
< Transmission Control Protocol, Src Port: 33368, Dst Port: 80, Seq: 79, Ack: 607, Len: 0
  Source Port: 33368
  Destination Port: 80
  [Stream index: 0]
  [Stream Packet Number: 15]
> [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 0]
  Sequence Number: 79      (relative sequence number)
  Sequence Number (raw): 3891314429
  [Next Sequence Number: 80      (relative sequence number)]
  Acknowledgment Number: 607      (relative ack number)
  Acknowledgment number (raw): 1142686244
  0101 .... = Header Length: 20 bytes (5)
> Flags: 0x011 (FIN, ACK)
  Window: 63784
  [Calculated window size: 63784]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0x7f6a [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
> [Timestamps]

```

2. server -> client 第二次挥手与第三次挥手合并, FIN, ACK, ack=80, seq=607, len=0 服务器的下一个状态是 last_ack, 客户端的上一个状态是 fin_wait_1 + fin_wait_2

```

> Frame 20: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: CIMSYS_33:44:55 (00:11:22:33:44:55), Dst: be:67:30:9c:b6:46 (be:67:30:9c:b6:46)
> Internet Protocol Version 4, Src: 198.19.249.148, Dst: 198.19.249.147
< Transmission Control Protocol, Src Port: 80, Dst Port: 33368, Seq: 607, Ack: 80, Len: 0
  Source Port: 80
  Destination Port: 33368
  [Stream index: 0]
  [Stream Packet Number: 16]
  > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 607      (relative sequence number)
    Sequence Number (raw): 1142686244
    [Next Sequence Number: 608      (relative sequence number)]
    Acknowledgment Number: 80      (relative ack number)
    Acknowledgment number (raw): 3891314430
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x011 (FIN, ACK)
  Window: 65535
  [Calculated window size: 65535]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0xc0ab [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]

```

3. client -> server 第四次挥手，ACK, seq=80, ack=608 客户端的下一个状态是 time_wait，服务端的上一个状态是 last_ack，服务端的下一个状态是 closed。客户端等待两个 MSL 后进入 closed 状态

```

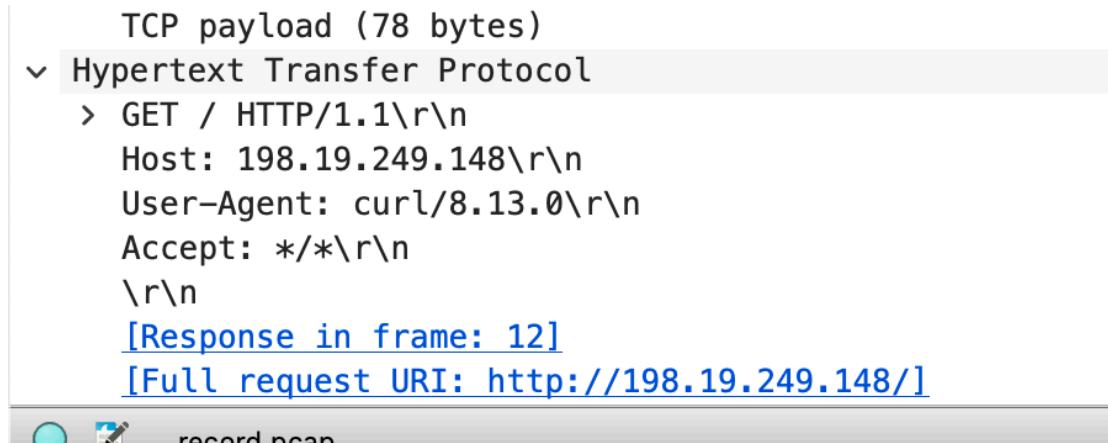
> Frame 22: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
> Ethernet II, Src: be:67:30:9c:b6:46 (be:67:30:9c:b6:46), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55)
> Internet Protocol Version 4, Src: 198.19.249.147, Dst: 198.19.249.148
< Transmission Control Protocol, Src Port: 33368, Dst Port: 80, Seq: 80, Ack: 608, Len: 0
  Source Port: 33368
  Destination Port: 80
  [Stream index: 0]
  [Stream Packet Number: 18]
  > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 80      (relative sequence number)
    Sequence Number (raw): 3891314430
    [Next Sequence Number: 80      (relative sequence number)]
    Acknowledgment Number: 608      (relative ack number)
    Acknowledgment number (raw): 1142686245
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x010 (ACK)
  Window: 63784
  [Calculated window size: 63784]
  [Window size scaling factor: -2 (no window scaling used)]
  Checksum: 0x7f6a [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]

```

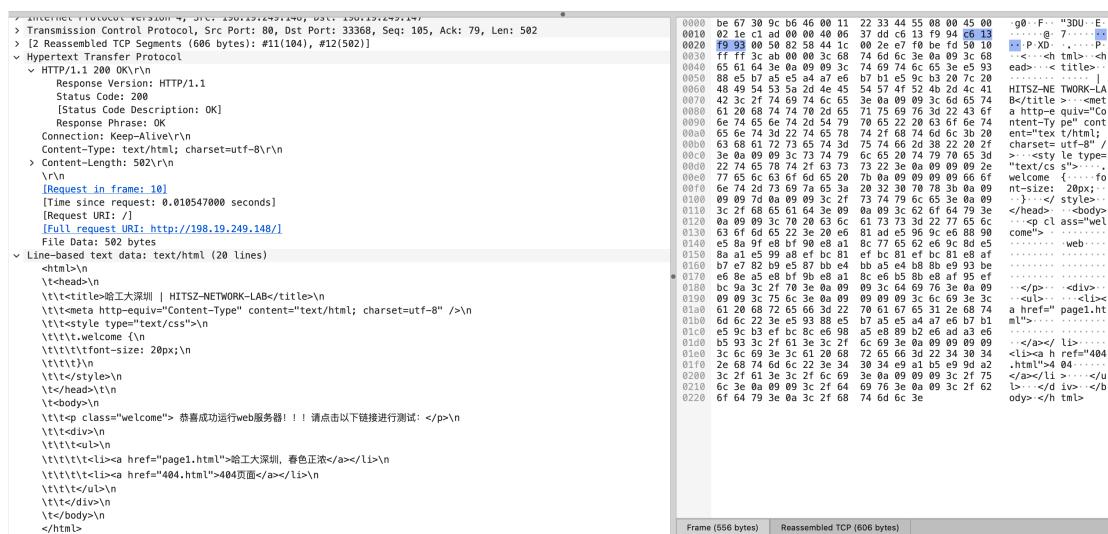
7. web 服务器实验结果及分析

(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。)

请求由客户端发起, 请求方法是 GET, 请求 url 是 <http://198.19.249.148/>, 请求头如下图



响应由服务端响应, 响应状态码是 200 OK, 响应体是 index.html 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。)

- 我遇到了严重的内存 bug，因为我笔误了，我通过链接-fsanitize=address 知道了内存 bug 的位置。

```
19
20 add_compile_options(-Wall -g -O0)
21 # add_link_options(-fsanitize=address)
22 # set(EXECUTABLE_OUTPUT_PATH ${CMAKE_CURRENT_SOURCE_DIR}/test)
23 include_directories(./include ./Npcap/Include)
24 link_directories(../Npcap/Lib ./Npcap/Lib/x64)
25 aux_source_directory(.src DIR_SRCS)
26
```

- Gdb 并不好打断点（因为有时序问题），我通过插入内联汇编，手动触发调试中断成功打上断点，正确的观察到了程序的行为

```
119 * @param dst_port 目标端口号
120 * @param flags TCP 标志位
121 */
122 void tcp_out(tcp_conn_t *tcp_conn, buf_t *buf, uint16_t src_port, uint8_t *dst_ip, uint16_t dst_port, uint8_t flags) {
123     /* ===== TODO 1 BEGIN ===== */
124     if (buf->len != 0) {
125         __asm__ volatile("brk #0"); // aarch64
126     }
127     uint32_t payload_len = buf->len;
128     buf_add_header(buf, len: sizeof(tcp_hdr_t));
129     tcp_hdr_t *tcp_hdr = (tcp_hdr_t *)buf->data;
130     tcp_hdr->src_port16 = htons(hostshort: src_port);
131     tcp_hdr->dst_port16 = htons(hostshort: dst_port);
132     tcp_hdr->seq = htonl(hostlong: tcp_conn->seq);
133     tcp_hdr->ack = htonl(hostlong: tcp_conn->ack);
134     tcp_hdr->doff = sizeof(tcp_hdr_t) >> 2;
135     tcp_hdr->flags = flags | TCP_FLG_ACK; // as server, always send ACK
136     tcp_hdr->win = htons(hostshort: TCP_MAX_WINDOW_SIZE);
137     tcp_hdr->checksum16 = 0;
138     tcp_hdr->uptr = 0;
139     tcp_hdr->checksum16 = transport_checksum(protocol: NET_PROTOCOL_TCP, buf, src_ip: net_if_ip, dst_ip);
140     ip_out(buf, ip: dst_ip, protocol: NET_PROTOCOL_TCP);
141     /* ===== TODO 1 END ===== */
142     uint32_t next_seq = tcp_conn->seq + payload_len;
143     tcp_conn->seq = tcp_conn->remote_ack < next_seq ? tcp_conn->remote_ack : next_seq;
144 }
```

- 为了方便观察，我在 driver_send, driver_recv 处加入了能 dump 数据包的代码，相当于是做了一个内置的抓包。

```

166 /**
167 * @brief 尝试从网卡接收数据包
168 *
169 * @param buf 收到的数据包
170 * @return int 数据包的长度, 未收到为0, 错误为-1
171 */
172 int driver_recv(buf_t *buf) {
173     struct pcap_pkthdr *pkt_hdr;
174     const uint8_t *pkt_data;
175     int ret = pcap_next_ex(pcap, &pkt_hdr, &pkt_data); // read the next packet
176     if (ret == 0)
177         return 0;
178     else if (ret == 1) {
179         memcpy(dest: buf->data, src: pkt_data, n: pkt_hdr->len);
180         buf->len = pkt_hdr->len;
181         if (dump_file != NULL) {
182             pcap_dump((u_char *)dump_file, pkt_hdr, pkt_data);
183             pcap_dump_flush(dump_file);
184         }
185         return pkt_hdr->len;
186     }
187     fprintf(stream: stderr, format: "Error in driver_recv.\n%s.\n", pcap_geterr(pcap));
188     return -1;
189 }
190 /**
191 * @brief 使用网卡发送一个数据包
192 *
193 * @param buf 要发送的数据包
194 * @return int 成功为0, 失败为-1
195 */
196 int driver_send(buf_t *buf) {
197     if (pcap_sendpacket(pcap, buf->data, buf->len) == -1) {
198         fprintf(stream: stderr, format: "Error in driver_send.\n%s.\n", pcap_geterr(pcap));
199         return -1;
200     }
201     if (dump_file != NULL) {
202         struct pcap_pkthdr hdr;
203         gettimeofday(tv: &hdr.ts, tz: NULL);
204         hdr.caplen = buf->len;
205         hdr.len = buf->len;
206         pcap_dump((u_char *)dump_file, &hdr, buf->data);
207         pcap_dump_flush(dump_file);
208     }
209     return 0;
210 }

```

4. Web_server 的权限问题

```

47
48 # Add post-build command to set capabilities for web_server
49 add_custom_command(TARGET web_server POST_BUILD
50 COMMAND sudo setcap cap_net_raw,cap_net_admin,cap_net_bind_service,cap_sys_ptrace=eip $<TARGET_FILE:web_server>
51 COMMENT "Setting capabilities for web_server"
52 )
53

```

我在 cmakelists 中加了一条语句, 这样我在普通用户(非 root)下就能运行 web_server 了

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。)

实验指导书写的很详细，照着实验指导书做就行了。协议栈和网络配置。
如果希望学生写附加题，那么建议补充更多的，合适的测试用例。
网络配置部分，其实可以加入华为的 eNSP 模拟器，一方面节约成本，另一方面可能配置更方便？