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

实验报告

开课学期：2025 秋季

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：

学生学号：

学生姓名：

评阅教师：

报告成绩：

实验与创新实践教育中心制

2025 年 10 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。）

在封装过程中，ethernet_out 函数首先检查待发送数据包的长度，若小于以太网最小传输单元（46 字节），则通过 buf_add_padding 显式填充 0 以补齐长度；随后在数据包前端添加以太网头部空间，并将目标 MAC 地址、本机源 MAC 地址以及经过字节序转换的上层协议类型依次写入头部对应字段，最后调用 driver_send 将构建好的数据帧发送给驱动层。

在解封装过程中，ethernet_in 函数首先校验接收到的数据包长度，若小于以太网头部大小则视为不完整包并直接返回；若校验通过，则使用 buf_remove_header 移除以太网头部以剥离出上层数据，同时提取头部中的源 MAC 地址和协议类型（进行字节序转换），最终调用 net_in 函数将处理后的数据包及解析出的协议信息传递给上层网络协议栈。

```c
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 */
void ethernet_in(buf_t *buf) {
    ether_hdr_t *eth_hdr = (ether_hdr_t *)buf->data;
    if (buf->len < sizeof(ether_hdr_t)) return; // 如果数据长度小于以太网头部长度，则认为数据包不完整
    buf_remove_header(buf, sizeof(ether_hdr_t)); // 移除以太网包头
    net_in(buf, swap16(eth_hdr->protocol16), eth_hdr->src); // 调用net_in()函数向上层传递数据包
}
/**
 * @brief 处理一个要发送的数据包
 *
 * @param buf 要处理的数据包
 * @param mac 目标MAC地址
 * @param protocol 上层协议
 */
void ethernet_out(buf_t *buf, const uint8_t *mac, net_protocol_t protocol) {
    // 数据长度不足46则显式填充0
    if (buf->len < ETHERNET_MIN_TRANSPORT_UNIT) {
        buf_add_padding(buf, ETHERNET_MIN_TRANSPORT_UNIT - buf->len);
    }

    buf_add_header(buf, sizeof(ether_hdr_t));  // 添加以太网包头
    ether_hdr_t *hdr = (ether_hdr_t *)buf->data;
    memcpy(hdr->dst, mac, NET_MAC_LEN);  // 填写目的MAC地址
    memcpy(hdr->src, net_if_mac, NET_MAC_LEN);  // 填写源MAC地址
    hdr->protocol16 = swap16(protocol);  // 填写协议类型
    // 将添加了以太网包头的数据帧发送到驱动层
    driver_send(buf);
}
```

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。）

在数据发送流程中，arp_out 函数首先查询 ARP 缓存表，若目标 IP 对应的 MAC 地址存在，则直接调用以太网层接口发送数据；若表中缺失该表项，系统会进一步检查 arp_buf 是否存在针对该 IP 的挂起任务，若无挂起任务则将当前数据包缓存，并调用 arp_req 构造 ARP 请求报文，将其操作码置为 RP_REQUEST（需进行字节序转换），目标 MAC 地址设为广播地址（FF:FF:FF:FF:FF:FF），通过广播形式询问目标 IP 的物理地址。

```c
/**
 * @brief 处理一个要发送的数据包
 *
 * @param buf 要处理的数据包
 * @param ip 目标ip地址
 */
void arp_out(buf_t *buf, uint8_t *ip) {
    // 调用map_get()函数，根据IP地址来查找ARP表
    uint8_t *target_mac = (uint8_t *) map_get(&arp_table, ip);

    // 如果能找到该IP地址对应的MAC地址，则将数据包直接发送给以太网层
    if (target_mac != NULL) {
        ethernet_out(buf, target_mac, NET_PROTOCOL_IP);
        return;
    }

    // 否则，进一步判断arp_buf是否已经有包；有则说明正在等待该ip回应ARP请求，此时不能再发送arp请求，没有则将数据包缓存到arp_buf，然后调用arp_req()请求MAC地址
    if (map_get(&arp_buf, ip) == NULL) {
        map_set(&arp_buf, ip, buf);
        arp_req(ip);
    }
}
```

```c
/**
 * @brief 发送一个arp请求
 *
 * @param target_ip 想要知道的目标的ip地址
 */
void arp_req(uint8_t *target_ip) {
    // 调用buf_init()对txbuf进行初始化
    buf_init(&txbuf, sizeof(arp_pkt_t));

    // 填写ARP报头
    arp_pkt_t __arp_pkt = arp_init_pkt;
    // ARP操作类型为ARP_REQUEST，注意大小端转换
    __arp_pkt.opcode16 = swap16(ARP_REQUEST);

    memcpy(__arp_pkt.target_ip, target_ip, NET_IP_LEN);
    memcpy(txbuf.data, &__arp_pkt, sizeof(arp_pkt_t));

    // 广播地址
    uint8_t broadcast_mac[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

    // 调用ethernet_out函数将ARP报文发送出去
    ethernet_out(&txbuf, broadcast_mac, NET_PROTOCOL_ARP);
}
```

在数据接收与处理流程中，arp_in 函数首先校验接收报文的硬件类型、协议类型、地址长度及操作码的合法性。一旦通过校验，立即调用 map_set 更新 ARP 表，将发送方的 IP 与 MAC 地址写入 arp_table。随后，系统检查 arp_buf 中是否有等待该 IP 解析结果的缓存数据包，若有则立即通过 ethernet_out 将缓存的数据包发送出去并删除缓存。若没有则判断该报文是否为针对本机 IP 的请求（opcode 为 ARP_REQUEST 且 target_ip 匹配本机 IP），若是，则调用 arp_resp 函数封装 ARP 响应报文，将操作码置为 ARP_REPLY，填入本机 MAC 作为发送端地址，并将原请求的发送方 MAC 作为目的地址，以单播形式回复完成地址解析握手。

```c
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 * @param src_mac 源mac地址
 */
void arp_in(buf_t *buf, uint8_t *src_mac) {
    arp_pkt_t *arp_pkt = (arp_pkt_t *)buf->data;

    // 判断数据长度，并查看报文是否完整
    if (buf->len < sizeof(arp_pkt_t) ||
        arp_pkt->hw_type16 != swap16(ARP_HW_ETHER) ||
        arp_pkt->pro_type16 != swap16(NET_PROTOCOL_IP) ||
        arp_pkt->hw_len != NET_MAC_LEN ||
        arp_pkt->pro_len != NET_IP_LEN ||
        (arp_pkt->opcode16 != swap16(ARP_REQUEST) &&
        arp_pkt->opcode16 != swap16(ARP_REPLY))
    )   return;

    // 调用map_set()函数更新ARP表项
    map_set(&arp_table, arp_pkt->sender_ip, src_mac);

    // 调用map_get()函数查看该接收报文的IP地址是否有对应的arp_buf缓存
    buf_t *__arp_buf = (buf_t *)map_get(&arp_buf, arp_pkt->sender_ip);

    // 如果有缓存，处理该待发送的数据包
    if (__arp_buf != NULL) {
        ethernet_out(__arp_buf, src_mac, NET_PROTOCOL_IP);
        map_delete(&arp_buf, arp_pkt->sender_ip);
        return;
    }

    // 否则，进一步判断该数据包是否是请求本主机MAC地址的ARP请求报文
    if (arp_pkt->opcode16 == swap16(ARP_REQUEST) && memcmp(arp_pkt->target_ip, net_if_ip, NET_IP_LEN) == 0) {
        arp_resp(arp_pkt->sender_ip, arp_pkt->sender_mac);
    }
}
```

```c
/**
 * @brief 发送一个arp响应
 *
 * @param target_ip 目标ip地址
 * @param target_mac 目标mac地址
 */
void arp_resp(uint8_t *target_ip, uint8_t *target_mac) {
    // 调用buf_init()对txbuf进行初始化
    buf_init(&txbuf, sizeof(arp_pkt_t));

    // 填写ARP报头首部
    arp_pkt_t __arp_pkt = arp_init_pkt;
    __arp_pkt.opcode16 = swap16(ARP_REPLY);

    memcpy(__arp_pkt.target_ip, target_ip, NET_IP_LEN);
    memcpy(__arp_pkt.target_mac, target_mac, NET_MAC_LEN);
    memcpy(txbuf.data, &__arp_pkt, sizeof(__arp_pkt));

    // 调用ethernet_out函数将ARP报文发送出去
    ethernet_out(&txbuf, target_mac, NET_PROTOCOL_ARP);
}
```

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。）

在 IP 数据报输出处理中，ip_out 函数首先检查上层传递的数据长度是否超过了以太网最大传输单元（MTU）减去 IP 首部后的限制（即 1480 字节），若数据过大，则依据 MTU 限制将数据切割成多个分片，为每个分片计算精确的片偏移量，并对除最后一个分片外的所有分片设置 MF 标志位，随后调用 ip_fragment_out 函数为每个分片或未分片的完整数据包

添加 IP 头部；头部封装过程中，程序会在缓冲区前部预留 20 字节空间，依次填充版本号（IPv4）、首部长度、服务类型、总长度、标识符、TTL、上层协议类型以及源、目的 IP 地址，并调用 checksum16 函数——该函数采用标准的 16 位反码求和算法——计算首部校验和以确保头部数据的完整性，最后通过 arp_out 将封装好的 IP 数据报发送至链路层。

```
97  /**
98   * @brief 处理一个要发送的ip数据包
99   *
100  * @param buf 要处理的包
101  * @param ip 目标ip地址
102  * @param protocol 上层协议
103  */
104 void ip_out(buf_t *buf, uint8_t *ip, net_protocol_t protocol) {
105     static uint16_t ip_id = 0;
106 
107     // 若无需分段，则直接发送
108     if (buf->len <= ETHERNET_MAX_TRANSPORT_UNIT - 20) {
109         ip_fragment_out(buf, ip, protocol, ip_id++, 0, 0);
110         return;
111     }
112 
113     uint16_t cur = 0;
114 
115     // 分段发送，每段长度为1480B
116     buf_t fragment;
117     // buf_t *fragment = &__fragment;
118     while (buf->len > 1480) {
119         buf_init(&fragment, 1480);
120         memcpy((&fragment)->data, buf->data, 1480);
121         buf_remove_header(buf, 1480);
122         ip_fragment_out(&fragment, ip, protocol, ip_id, cur/IP_HDR_OFFSET_PER_BYTE, 1);
123         cur += 1480;
124     }
125 
126     // 发送最后一个分片
127     if (buf->len > 0) {
128         buf_init(&fragment, buf->len);
129         memcpy((&fragment)->data, buf->data, buf->len);
130         buf_remove_header(buf, buf->len);
131         ip_fragment_out(&fragment, ip, protocol, ip_id, cur/IP_HDR_OFFSET_PER_BYTE, 0);
132     }
133 
134     ip_id++;
135 }
```

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

    ip_hdr_t *ip_hdr = (ip_hdr_t *)buf->data;

    // 填写IP数据报头部字段
    ip_hdr->version = IP_VERSION_4;
    ip_hdr->hdr_len = 5;
    ip_hdr->tos = 0;
    ip_hdr->total_len16 = swap16(buf->len);
    ip_hdr->id16 = swap16(id);

    if (mf) ip_hdr->flags_fragment16 = swap16(IP_MORE_FRAGMENT | offset);
    else    ip_hdr->flags_fragment16 = swap16(offset);

    ip_hdr->ttl = IP_DEFALUT_TTL;
    ip_hdr->protocol = protocol;
    ip_hdr->hdr_checksum16 = 0;

    memcpy(ip_hdr->src_ip, net_if_ip, NET_IP_LEN);
    memcpy(ip_hdr->dst_ip, ip, NET_IP_LEN);

    // 计算校验和
    ip_hdr->hdr_checksum16 = checksum16((uint16_t*)ip_hdr, sizeof(ip_hdr_t));

    // 封装到buf中并发送
    arp_out(buf, ip);
}
```

```c
/**
 * @brief 计算16位校验和
 *
 * @param buf 要计算的数据包
 * @param len 要计算的长度
 * @return uint16_t 校验和
 */
uint16_t checksum16(uint16_t *data, size_t len) {
    uint32_t sum = 0;

    // 处理主体，循环处理类似IP分片的逻辑
    while (len > 1) {
        sum += *data++;
        len -= 2;
        while (sum >> 16)
            sum = (sum >> 16) + (sum & 0xffff);
    }

    // 处理剩余部分
    if (len)
        sum += *(uint8_t *)data;
    while (sum >> 16)
        sum = (sum >> 16) + (sum & 0xffff);

    sum =~ sum;
    return (uint16_t)sum;
}
```

在 IP 数据报输入处理中，ip_in 函数负责对流入的数据包进行严格的合法性验证，包括

检查数据包长度是否足以容纳 IP 头部、确认 IP 版本号、验证目的 IP 地址是否与本机匹配，以及重新计算校验和并与包头中的值比对以检测传输错误；一旦通过校验，函数将根据头部中的总长度字段去除尾部填充数据，剥离 IP 头部，并根据协议字段将有效载荷分发给 TCP、UDP 或 ICMP 等上层协议处理，若遇到不支持的协议类型，则触发 icmp_unreachable 函数回送 ICMP 协议不可达报文。

```c
void ip_in(buf_t *buf, uint8_t *src_mac) {
    ip_hdr_t *ip_hdr = (ip_hdr_t *)buf->data;

    // 检查数据包长度
    if (buf->len < sizeof(ip_hdr_t)) return;

    // 报头检测
    if (ip_hdr->version != IP_VERSION_4 || swap16(ip_hdr->total_len16) > buf->len || ip_hdr->hdr_len < 5)
        return;

    // 校验头部校验和
    uint16_t hdr_checksum = ip_hdr->hdr_checksum16;
    ip_hdr->hdr_checksum16 = 0;

    if (hdr_checksum != checksum16((uint16_t *)ip_hdr, ip_hdr->hdr_len * IP_HDR_LEN_PER_BYTE))
        return;

    ip_hdr->hdr_checksum16 = hdr_checksum;

    // 对比目标IP和本机IP
    if (memcmp(net_if_ip, ip_hdr->dst_ip, NET_IP_LEN) != 0)
        return;

    // 去除填充字段
    if (buf->len > swap16(ip_hdr->total_len16))
        buf_remove_padding(buf, buf->len - swap16(ip_hdr->total_len16));

    uint8_t src_ip[NET_IP_LEN];
    memcpy(src_ip, ip_hdr->src_ip, NET_IP_LEN);
    uint8_t protocol = ip_hdr->protocol;

    // 遇到不能识别的协议类型，返回ICMP不可达报文
    if (protocol != NET_PROTOCOL_ICMP && protocol != NET_PROTOCOL_UDP && protocol != NET_PROTOCOL_IP) {
        icmp_unreachable(buf, src_ip, ICMP_CODE_PROTOCOL_UNREACH);
        return;
    }

    buf_remove_header(buf, ip_hdr->hdr_len * IP_HDR_LEN_PER_BYTE);

    // 向上层传递数据包
    net_in(buf, protocol, src_ip);
}
```

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。）

处理 ICMP 请求：当网络层接收到协议字段为 ICMP 的数据包时，会将其剥离 IP 头后传递给 icmp_in。icmp_in 函数首先对数据包长度进行合法性检查（确保至少包含 ICMP 头部），然后解析报头。如果报文类型（Type）为 8 且代码（Code）为 0，即识别为 ICMP 回显请求，系统将调用 icmp_resp 函数进行响应处理。

```c
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 * @param src_ip 源ip地址
 */
void icmp_in(buf_t *buf, uint8_t *src_ip) {
    // 报头检测
    if (buf->len < sizeof(icmp_hdr_t)) return;
    icmp_hdr_t *icmp_hdr = (icmp_hdr_t *)buf->data;
    if (icmp_hdr->type == ICMP_TYPE_ECHO_REQUEST && icmp_hdr->code == 0)
        icmp_resp(buf, src_ip);
}
```

处理 ICMP 响应：icmp_resp 函数负责生成并发送 ICMP 回显应答。设计逻辑如下：首先初始化发送缓冲区 txbuf，并将接收到的请求报文内容完整拷贝到发送缓冲区中。这样做保留了原报文中的标识符、序列号以及可选数据部分，符合 Ping 操作的要求。然后，将 ICMP 头部的类型字段（Type）修改为 ICMP_TYPE_ECHO_REPLY（0），代码字段（Code）保持为 0，并将校验和字段置零以便重新计算。接着，调用 checksum16 函数，对缓冲区中的数据（包含头部和载荷）计算校验和并填回头部。最后，调用 ip_out 函数，将源 IP 地址作为目的 IP，封装 IP 头部并发送报文，完成一次 Ping 的应答过程。

```c
/**
 * @brief 发送icmp响应
 *
 * @param req_buf 收到的icmp请求包
 * @param src_ip 源ip地址
 */
static void icmp_resp(buf_t *req_buf, uint8_t *src_ip) {
    buf_init(&txbuf, req_buf->len);
    memcpy(txbuf.data, req_buf->data, req_buf->len);

    // 封装头部数据
    icmp_hdr_t *icmp_hdr = (icmp_hdr_t*)txbuf.data;
    icmp_hdr->type = ICMP_TYPE_ECHO_REPLY;
    icmp_hdr->code = 0;
    icmp_hdr->checksum16 = 0;

    // 计算校验和
    icmp_hdr->checksum16 = checksum16((uint16_t *)txbuf.data, txbuf.len);

    // 发送数据报
    ip_out(&txbuf, src_ip, NET_PROTOCOL_ICMP);
}
```

网络故障诊断：icmp_unreachable 函数用于生成目的地不可达报文，这是 ICMP 进行故障诊断的核心机制。当网络层发现不支持的上层协议，或传输层发现目标端口未打开时，会调用此函数。根据 RFC 标准，差错报文的数据部分必须包含原出错数据包的 IP 头部以及其载荷的前 8 个字节。函数首先初始化 txbuf，大小为 IP 头长度加上 8 字节，并将原数据包的相应部分拷贝进去。然后，使用 buf_add_header 在数据前方添加 ICMP 头部。设置类型字段（Type）为 ICMP_TYPE_UNREACH（3），并根据传入的参数设置具体的错误代码（Code），例如 ICMP_CODE_PROTOCOL_UNREACH（协议不可达）或 ICMP_CODE_PORT_UNREACH（端口不可达）。接着同样清零校验和，利用 checksum16 计算并填充校验和，最终通过 ip_out 将该差错报文发送回原数据包的发送者，通知其网络不可达的具体原因。

```c
/**
 * @brief 发送icmp不可达
 *
 * @param recv_buf 收到的ip数据包
 * @param src_ip 源ip地址
 * @param code icmp code，协议不可达或端口不可达
 */
void icmp_unreachable(buf_t *recv_buf, uint8_t *src_ip, icmp_code_t code) {
    // icmp报文 = icmp_hdr + ip_hdr + ip_data(8 bytes)
    buf_init(&txbuf, sizeof(ip_hdr_t) + 8);
    memcpy(txbuf.data, recv_buf->data, sizeof(ip_hdr_t) + 8);

    buf_add_header(&txbuf, sizeof(icmp_hdr_t));

    // 构造icmp头部
    icmp_hdr_t *icmp_hdr = (icmp_hdr_t*)txbuf.data;
    icmp_hdr->type = ICMP_TYPE_UNREACH;
    icmp_hdr->code = code;
    icmp_hdr->checksum16 = 0;
    icmp_hdr->id16 = 0;
    icmp_hdr->seq16 = 0;

    // 计算校验和
    icmp_hdr->checksum16 = checksum16((uint16_t *)txbuf.data, txbuf.len);

    // 发送数据报
    ip_out(&txbuf, src_ip, NET_PROTOCOL_ICMP);
}
```

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。）

UDP 封装流程由 udp_out 函数实现。首先，函数根据待发送数据的长度，调用 buf_add_header 在数据载荷前预留 UDP 首部空间。接着，初始化 UDP 首部结构体，依次填充源端口号、目的端口号，并将数据报总长度写入 total_len16 字段。在计算校验和之前，先将 checksum16 字段置为 0。随后调用 transport_checksum 函数计算校验和。设计中特别处理了校验和的特殊情况：若计算结果为 0x0000，根据 RFC 768 标准，必须将其转换为 0xFFFF 写入首部，以区分“校验和为 0”与“未使用校验和”的情况。最后，调用 ip_out 函数，指定协议类型为 NET_PROTOCOL_UDP，将封装好的 UDP 数据报发送至目的 IP 地址。

UDP 解封装流程由 udp_in 函数主导。函数首先检查接收到的缓冲区长度是否小于 UDP 首部最小长度，以及 UDP 首部中记录的 total_len16 是否与实际接收长度匹配，不满足条件则直接丢弃。为了确保数据完整性，若接收报文的校验和字段不为 0，则进行校验和验证：先暂存原校验和并将其置 0，重新调用 transport_checksum 计算，若计算结果（经 0 转 0xFFFF 处理后）与原校验和不一致，则视为误码丢弃。验证通过后，提取目的端口号，通过 map_get 在 udp_table 映射表中查找对应的端口处理回调函数。若未找到注册的端口处理程序，说明目标端口未开放，此时需恢复 IP 首部（调用 buf_add_header），并调用 icmp_unreachable 发送“端口不可达”差错报文通知发送方。若查找成功，则移除 UDP 首部，直接调用对应的回调函数将数据载荷交付给上层应用处理。

UDP 校验和计算在 udp_checksum 函数中实现。首先需要构造 UDP 伪首部，伪首部包含源 IP 地址、目的 IP 地址、全零保留字节、协议号（17）以及 UDP 数据报总长度。为了避免在内存中频繁申请和释放空间，设计采用了复用数据缓冲区前端空间的策略：首先调用 buf_add_header 向前扩展缓冲区以覆盖 IP 首部空间，将原有的 IP 首部数据暂存至局部变量中；随后调整指针位置，在紧邻 UDP 首部之前填充构造好的 12 字节伪首部。若数据载荷长

度为奇数，则在末尾填充一个字节 0 以确保两字节对齐。接着，调用 checksum16 函数对伪首部、UDP 首部及数据部分进行 16 位反码求和计算。计算完成后，需移除填充字节（如有），并将缓冲区指针恢复至初始状态，同时将暂存的 IP 首部数据还原回缓冲区，确保不破坏原有的网络层信息，最终返回计算出的校验和结果。

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)：连接建立过程完全依据 TCP 状态机逻辑在 tcp_in 函数中实现。当连接处于 TCP_STATE_LISTEN 状态并收到 SYN 报文时，系统生成初始随机序列号（ISN）写入 TCB 的 seq 字段，将 ack 字段设置为对方序列号加一，并构造带有 SYN | ACK 标志的回复报文，随后状态迁移至 TCP_STATE_SYN_RECEIVED。当连接处于 TCP_STATE_SYN_RECEIVED 状态并收到合法的 ACK 报文时，系统验证确认号无误后，将状态更新为 TCP_STATE_ESTABLISHED，标志着连接正式建立，可以开始双向数据传输。

数据传输与可靠性保障：在 TCP_STATE_ESTABLISHED 状态下，协议实现了严格的序列号检查与确认机制。对于接收到的数据包，系统首先校验其序列号是否与本地期望的 ack 一致，若不一致（如乱序或丢包），则丢弃数据并发送重复 ACK 以触发对方重传。若序列号匹配，系统计算有效数据负载长度，更新本地 ack 值（ack += data_len），并将数据剥离头部后通过回调函数交付给上层应用。发送方向（tcp_send 与 tcp_out）实现了捎带确认机制，在发送数据或控制报文时，总是填入当前最新的确认号和窗口大小（固定为 TCP_MAX_WINDOW_SIZE）。此外，发送逻辑中包含了伪首部校验和的计算，确保了端到端的完整性验证。

连接释放（挥手过程）：本设计实现了一种优化的被动关闭逻辑。当处于 TCP_STATE_ESTABLISHED 状态的连接收到 FIN 报文时，表明对端已完成数据发送。系统将本地 ack 值加一（消费 FIN 的序列号），并在回复报文中同时置位 ACK 和 FIN 标志，直接进入 TCP_STATE_LAST_ACK 状态。这一处理实际上将标准的四次挥手过程中的中间两步（发送 ACK 和发送 FIN）合并执行。最后，当在 TCP_STATE_LAST_ACK 状态下收到对端对自己 FIN 的 ACK 确认报文后，系统调用 tcp_close_connection 移除连接上下文，彻底释放连接资源。

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。）

Web 服务器基于 TCP 协议设计，通过 main 函数中的 net_init() 初始化网络协议栈，并利用 tcp_open(HTTP_LISTEN_PORT, http_request_handler) 在 80 端口注册连接请求的回调函数，随后进入 net_poll() 主循环轮询网络事件。当 TCP 连接建立并收到数据时，核心处理函数 http_request_handler 首先解析 HTTP 请求报文，验证请求方法是否为 "GET" 并提取目标 URL 路径。随后调用 http_respond 函数处理资源请求：首先构建本地文件路径，若请求路径为根目录“/”则默认映射为“/index.html”，否则拼接 HTTP_RESOURCE_DIR 与请求路径。服务器尝试以二进制只读方式 (rb) 打开目标文件，若文件打开失败，系统将构建包含 404 状态行、Keep-Alive 连接信息及 HTML 报错页面的响应报文，并通过 tcp_send 发送至客户端；若文件存在，服务器则构建 HTTP 200 OK 响应，利用 http_get_mime_type 辅助函数根据文件后缀（如 .html, .css, .jpg）判定并设置 Content-Type，

同时通过 fseek 和 ftell 计算文件大小以设置 Content-Length。发送完标准 HTTP 头部和空行分隔符后，程序进入循环读取阶段，使用 fread 分块读取文件内容并发送，直至文件传输完毕并关闭文件句柄，从而完成一次完整的 HTTP 请求响应流程。

二、 实验结果截图及分析

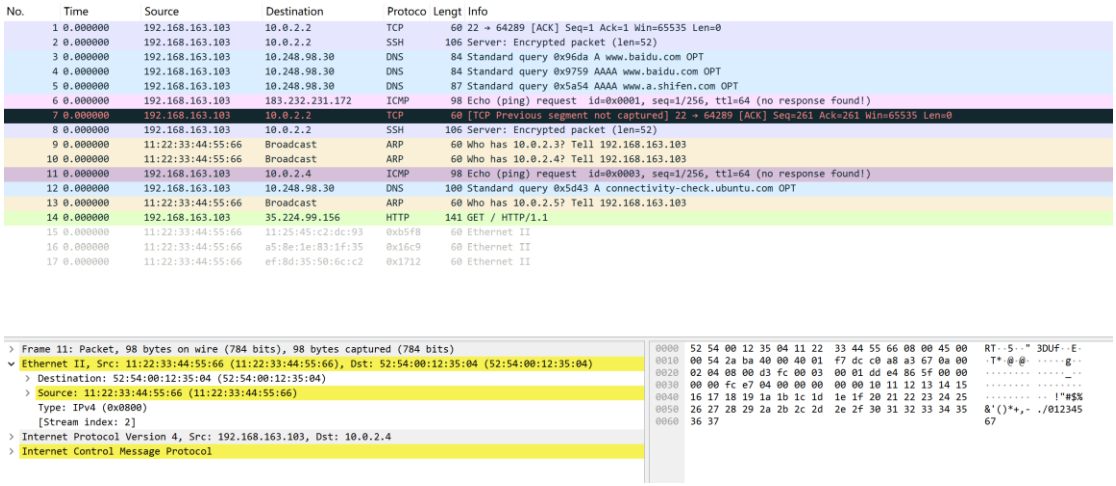
（请展示并详细分析实验结果的截图。可以利用 log 文件、通过 Wireshark 打开的 pcap 文件或 Wireshark 实时捕获的网络报文。）

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）

```
Round 03 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 00 00 46 fb 7c 40 00 40 11 c6 05 0a 00 02 0f c0 a8 a3 67 ae 1b 00 35 00 32 79 68 96 da 01 00 00 01 00 00 00 00 00 01 03 77 77 77 05 62 61 69 64 75 03

Round 04 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 00 00 46 fb 7d 40 00 40 11 c6 04 0a 00 02 0f c0 a8 a3 67 bf 6a 00 35 00 32 79 68 97 59 01 00 00 01 00 00 00 00 00 01 03 77 77 77 05 62 61 69 64 75 03

Round 05 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 00 00 49 fb 7e 40 00 40 11 c6 00 0a 00 02 0f c0 a8 a3 67 84 9f 00 35 00 35 79 6b 5a 54 01 00 00 01 00 00 00 00 00 01 03 77 77 77 01 61 06 73 68 69 66

Round 06 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 00 00 54 01 f4 40 00 40 01 8d 11 0a 00 02 0f c0 a8 a3 67 08 00 3b 6a 00 01 00 01 c8 e4 86 5f 00 00 00 00 ae 7c 00 00 00 00 00 00 10 11 12 13 14 15 16

Round 07 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 08 00 28 06 fe 40 00 40 06 1b ba 0a 00 02 0f c0 a8 a3 67 00 16 fb 21 4f 43 b1 3b 22 ea f9 23 50 10 ff ff 18 2b 00 00

Round 08 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 08 00 5c 06 ff 40 00 40 06 1b 85 0a 00 02 0f c0 a8 a3 67 00 16 fb 21 4f 43 b1 3b 22 ea f9 23 50 18 ff ff 18 5f 00 00 43 73 d2 49 55 63 e4 ce 3c 5e 8e

Round 09 ----------------------------
arp_in:
    mac:08:00:27:6c:48:4f
    buf: 00 01 08 00 06 04 00 01 08 00 27 6c 48 4f 0a 00 02 0f 11 22 33 44 55 66 c0 a8 a3 67

Round 10 ----------------------------
arp_in:
    mac:08:00:27:6c:48:4f
    buf: 00 01 08 00 06 04 00 01 08 00 27 6c 48 4f 0a 00 02 0f 11 22 33 44 55 66 c0 a8 a3 67
```

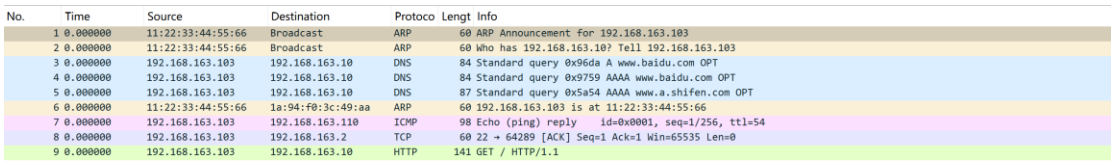
基于日志文件中的实例分析，以太网帧的封装与解封装过程完全符合 ethernet.c 的逻辑设计。对于 IP 协议数据包，日志标签 ip_in 表明以太网头部中的协议类型字段 0x0800 已被正确识别；此时 buf 数据均以 0x45 开头，证明 14 字节的以太网头已被 buf_remove_header 移除，显露出的 IPv4 头部及其后续负载保持了结构完整。对于 ARP 协议数据包，日志标签 arp_in 对应了以太网类型 0x0806 的分发逻辑，其 buf 内容起始于 00 01 08 00，准确对应了 ARP 包头中的硬件类型（以太网）和协议类型（IPv4）字段；同时，无论是 IP 还是 ARP 帧，日志中记录的 mac:08:00:27:6c:48:4f 均证实了 ethernet_in 函数在剥离头部前已成功提取了源 MAC 地址并将其传递给上层处理。

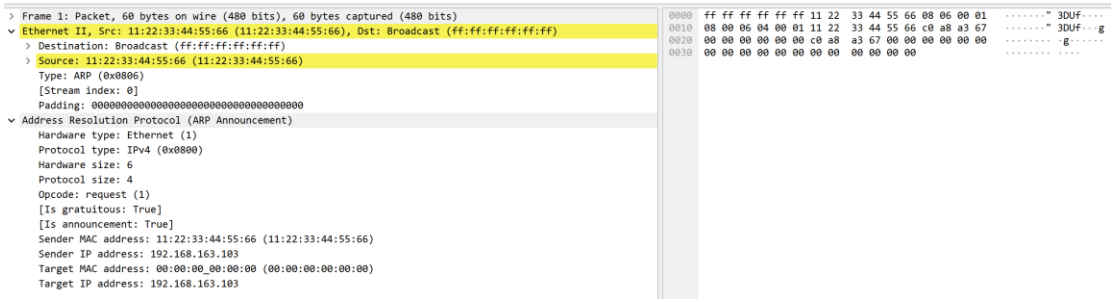


该帧的结构和内容完全符合以太网协议（Ethernet II）的标准预期。帧头的前 6 个字节正确封装了目的 MAC 地址 52:54:00:12:35:04，紧随其后的是源 MAC 地址 11:22:33:44:55:66，这与下方十六进制视图中的 52 54 00 12 35 04 及 11 22 33 44 55 66 一一对应。协议类型字段为 0x0800，准确标识了上层载荷为 IPv4 协议。数据部分以 0x45 开头，表明这是一个标准的 IPv4 头部，Wireshark 已成功将其解析为 ICMP Echo 请求（Ping 包），证明以太网帧对上层数据的封装是完整且正确的。

2. ARP 协议实验结果及分析

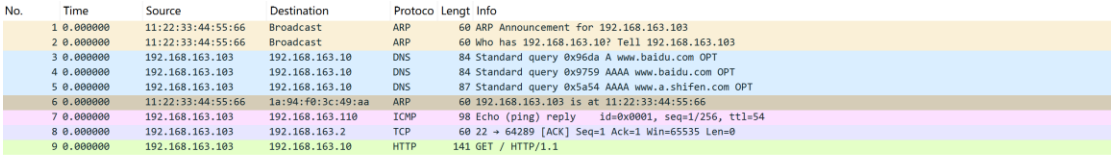
（展示 ARP 请求和响应包的捕获截图，分析其请求和响应过程是否正常。检查 ARP 请求中的目标 IP 地址和发送方 MAC 地址，ARP 响应中的目标 MAC 地址。）



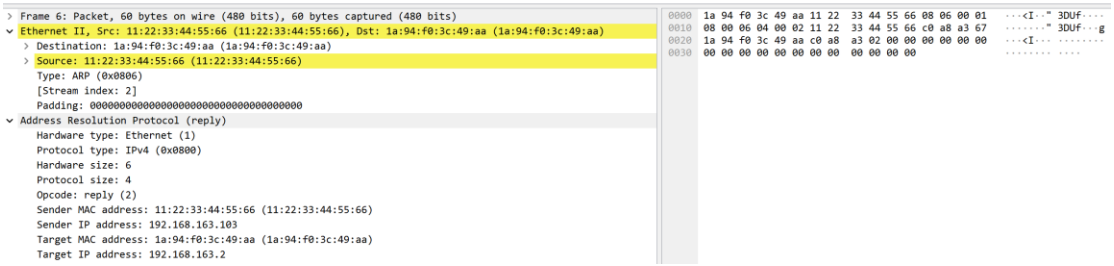


ARP 协议的请求与响应过程均符合预期规范，交互逻辑正常。

在 ARP 请求报文（Frame 1）中，Opcode 被标记为 1（Request），发送方 MAC 地址正确填充为 11:22:33:44:55:66，目标 IP 地址正确填充为本机 IP 地址（无回报 ARP 包规定），且以太网帧的目的地址为广播地址 ff:ff:ff:ff:ff:ff，表明主机正在通过广播方式通告本机要使用 IP 地址 192.168.163.103。



No.	Time	Source	Destination	Protoco	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=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



```
> Frame 6: Packet, 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
v 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: 000000000000000000000000000000000000
v 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 00 01   ···<I·· 3DUf····
0010  08 00 06 04 00 02 11 22  33 44 55 66 c0 a8 a3 67   ·······" 3DUf···g
0020  1a 94 f0 3c 49 aa c0 a8  a3 02 00 00 00 00 00 00   ···<I··· ········
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

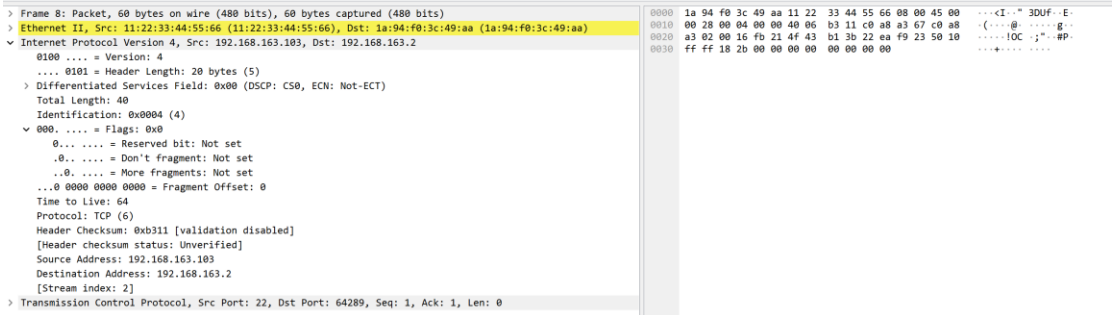
在 ARP 响应报文（Frame 6）中，Opcode 为 2（Reply），报文通过单播形式回复，其中目标 MAC 地址准确指向了请求端 1a:94:f0:3c:49:aa，同时发送方在 Sender MAC 字段中填入了自身的物理地址 11:22:33:44:55:66 以响应该 ARP 请求。

综上所述，ARP 报文中的 MAC 地址与 IP 地址映射关系正确，请求的广播机制与响应的单播机制均执行无误，证明了 ARP 协议功能的正确性。

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。）

No.	Time	Source	Destination	Protoco	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

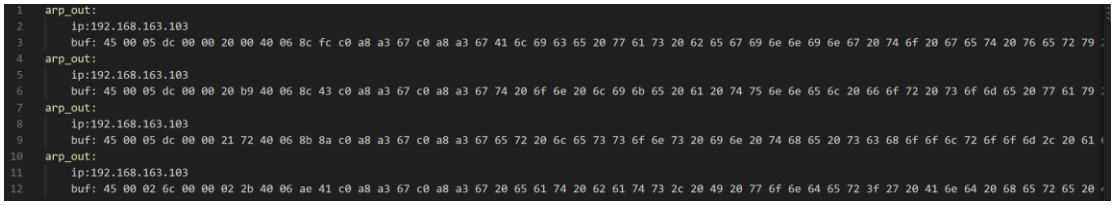


```
> Frame 8: Packet, 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 1a:94:f0:3c:49:aa (1a:94:f0:3c:49:aa)
v Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.2
      0100 .... = Version: 4
      .... 0101 = Header Length: 20 bytes (5)
    > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      Total Length: 40
      Identification: 0x0004 (4)
    v 000. .... = Flags: 0x0
        0... .... = Reserved bit: Not set
        .0.. .... = Don't fragment: Not set
        ..0. .... = More fragments: Not set
      ...0 0000 0000 0000 = Fragment Offset: 0
      Time to Live: 64
      Protocol: TCP (6)
      Header Checksum: 0xb311 [validation disabled]
      [Header checksum status: Unverified]
      Source Address: 192.168.163.103
      Destination Address: 192.168.163.2
      [Stream index: 2]
> Transmission Control Protocol, Src Port: 22, Dst Port: 64289, Seq: 1, Ack: 1, Len: 0
```

```
0000  1a 94 f0 3c 49 aa 11 22  33 44 55 66 08 00 45 00   ···<I·· 3DUf··E·
0010  00 28 00 04 00 00 40 06  b3 11 c0 a8 a3 67 c0 a8   ·(····@· ·····g··
0020  a3 02 00 16 fb 21 4f 43  b1 3b 22 ea f9 23 50 10   ·····!OC ·;"··#P·
0030  ff ff 18 2b 00 00 00 00  00 00 00 00               ···+···· ····
```

根据实验捕获的第 8 号数据包（Frame 8）分析，该报文为从源地址 192.168.163.103 发

往目的地址 192.168.163.2 的 TCP 确认报文，IP 协议头部字段填写均正确且符合预期。版本号（Version）为 4，与代码中定义的 IP_VERSION_4 一致，表明使用 IPv4 协议；首部长度（Header Length）为 20 字节（值为 5），符合 ip_fragment_out 函数中固定设置的头部长度 5（即 5×4=20 字节），且无可选字段；总长度（Total Length）显示为 40 字节，这由 20 字节 IP 头部和 20 字节 TCP 头部组成（数据载荷为 0），表明 total_len16 字段计算及大小端转换正确；标识（Identification）字段值为 0x0004，反映了 ip_out 函数中静态计数器 ip_id 的累加逻辑；标志位（Flags）为 0x00 且片偏移（Fragment Offset）为 0，这是因为该包总长度（40 字节）远小于以太网 MTU 限制，代码进入 ip_out 中的非分片分支，直接调用 ip_fragment_out 并传入 offset=0 和 mf=0；生存时间（TTL）为 64，与头文件中定义的宏 IP_DEFALUT_TTL 一致；协议（Protocol）字段为 6（TCP），正确标识了上层协议类型。综上所述，捕获的数据包证明了 IP 数据包的封装、长度计算及字段赋值逻辑在非分片情况下均运行正确。

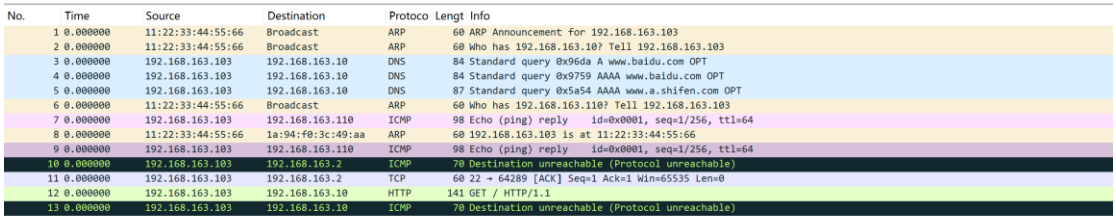


根据日志文件（ip_frag_test/log），我们可以对 IP 协议的分片机制进行详细的数据包分析，以验证其正确性。

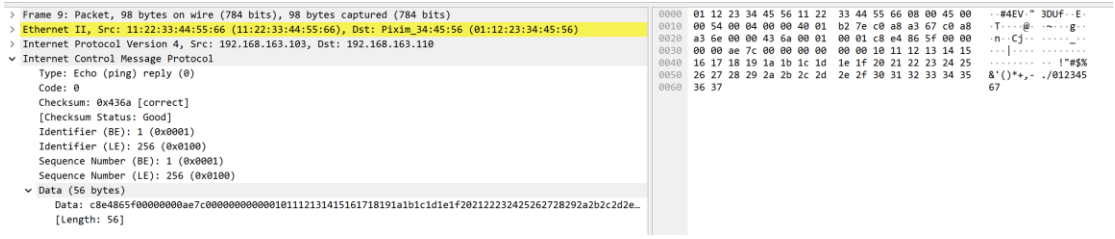
日志记录了 ip_out 函数将一个总载荷为 5040 字节的大数据包分片发送的过程，共产生了四个 IP 分片数据包。第一个分片的头部数据显示为 45 00 05 dc 00 00 20 00，其中总长度字段 05 dc 即 1500 字节，减去 20 字节 IP 头部后，有效载荷为 1480 字节，符合以太网 MTU 限制；标志与偏移字段 20 00 表明更多分片（MF）置位为 1，片偏移为 0，即数据的起始部分。第二个分片头部包含 45 00 05 dc ... 20 b9，总长度仍为 1500 字节（载荷 1480 字节），标志位 MF 仍为 1，而片偏移字段 00 b9（十进制 185）乘以 8 后恰好等于 1480 字节，紧密衔接了第一个分片的尾部。第三个分片头部为 45 00 05 dc ... 21 72，同样满载发送 1480 字节数据，MF 为 1，片偏移 01 72（十进制 370）乘以 8 等于 2960 字节，准确对应前两个分片的总和（1480 + 1480）。第四个分片作为最后一个包，其头部显示 45 00 02 6c ... 02 2b，总长度 02 6c 即 620 字节，减去头部后剩余载荷为 600 字节（总数据 5040 - 4440）；标志位 02 2b 显示 MF 置 0，表示分片结束，且片偏移 02 2b（十进制 555）乘以 8 等于 4440 字节，精确接续了前三个分片的数据。此外，所有分片的标识字段（Identification）均为 00 00，证明它们属于同一个原始数据包。综上所述，日志数据证实该 IP 协议实现的分片逻辑在长度计算、标志位设置以及偏移量推导上均准确无误。

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。）



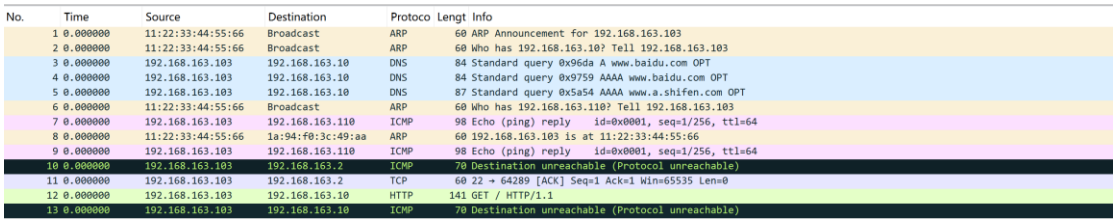
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	ARP Announcement for 192.168.163.103
2	0.000000	11:22:33:44:55:66	Broadcast	ARP	60	Who has 192.168.163.10? Tell 192.168.163.103
3	0.000000	192.168.163.103	192.168.163.10	DNS	84	Standard query 0x96da A www.baidu.com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



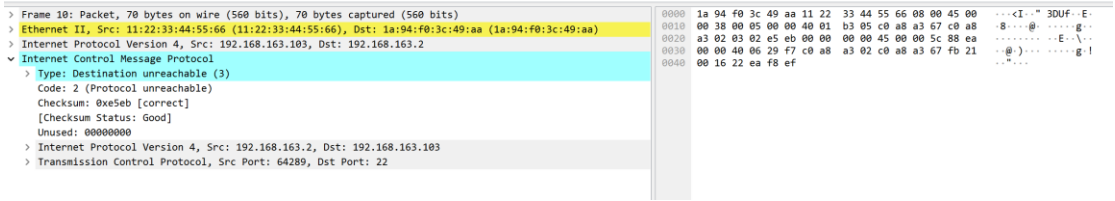
```
> Frame 9: Packet, 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
> Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: Pixim_34:45:56 (01:12:23:34:45:56)
> Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.110
v Internet Control Message Protocol
    Type: Echo (ping) reply (0)
    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)
  v Data (56 bytes)
      Data: c8e4865f00000000ae7c0000000000001011121314151617181 9a1b1c1d1e1f2021222324252627282 92a2b2c2d2e…
      [Length: 56]
```

```
0000  01 12 23 34 45 56 11 22  33 44 55 66 08 00 45 00   ··#4EV·" 3DUf··E·
0010  00 54 00 04 00 00 40 01  b2 7e c0 a8 a3 67 c0 a8   ·T····@· ·~···g··
0020  a3 6e 00 00 43 6a 00 01  00 01 c8 e4 86 5f 00 00   ·n··Cj·· ·····_··
0030  00 00 ae 7c 00 00 00 00  00 00 10 11 12 13 14 15   ···|···· ········
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ········ ·· !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                              67
```

查询报文：在第 9 帧中，捕获到了一个由本机（192.168.163.103）发出的 ICMP 报文。详细信息显示其 Type 字段为 0，Code 字段为 0，这表明该报文为 Echo Reply（回显应答）。这验证了 icmp_in 函数正确识别了 Type 为 8 的请求报文，并调用 icmp_resp 函数成功构造了响应。报文的 Checksum 字段显示为“correct”，说明 checksum16 计算正确。此外，报文的数据部分（Data）完整保留了标识符（Identifier: 1）和序列号（Sequence Number: 1）以及 56 字节的载荷数据，证明 icmp_resp 函数中的 memcpy 操作正确地将请求报文的载荷复制到了应答报文中，实现了 Ping 功能的回路测试。



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



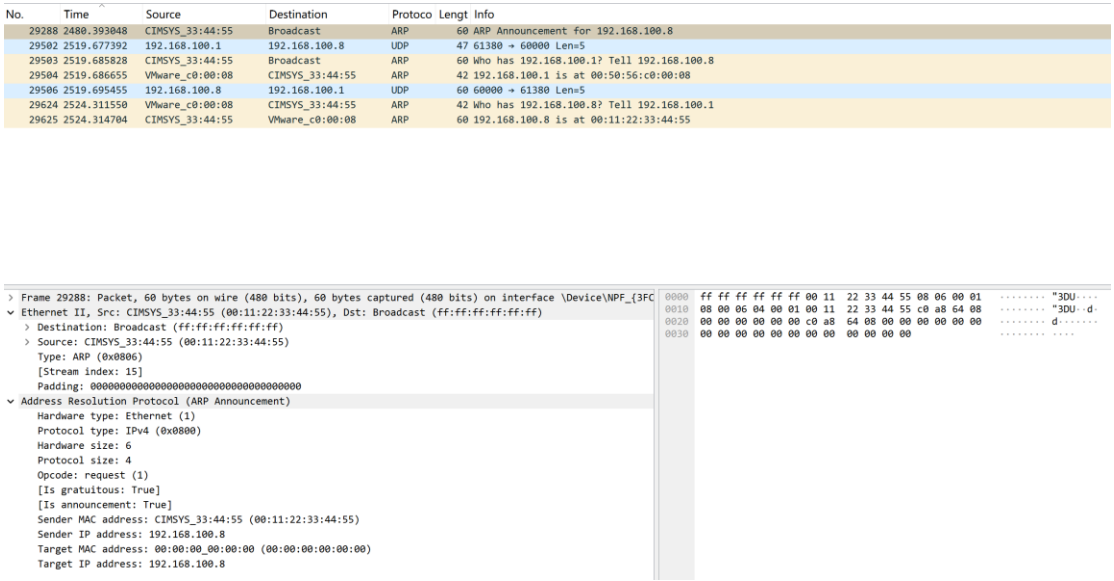
```
> Frame 10: Packet, 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
> Ethernet II, Src: 11:22:33:44:55:66 (11:22:33:44:55:66), Dst: 1a:94:f0:3c:49:aa (1a:94:f0:3c:49:aa)
> Internet Protocol Version 4, Src: 192.168.163.103, Dst: 192.168.163.2
v Internet Control Message Protocol
  > Type: Destination unreachable (3)
    Code: 2 (Protocol unreachable)
    Checksum: 0xe5eb [correct]
    [Checksum Status: Good]
    Unused: 00000000
  > Internet Protocol Version 4, Src: 192.168.163.2, Dst: 192.168.163.103
  > Transmission Control Protocol, Src Port: 64289, Dst Port: 22
```

```
0000  1a 94 f0 3c 49 aa 11 22  33 44 55 66 08 00 45 00   ···<I·" 3DUf··E·
0010  00 38 00 05 00 00 40 01  b3 05 c0 a8 a3 67 c0 a8   ·8····@· ·····g··
0020  a3 02 03 02 e5 eb 00 00  00 00 45 00 00 5c 88 ea   ········ ··E··\··
0030  00 00 40 06 29 f7 c0 a8  a3 02 c0 a8 a3 67 fb 21   ··@·)··· ·····g·!
0040  00 16 22 ea f8 ef                                 ··"···
```

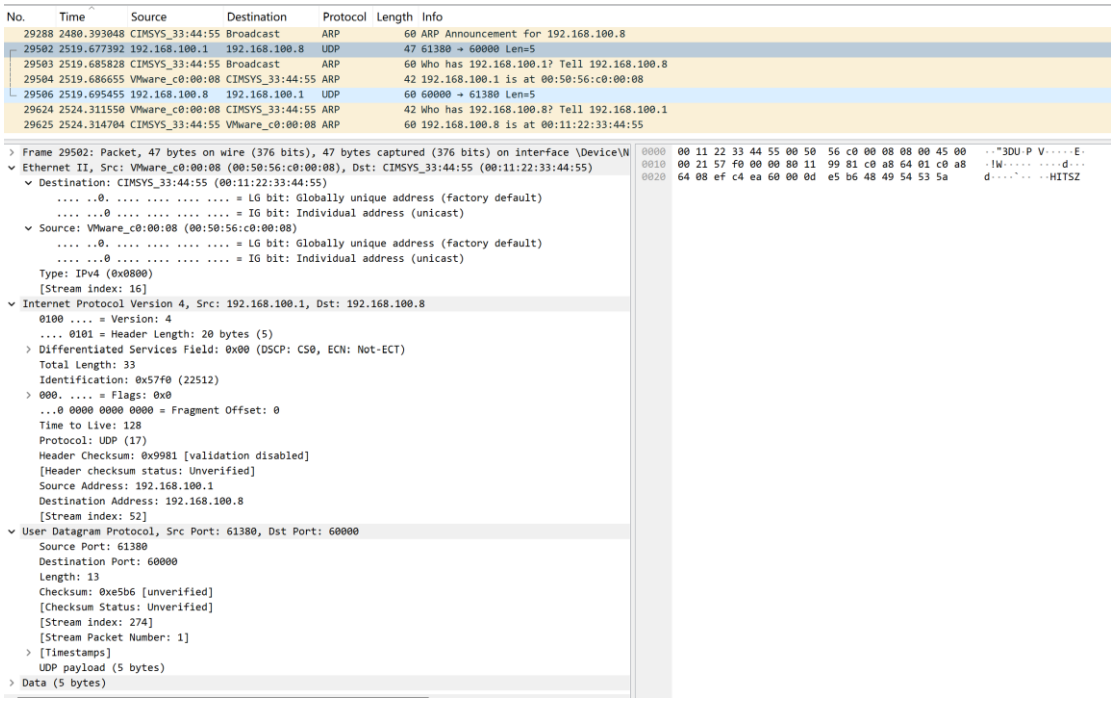
差错报文：在第 10 帧中，捕获到了一个 Type 字段为 3，Code 字段为 2 的 ICMP 报文，即 Protocol Unreachable（协议不可达）。这对应了 icmp_unreachable 函数的逻辑：当底层接收到无法处理的上层协议包时，系统触发了差错报告。报文的 Checksum 字段显示为“correct”，说明 checksum16 计算正确。分析该报文的载荷部分，可以发现它严格遵循标准，包含原数据包的 IP 首部以及其载荷的前 8 个字节，这使得发送方能够精确定位是哪一个通信尝试遭到了拒绝，从而有效地实现了网络故障诊断功能。

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。）

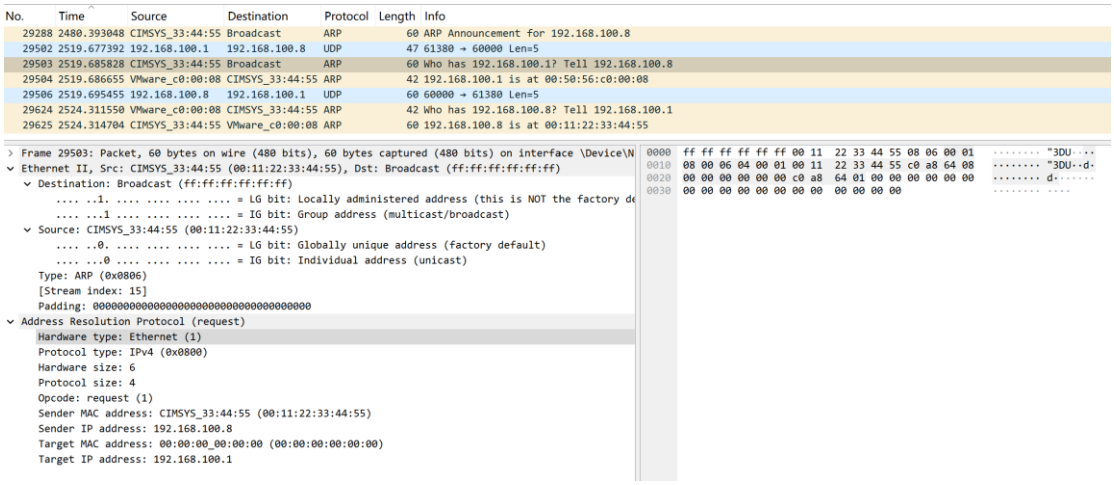


实验启动时，协议栈首先向网络发送了一个免费 ARP 报文，如 Frame 29288 所示。该报文以太网首部的目的 MAC 地址为广播地址 ff:ff:ff:ff:ff:ff，源 MAC 地址为协议栈配置的虚拟 MAC 00:11:22:33:44:55。ARP 载荷中，Sender IP 和 Target IP 均为协议栈自身的 IP 地址 192.168.100.8。这一机制的主要目的是向局域网内的其他设备（如宿主机）通告自身的 IP 与 MAC 地址映射关系，同时检测网络中是否存在 IP 地址冲突。此报文的成功发送表明协议栈的底层驱动及 ARP 模块初始化正常，具备了基本的广播发送能力。

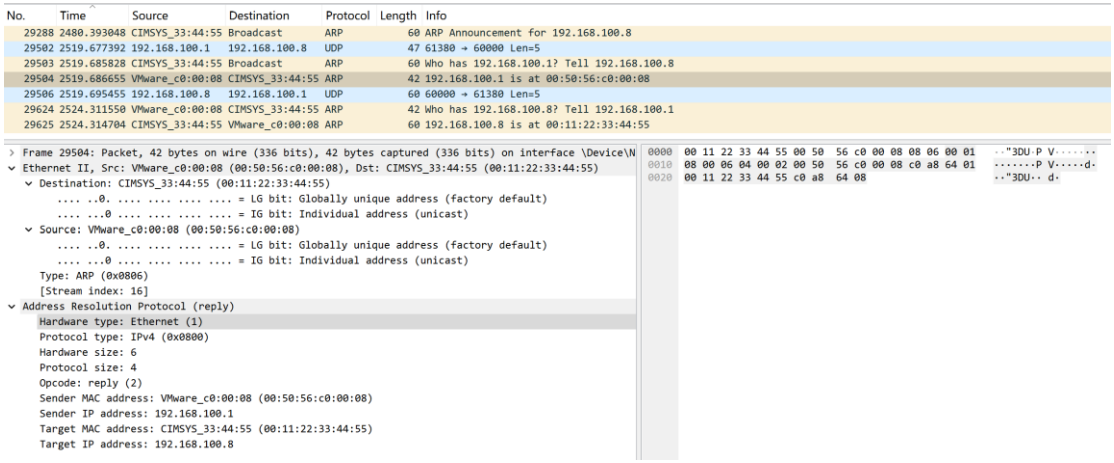


Frame 29502 记录了宿主机向协议栈发送 UDP 测试数据的详细信息。IP 首部显示源 IP 为 192.168.100.1，目的 IP 为 192.168.100.8，协议号 17（UDP）。UDP 首部中，源端口为宿主机随机分配的 61380，目的端口为设置好的 60000。UDP 长度字段为 13 字节，由 8 字节首部和 5 字节数据组成。数据载荷（Payload）部分的十六进制内容为 48 49 54 53 5a，对应

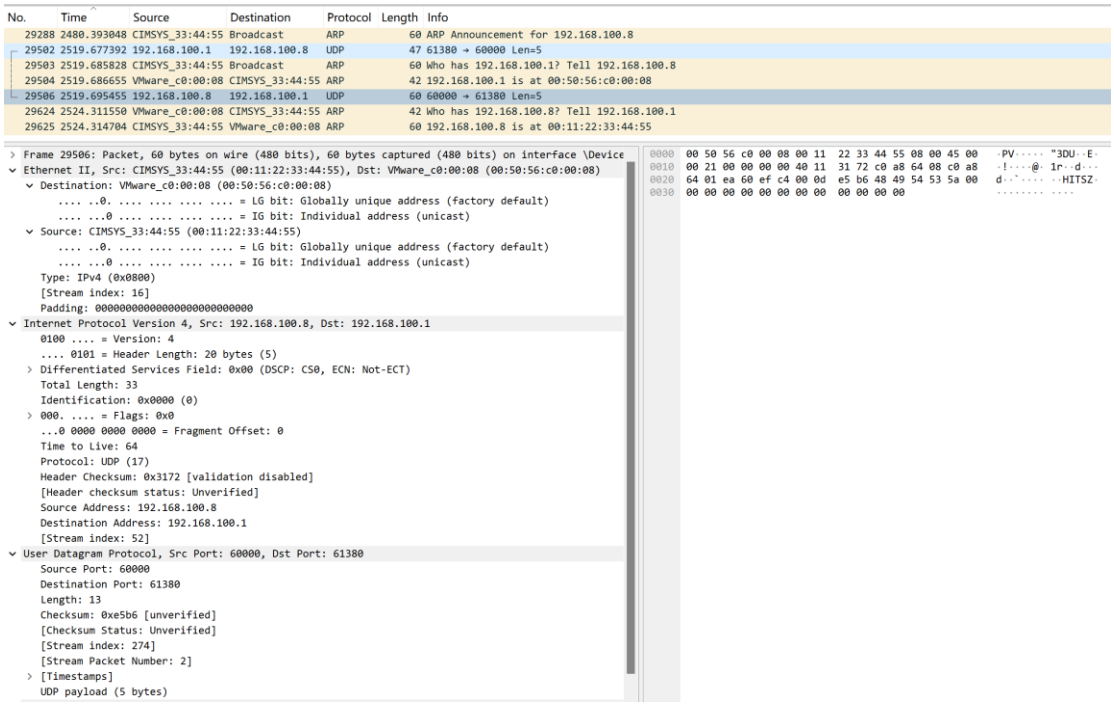
的 ASCII 码正是预期的字符串“HITSZ”。这证明宿主机成功构建并发送了符合协议规范的 UDP 数据包。



为了与宿主机进行通信，协议栈必须先获取目的 IP 的物理地址。Frame 29503 展示了协议栈通过 ARP 广播请求对应物理地址的过程。如图所示，源 IP 地址为 192.168.100.8（协议栈）的主机正在广播一个 ARP 请求，询问宿主机的 MAC 地址。该数据包的 Opcode 为 Request(1)，目标 MAC 地址全为 0，表明这是一个广播帧。这说明在虚拟机尝试向宿主机回送数据时，首先需要通过 ARP 协议获取宿主机的物理地址，为后续的数据链路层封装做准备。



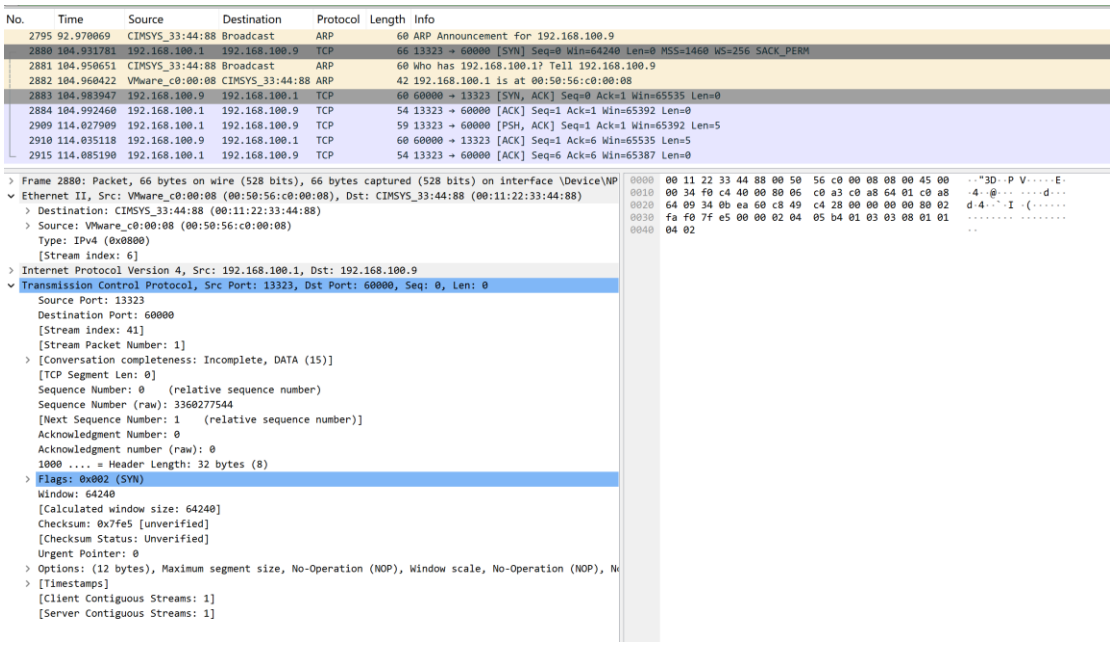
Frame 29504 展示了宿主机对协议栈 ARP 请求的响应过程。报文显示源 MAC 地址为宿主机网卡地址 00:50:56:c0:00:08，目的 MAC 地址为协议栈的 00:11:22:33:44:55。ARP 协议字段中 Opcode 为 2（Reply），明确告知协议栈 IP 192.168.100.1 对应的 MAC 地址。此步骤的完成建立了链路层的单播通信基础，使得协议栈能够正确封装后续回发给宿主机的 IP 数据报。



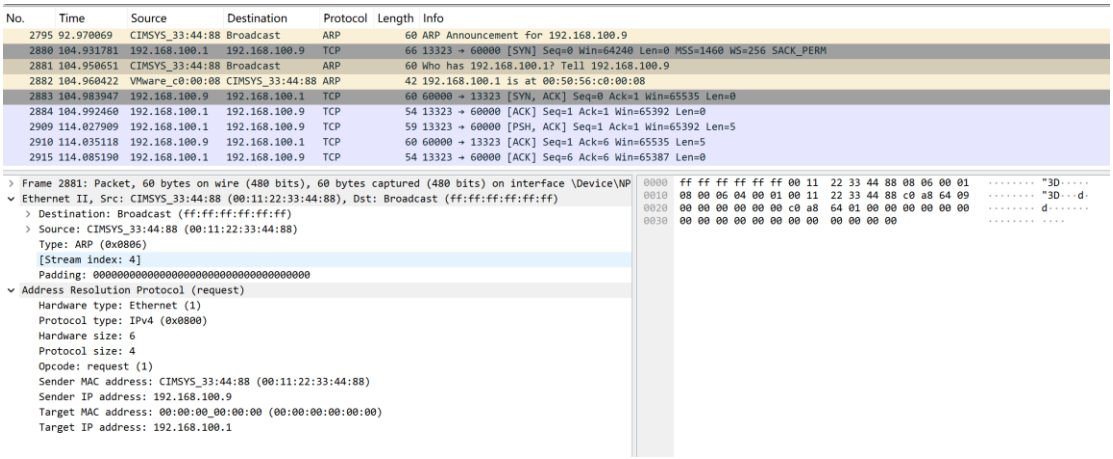
Frame 29506 记录了协议栈向宿主机发送 UDP 测试回复数据的详细信息。IP 首部显示源 IP 为 192.168.100.8，目的 IP 为 192.168.100.1，协议号 17（UDP）。UDP 首部中，源端口为设定好的 60000，目的端口为 61380。UDP 长度字段为 13 字节，由 8 字节首部和 5 字节数据组成。数据载荷（Payload）部分的十六进制内容依然为 48 49 54 53 5a，对应的 ASCII 码正是预期的字符串“HITSZ”。这证明协议栈成功构建并回发了符合协议规范的 UDP 数据包。

6. TCP 协议实验结果及分析

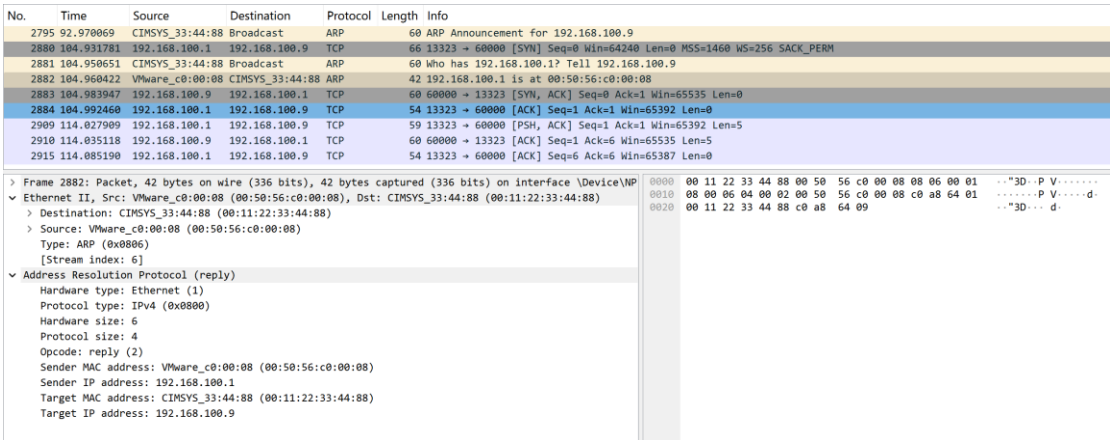
（展示 TCP 数据包的捕获截图，分析 TCP 连接的建立、数据传输和关闭过程。检查 TCP 头部的源端口号、目的端口号、序列号、确认号、标志位等字段，以及连接的状态转换。）



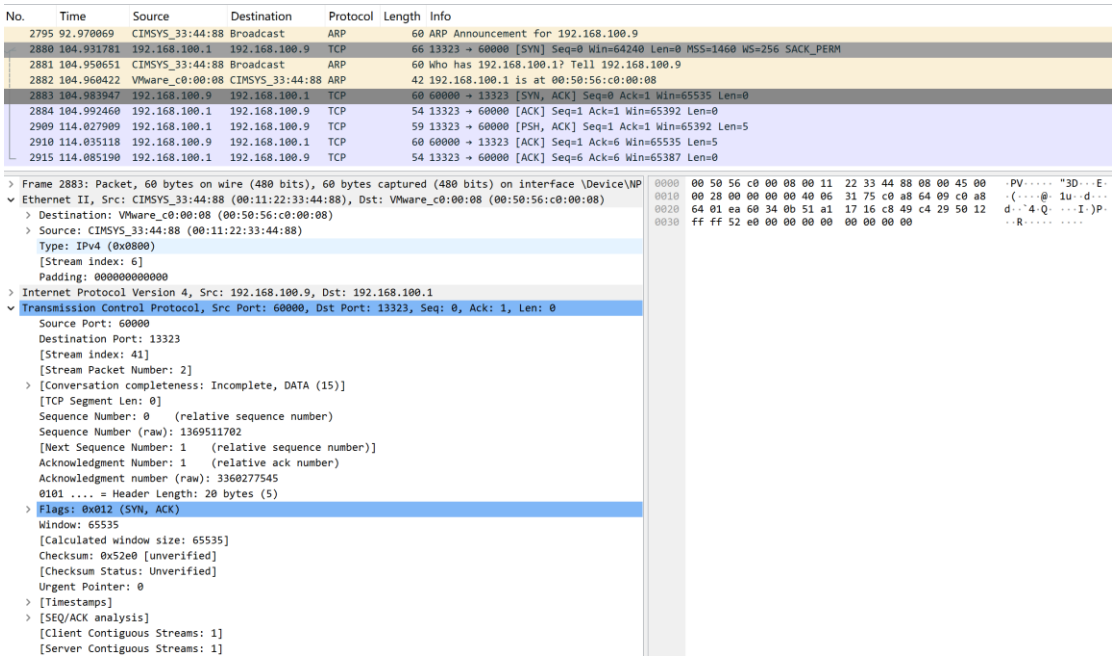
实验开始时，客户端（宿主机 IP: 192.168.100.1）向服务端（虚拟机协议栈 IP: 192.168.100.9）发起主动连接。捕获的数据包显示，源端口为 13323，目的端口为 60000。在 TCP 头部中，标志位 SYN 被置为 1（Flags: 0x002），表明这是一个连接请求报文。此时，客户端生成的初始序列号（Sequence Number）为 0（相对值），确认号（Acknowledgment Number）为 0，数据载荷长度（Len）为 0。该报文标志着 TCP 三次握手过程的第一步，客户端进入 SYN_SENT 状态，等待服务器的确认。



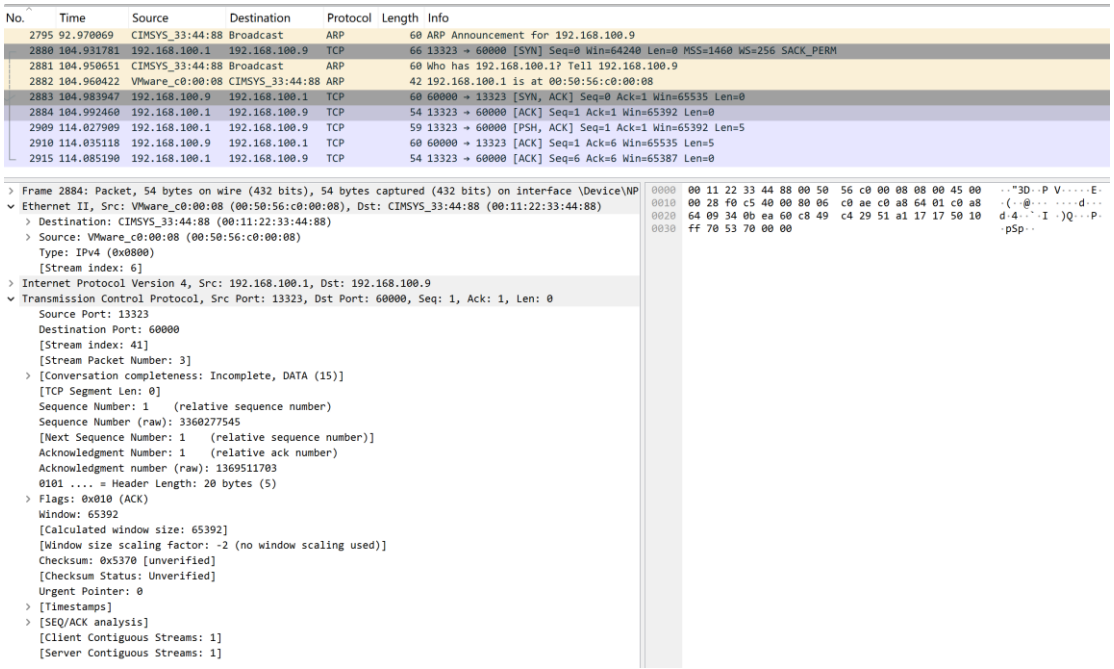
在服务器（192.168.100.9）准备响应客户端的 TCP 连接请求之前，链路层需要获知网关或目的主机的物理地址。截图显示，源地址为 192.168.100.9 的设备发送了一个 ARP 广播请求（Broadcast），询问“Who has 192.168.100.1? Tell 192.168.100.9”。这表明此时虚拟机协议栈虽然收到了 IP 层的数据包，但在构建链路层回复帧时，ARP 缓存中缺失目标 IP 对应的 MAC 地址，因此触发了 ARP 地址解析过程。



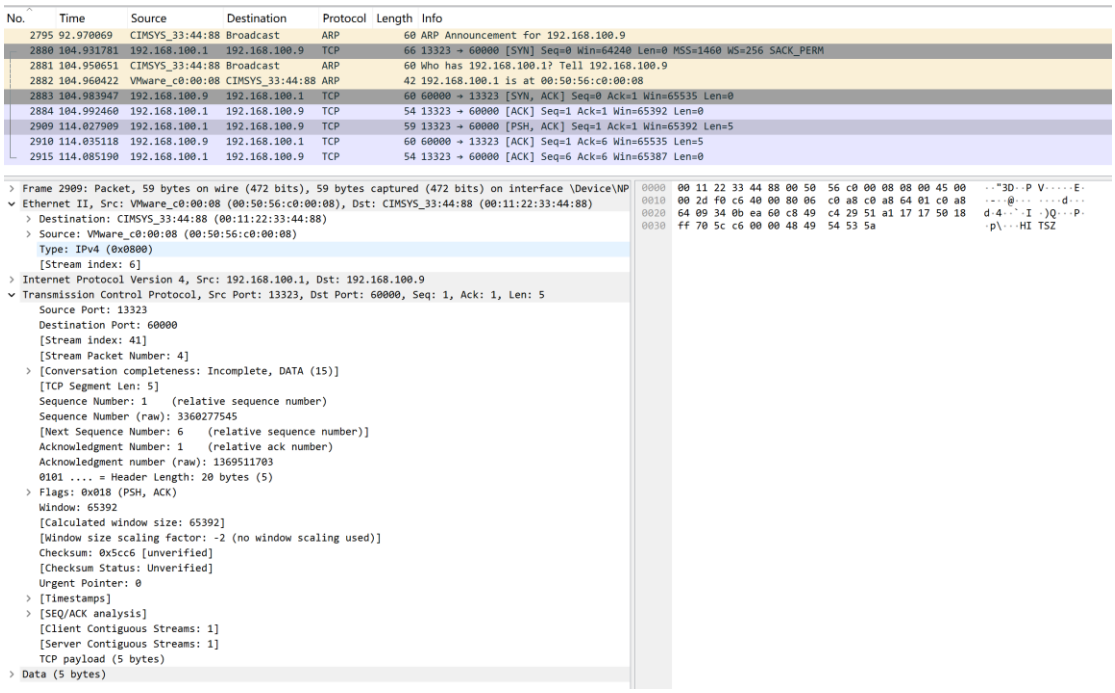
宿主机（192.168.100.1）在收到 ARP 请求后，立即进行了单播回复。数据包显示，源 MAC 地址为 VMware_c0:00:08（对应宿主机虚拟网卡），目的 MAC 地址为 CIMSYS_33:44:88（对应虚拟机协议栈）。ARP 响应报文包含 Opcode: reply (2)，告知对方 192.168.100.1 的物理地址。至此，链路层地址解析完成，双方具备了在以太网层面进行点对点通信的条件，为后续 TCP 握手报文的传输扫清了障碍。



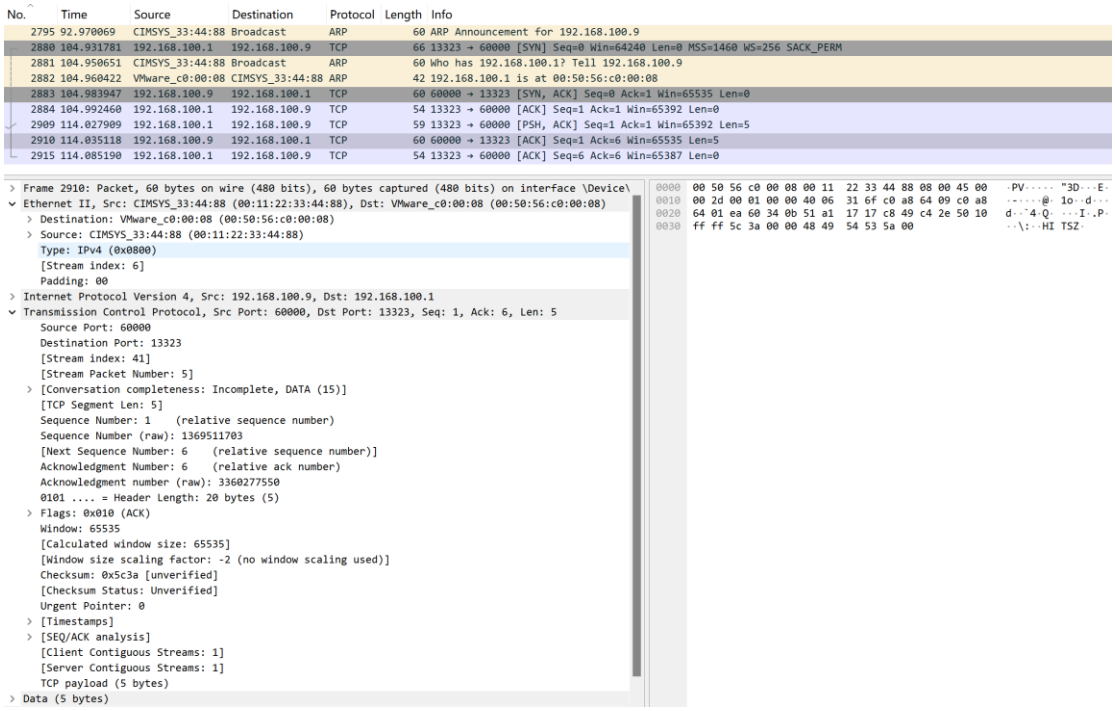
服务器（192.168.100.9）在获取 MAC 地址后，向客户端发送了回复报文。该数据包源端口为 60000，目的端口为 13323。TCP 头部标志位 SYN 和 ACK 同时被置位（Flags: 0x012），表明服务器同意建立连接并确认收到了客户端的 SYN 包。此时，服务器生成的初始序列号（Seq）为 0，确认号（Ack）设置为 1（即客户端初始 Seq + 1），表明期望收到客户端的下一个序列号为 1 的数据。此步骤标志着服务端进入 SYN_RCVD 状态。



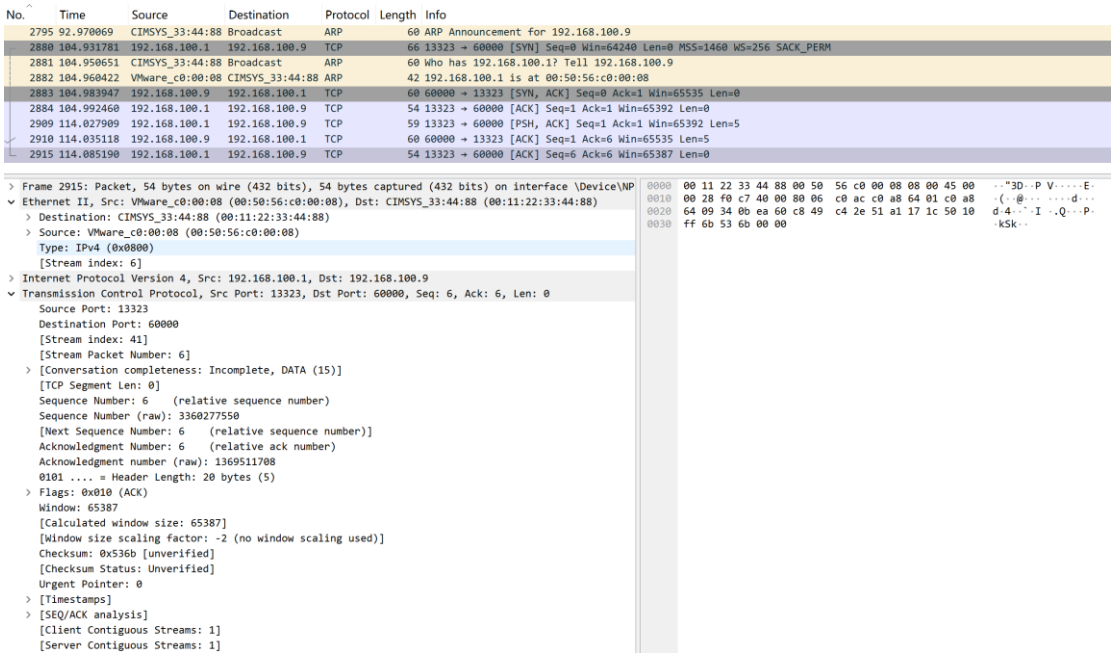
客户端收到服务器的 SYN+ACK 报文后，发送了最终的确认报文。数据包显示，标志位 ACK 被置位（Flags: 0x010）。此时，序列号（Seq）更新为 1，确认号（Ack）更新为 1（即服务器初始 Seq + 1）。该数据包不携带任何数据载荷（Len: 0）。发送该报文后，客户端与服务器端的 TCP 连接状态均转换为 ESTABLISHED，标志着三次握手过程顺利完成，双方建立起了可靠的双向通信通道。



连接建立后，客户端通过 TCP 调试工具发送了文本数据 "HITSZ"。捕获的数据包显示，标志位 PSH 和 ACK 被置位（Flags: 0x018），PSH 标志指示接收方协议栈应尽快将数据交付给上层应用，而不是要在缓冲区排队。数据载荷长度（Len）为 5 字节，右侧 Hex Dump 区域清晰显示了十六进制数据 48 49 54 53 5a，对应 ASCII 码的 "HITSZ"。此时序列号（Seq）仍为 1，确认号（Ack）保持为 1。



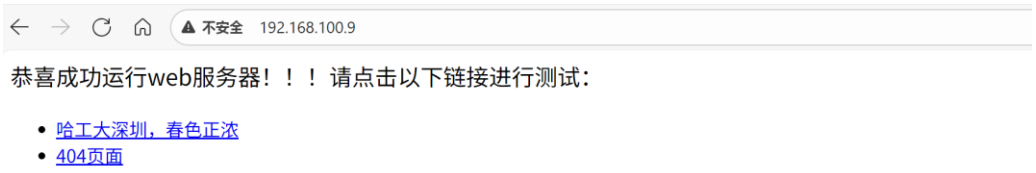
服务器协议栈在收到数据后，立即回复了一个确认报文。数据包显示标志位 ACK 被置位（Flags: 0x010）。关键字段确认号（Ack）更新为 6。这是根据 TCP 累积确认机制计算得出的：收到 Seq=1 且长度为 5 的数据包后，期望的下一个字节序号为 1 + 5 = 6。此报文表明服务器已成功且完整地接收了 "HITSZ" 字符串，实现了数据的可靠传输。



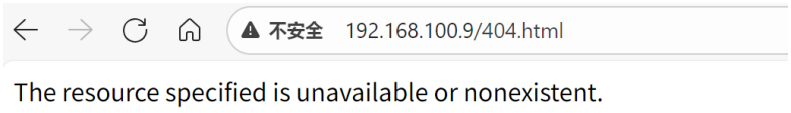
在数据传输结束后的 50ms 左右，客户端发送了一个纯 ACK 报文。该报文序列号（Seq）更新为 6，确认号（Ack）维持为 6。虽然不携带数据，但这一报文通常用于确认链路状态或更新接收窗口（Window Size）。截图显示此时客户端通告的窗口大小为 65387。由于本次实验仅进行了单次数据发送，且未包含断开连接（FIN）的操作，该数据包维持了连接的 ESTABLISHED 状态，等待后续可能的数据交互。

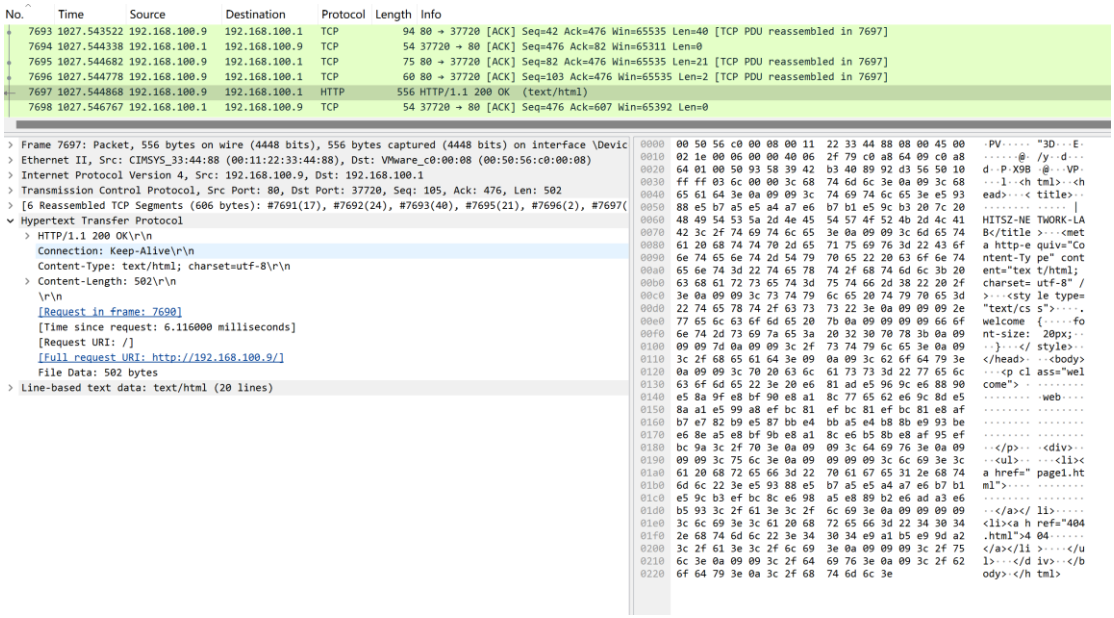
7. web 服务器实验结果及分析

（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。）

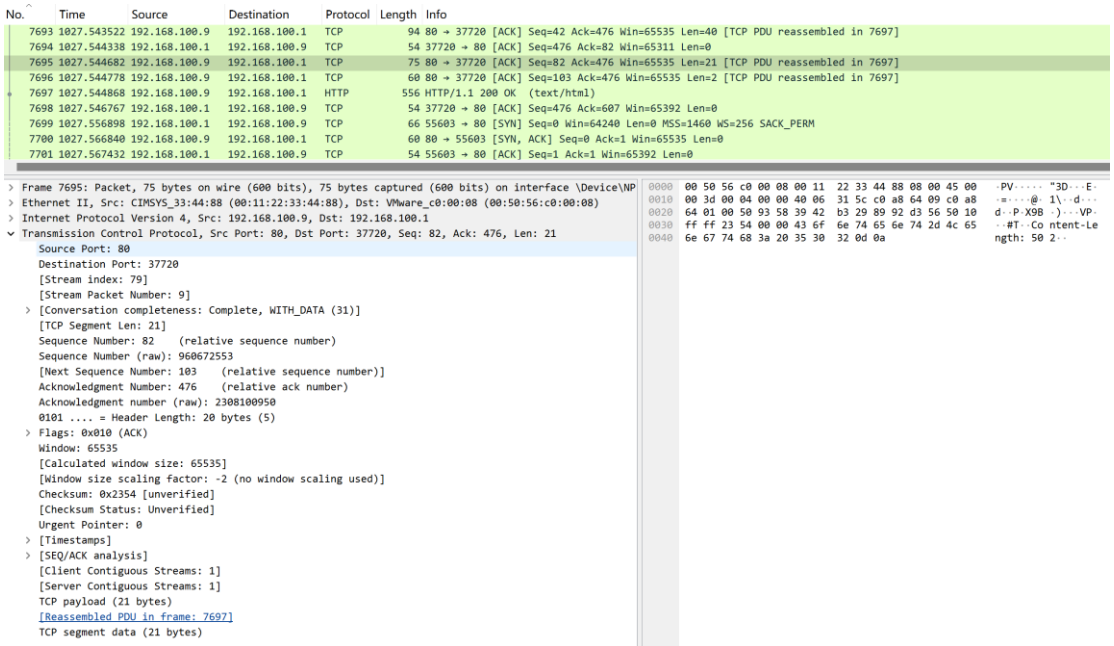



不安全 192.168.100.9/page1.html
图片来自哈尔滨工业大学深圳校区微信公众号，链接：https://mp.weixin.qq.com/s/-GO8LwPcSK1HW86F0hWGew
如期而至的春天，你好！
HITSZ
慢
学苑大道


不安全 192.168.100.9/404.html
The resource specified is unavailable or nonexistent.

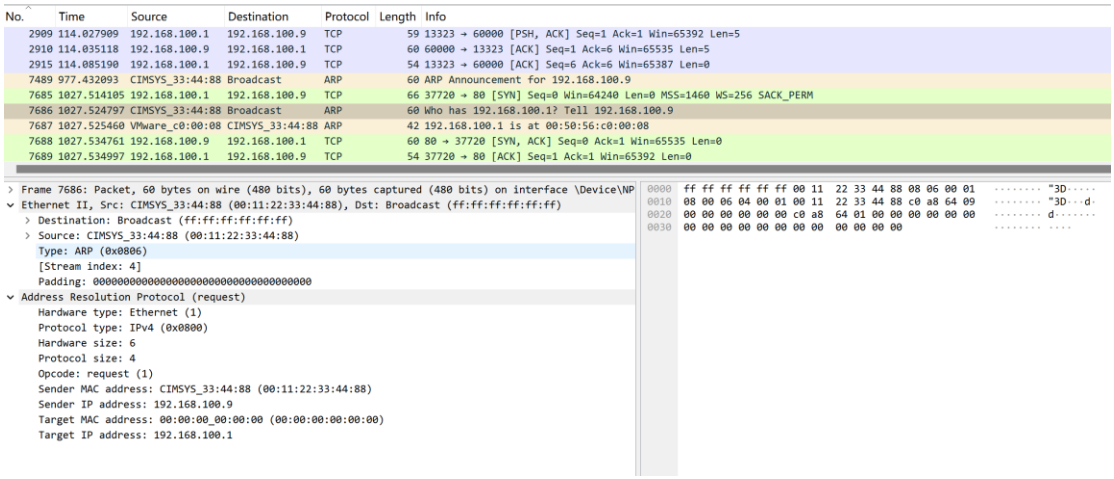


Frame 7697 详细展示了服务器返回的 HTTP 响应报文。从 Info 列及具体解码内容可见，该次通信基于 HTTP/1.1 协议，响应状态码为 200 OK，表明客户端的请求已被服务器成功处理。在响应头（Response Headers）中，Content-Type: text/html; charset=utf-8 标识了返回的数据类型为 UTF-8 编码的 HTML 文档；Content-Length: 502 指明了响应体的大小为 502 字节；Connection: Keep-Alive 表示连接将保持激活状态以复用 TCP 通道。通过右侧的 Hex/ASCII 视图可以清晰地看到响应体（Response Body）的具体 HTML 代码，其中包含了 <title>HITSZ-NETWORK-LAB</title> 以及 <p class="welcome"> 等标签，证实了 Web 服务器正在正常运行并将预设的实验页面正确返回给了客户端。



Frame 7695 展示了 HTTP 应用层数据在传输层被封装为 TCP 数据段（Segment）的细节。由于 HTTP 响应报文的大小（头部加实体共约 556 字节）可能超过了单个 TCP 段的传输策略或受限于滑动窗口机制，Wireshark 提示 "[TCP PDU reassembled in 7697]"，说明完整的 HTTP 响应是由多个 TCP 数据段重组而成的。图中的 Frame 7695 是其中一个

分段，源端口为 80，目的端口为 37720，虽然该段仅携带了 21 字节的有效载荷（Len=21），但它与序列号（Seq）相邻的其他数据包（如 Frame 7693, 7694, 7696 等）共同依序组成了上层完整的 HTTP 响应报文，体现了 TCP 协议面向字节流及分段传输的特性。



根据捕获的数据包 Frame 7686 显示，在进行 HTTP 通信前，首先完成了链路层地址解析与传输层连接建立。截图中的 Frame 7686 和 7687 展示了 ARP 协议的工作过程：由于主机（192.168.100.1）与虚拟机服务器（192.168.100.9）处于同一网段，双方通过 ARP 广播（Broadcast）询问并获取了对方的 MAC 地址（物理地址），完成了 IP 地址到 MAC 地址的映射。随后，Frame 7685、7688 和 7689 展示了完整的 TCP 三次握手过程：客户端（192.168.100.1）首先向服务器 80 端口发送 SYN 同步包（Seq=0）；服务器收到后回复 SYN, ACK 包（Seq=0, Ack=1）；最后客户端回复 ACK 包（Seq=1, Ack=1）。至此，双方成功建立了可靠的 TCP 连接，为后续的 HTTP 数据传输奠定了基础。

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。）

暂无。

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。）

暂无。