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：计算机 1 班

学生学号：2023311126

学生姓名：邓铭轩

评阅教师：

报告成绩：

实验与创新实践教育中心制

2025 年 10 月

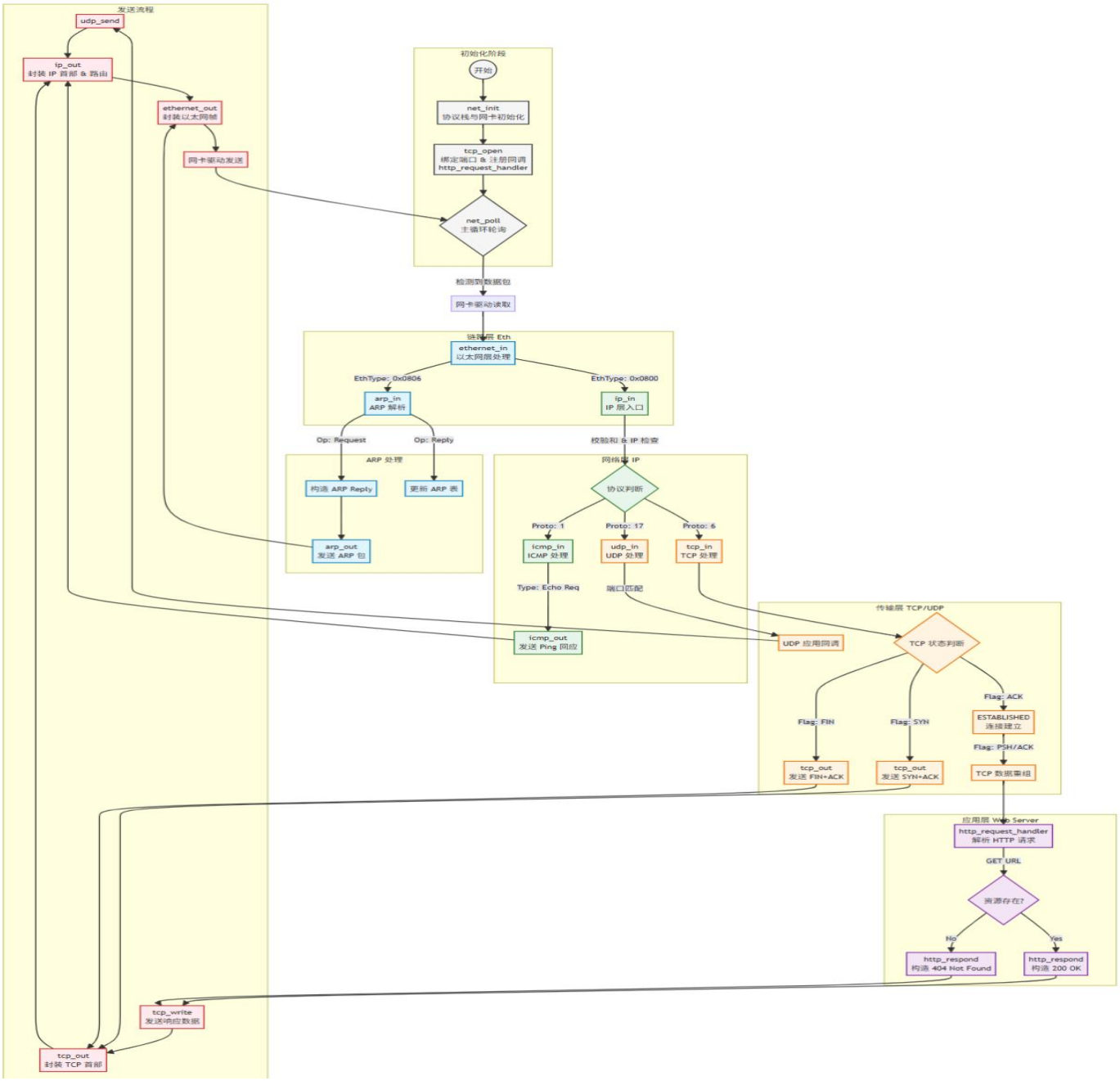
一、 协议实现详述

（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。）

协议栈实验的整体流程遵循分层封装与解封装逻辑，先通过 net_init()完成以太网、ARP、IP、TCP 各层协议初始化及虚拟网卡配置，再由 tcp_open(HTTP_LISTEN_PORT, http_request_handler)注册 80 端口的 HTTP 请求回调函数并进入 net_poll()循环监听网络事件；当客户端发起通信时，先通过 ARP 协议广播发送请求包，查询目标 IP 对应的 MAC 地址，目标设备收到后以单播形式返回 ARP 响应包完成 IP-MAC 映射，随后客户端与服务端通过 TCP 三次握手建立连接（客户端发送 SYN 报文、服务端回复 SYN+ACK 报文、客户端发送 ACK 报文），连接建立后客户端发送 HTTP GET 请求（经 TCP 分段、IP 封装、以太网帧封装后传输），服务端通过链路层接收以太网帧并剥离帧头提取 IP 数据包，再剥离 IP 头得到 TCP 段，经 TCP 层校验序列号、重组数据后传递至应用层，由 http_request_handler 解析 URL 路径，http_respond 函数根据资源存在性构造 200 或 404 HTTP 响应，响应数据经 TCP、IP、以太网逐层封装后回传客户端，客户端接收后逐层解封装获取响应内容，最终完成一次完整的网络通信。



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。）

以太网协议的数据封装与解封装过程围绕 ethernet_out（封装发送）和 ethernet_in（解封装接收）两大核心函数展开：在发送侧（ethernet_out）首先校验上层传入数据的长度，若不足以太网帧最小载荷长度 46 字节则填充 0 补足，随后为数据包头部预留 14 字节以太网帧头空间，依次填充目的 MAC 地址、本机源 MAC 地址及转换为网络字节序的上层协议类型（如 ARP/IP），完成封装后调用驱动层函数将完整以太网帧发送至物理网卡；在接收侧（ethernet_in），首先检查数据包长度是否包含完整以太网帧头，接着剥离 14 字节帧头，提取并转换协议类型为主机字节序，将剥离帧头后的上层数据调用 net_in 函数，完成解封装与协议分发。整个过程严格遵循以太网帧格式规范，封装时保证帧结构完整性（填充最小长度、字节序转换），解封装时通过 MAC 地址过滤保证数据有效性，实现了上层数据与物理层传输帧的双向转换。

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。）

本 ARP 协议实现围绕 IP 与 MAC 地址映射的核心需求，构建了完整的请求/响应处理、表项管理和数据包缓存逻辑：协议初始化阶段（arp_init）完成 ARP 表（arp_table，<IP，MAC>映射）和数据包缓存（arp_buf，<IP，buf_t>映射）的容器初始化，注册 ARP 协议处理函数并发送无回报 ARP 请求（目标 IP 为本机 IP）宣告本机 IP-MAC 映射；发送 IP 数据包时（arp_out）优先查询 ARP 表，若存在目标 IP 对应的 MAC 则直接通过以太网层发送，若不存在则检查 arp_buf，无缓存时缓存数据包并调用 arp_req 发送 ARP 请求（填充以太网硬件类型、IPv4 协议类型等固定字段，设置操作类型为 ARP_REQUEST，本机 MAC/IP、目标 IP（目标 MAC 置 0），以广播 MAC 发送），有缓存则避免重复请求；接收 ARP 报文时（arp_in）先校验报文长度和报头合法性（硬件类型、协议类型、地址长度、操作类型等），合法报文均通过 map_set 更新 ARP 表中发送方 IP 对应的 MAC 映射以保证表项最新，若 arp_buf 中存在该 IP 的缓存数据包则补发并删除缓存，若无缓存且报文为针对本机 IP 的 ARP_REQUEST，则调用 arp_resp 发送 ARP 响应（填充 ARP_REPLY 类型报头，本机 MAC/IP、请求方 IP/MAC，以单播 MAC 发送）；ARP 表项通过设置超时时间（ARP_TIMEOUT_SEC）管理有效性，arp_buf 通过最小请求间隔（ARP_MIN_INTERVAL）限制避免广播风暴，整体形成“查询-响应-更新-补发”的闭环，保障 IP 数据包基于 MAC 地址的可靠传输。

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。）

本 IP 协议实现围绕数据包封装与解封装、分片传输、校验和验证三大核心功能展开，构建了完整的 IPv4 报文处理链路：解封装流程由 ip_in()函数实现，接收数据包后先校验长度是否不小于 IP 头部最小长度，再验证版本号为 IPv4、头部长度合法、总长度字段有效，随后通过保存原始校验和、置零校验和字段、调用 checksum16()重新计算并对比的方式完成头部完整性校验，校验通过后对比目的 IP 是否为本机，再去除填充字段和 IP 头部，将上层载荷传递给 net_in()函数，若上层不识别协议则恢复 IP 头部并调用 icmp_unreachable()返回不可达信息；封装与分片流程由 ip_out()和 ip_fragment_out()函数协同实现，ip_out()先判断上层数据长度是否超过最大载荷（1480 字节），未超过则直接调用 ip_fragment_out()封装完整报文，超过则按最大载荷分片，所有分片共用同一个全局递增的 ip_id，前 N-1 个分片设置 MF 标志为 1，最后一个分片置 0，ip_fragment_out()负责为每个分片（或完整报文）添加 20 字节 IP 头部，按协议规范填写版本、头部长度、总长度、ID、分片偏移、MF 标志、TTL、上层协议、源目 IP 等字段，先将校验和字段置零，再调用 checksum16()计算头部校验和并填充，最后通过 arp_out()发送；校验和计算由 checksum16()函数实现，按 16 位分组累加数据，处理奇数长度的剩余字节，循环合并累加和的高 16 位与低 16 位直至高 16 位为 0，最终对结果取反得到校验和，用于 IP 头部的完整性验证，保证报文传输过程未被篡改。

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。）

本 ICMP 协议实现围绕回显请求 / 响应处理和不可达报文发送两大核心功能展开，构建了支持 ping 通信和网络故障诊断的完整逻辑：协议初始化时通过 icmp_init()注册 ICMP 协议处理函数，当 icmp_in()收到数据包后，先校验数据包长度是否不小于 ICMP 头部长度，再判断报文类型是否为回显请求，若是则调用 icmp_resp()封装回显响应报文，该函数会初始化全局发送缓冲区，拷贝请求报文的全部数据（保持标识符和序号不变），将报文类型改为回显响应并置零校验和字段，通过 checksum16()计算整个 ICMP 报文（头部 + 数据）的校验和后，调用 ip_out()发送至请求方 IP，实现 ping 应答功能；当网络层出现协议不可达或端口不可达错误时，icmp_unreachable()函数会按规范封装目的不可达报文，先初始化缓冲区并填写 ICMP 头部（类型为目的不可达、代码为对应错误类型），再将原始错误 IP 数据包的完整头部和载荷前 8 字节拷贝至 ICMP 数据部分，计算整个报文的校验和后调用 ip_out()发送至错误报文的源 IP，接收方通过解析该不可达报文的原始 IP 头和载荷信息，可定位到具体的故障数据包和故障类型（如未知上层协议、目标端口未监听），从而实现网络故障诊断的核心需求，整体逻辑既符合 ICMP 协议规范，又与 IP 层、ARP 层形成完整的协议栈联动。

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 协议的详细设计围绕数据包封装、解封装及校验和计算三大核心展开：封装过程在 udp_out 函数中实现，先为数据缓冲区添加 8 字节 UDP 头部空间，填充源/目的端口（网络字节序）、UDP 总长度（头部+数据，网络字节序）并将校验和字段置 0，随后调用 transport_checksum 构造包含源 IP、目的 IP、协议号（UDP 为 17）、UDP 长度的 12 字节伪头部，结合 UDP 头部与数据段通过 checksum16 函数按 16 位分组累加（奇数长度补 0，32 位累加避免溢出，循环处理高 16 位）并取反生成校验和，最后交由 IP 层封装发送；解封装过程在 udp_in 函数中完成，先校验数据包长度是否满足 UDP 头部最小要求、实际长度与声明长度的一致性，再保存原始校验和并置 0 后重新计算校验和，不匹配则丢弃数据包，匹配则根据目的端口查询 udp_table 获取处理函数，未找到则发送 ICMP 端口不可达报文，找到则移除 UDP 头部并传递数据段至处理函数；校验和计算由 transport_checksum 和 checksum16 协同完成，transport_checksum 负责伪头部的构造与缓冲区的临时调整，checksum16 实现 16 位校验和的核心累加与取反逻辑，且伪头部仅参与计算不随包传输，接收端通过相同流程验证校验和（结果为 0xFFFF 则通过），确保 UDP 数据包传输的完整性与正确性，同时 udp_init 完成协议注册与端口映射表初始化，udp_open/udp_close 实现端口注册注销，udp_send 为应用层提供发送接口，形成 UDP 协议的完整功能闭环。

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。）

TCP 协议的连接建立遵循三次握手流程：服务端先处于 LISTEN 状态监听端口，客户端发送 SYN 报文发起连接，服务端收到 SYN 后生成初始序列号（ISN），将确认号设为客户端 SYN 序列号+1，回复 SYN+ACK 报文并转入 SYN_RECEIVED 状态；客户端收到 SYN+ACK 后回复 ACK 报文，服务端收到该 ACK 后转入 ESTABLISHED

状态，连接正式建立。连接关闭则采用四次挥手流程：通信一方发送 FIN 报文请求关闭，另一方收到 FIN 后回复 ACK 确认，此时发送 FIN 方进入 FIN_WAIT 状态，接收 FIN 方进入 CLOSE_WAIT 状态；接收 FIN 方完成数据发送后也发送 FIN 报文，发起关闭方收到后回复 ACK 确认，最终双方完成连接关闭并释放资源。对于 TCP 数据包的确认，接收方会校验数据包的序列号是否为期望的确认号，若匹配则更新确认号为发送方序列号+数据长度（含 SYN/FIN 的 1 字节序列空间），并根据数据接收情况回复 ACK；若序列号不匹配则发送重复 ACK，确保数据有序接收。连接状态管理通过维护 TCP 连接表实现，表中存储连接的三元组（源 IP、源端口、目的端口）、当前状态（LISTEN/SYN_RECEIVED/ESTABLISHED 等）、序列号和确认号等关键信息，处理数据包时根据连接状态执行对应的逻辑分支，如 LISTEN 状态仅处理 SYN 报文，ESTABLISHED 状态处理数据传输、FIN 报文及序列号校验，同时通过状态转移函数严格管控连接在不同阶段的状态变迁，确保 TCP 协议的可靠性和规范性。

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。）

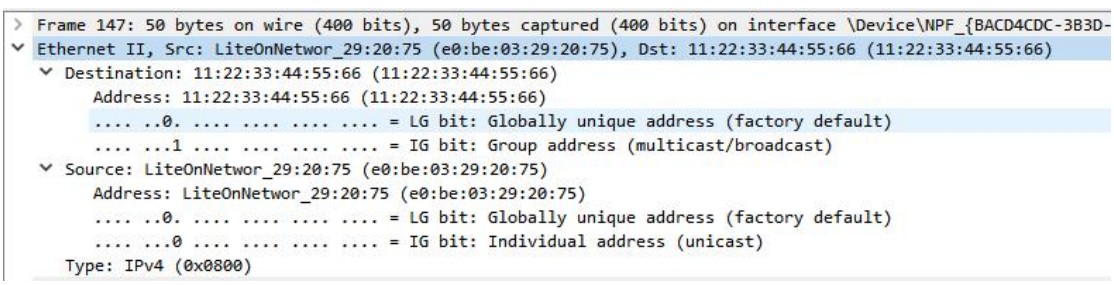
Web 服务器基于 TCP 协议栈构建，初始化时通过 net_init()启动网络协议栈，并注册 80 端口的 TCP 监听回调 http_request_handler，随后进入循环持续监听网络事件；当客户端通过 TCP 连接发送 HTTP 请求时，TCP 协议栈触发回调函数，先校验请求方法仅支持 GET，再从请求数据中提取 URL 路径（若路径为/则默认指向 index.html），接着调用 http_respond 构造响应：若拼接资源目录后的文件不存在，会发送包含状态行、连接保持、内容类型等头部的 404 响应及对应 HTML 页面；若文件存在，则发送 200 状态行，结合文件后缀通过 http_get_mime_type 设置对应 MIME 类型的 Content-Type 头部，计算文件长度填入 Content-Length，再通过空行分隔响应头与响应体，最后分块读取文件内容并通过 TCP 发送，完成资源的响应，同时通过 Keep-Alive 保持 TCP 连接以复用。

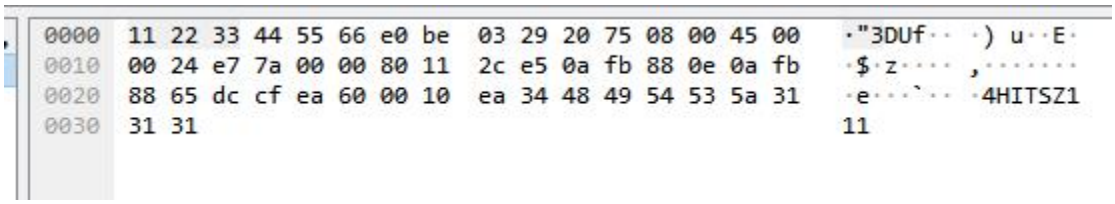
二、 实验结果截图及分析

（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）



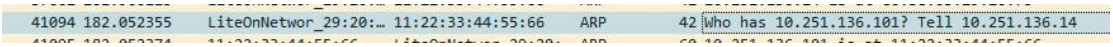


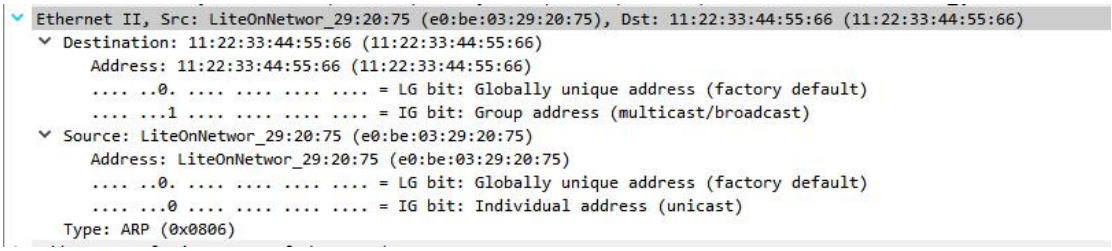
目的 MAC 为 11:22:33:44:55:66
源 MAC 为 LiteOnNetwor_29:20:75（对应实际地址 e0:be:03:29:20:75）
协议类型字段：协议类型为 IPv4（0x0800）末尾的
HITSZ111 是应用层数据（结合之前的测试场景，对应发送的测试字符串），数据部分从链路层到应用层的封装逻辑完整，无异常。

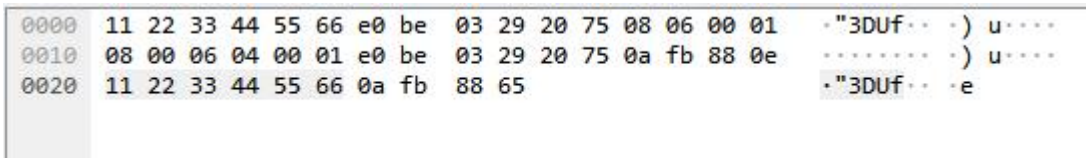
2. ARP 协议实验结果及分析

（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。）

请求包：

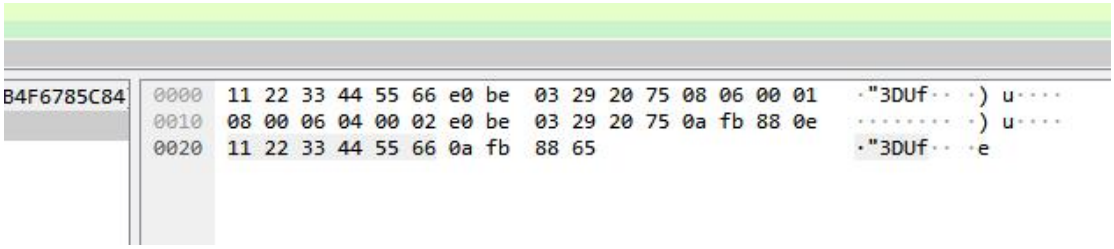






请求过程正常，目标 IP 地址：从抓包的“Info”（Who has 10.251.136.101? Tell 10.251.136.14）可知，ARP 请求的目标 IP 是 10.251.136.101，符合 ARP“查询目标 IP 对应 MAC”的请求逻辑，发送方 MAC 地址：以太网帧的源 MAC 是 LiteOnNetwor_29:20:75（对应实际地址 e0:be:03:29:20:75），属于合法的单播 MAC 地址，且与发送方设备（IP 为 10.251.136.14）的 MAC 一致，符合 ARP 请求中“携带发送方自身 MAC/IP”的规范。

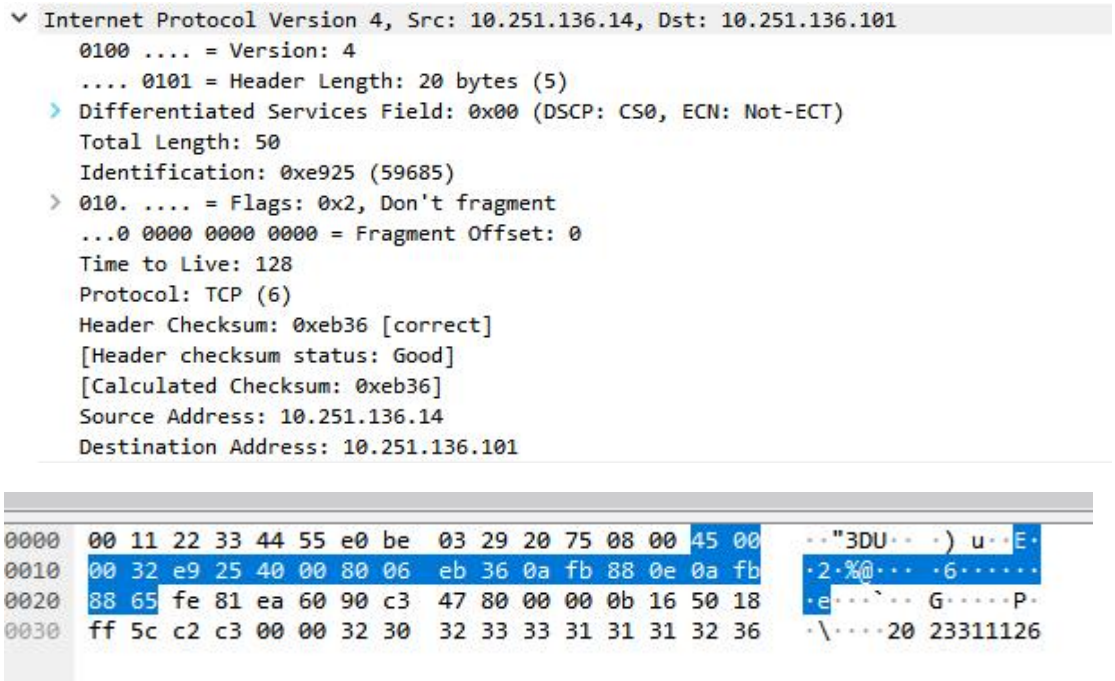
相应包：



ARP 响应的过程正常
响应的源 MAC 是 LiteOnNetwor_29:20:75(对应 e0:be:03:29:20:75),Wireshark 标注为 “Globally unique address” “Individual address (unicast)”，是合法的单播 MAC 地址

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。）

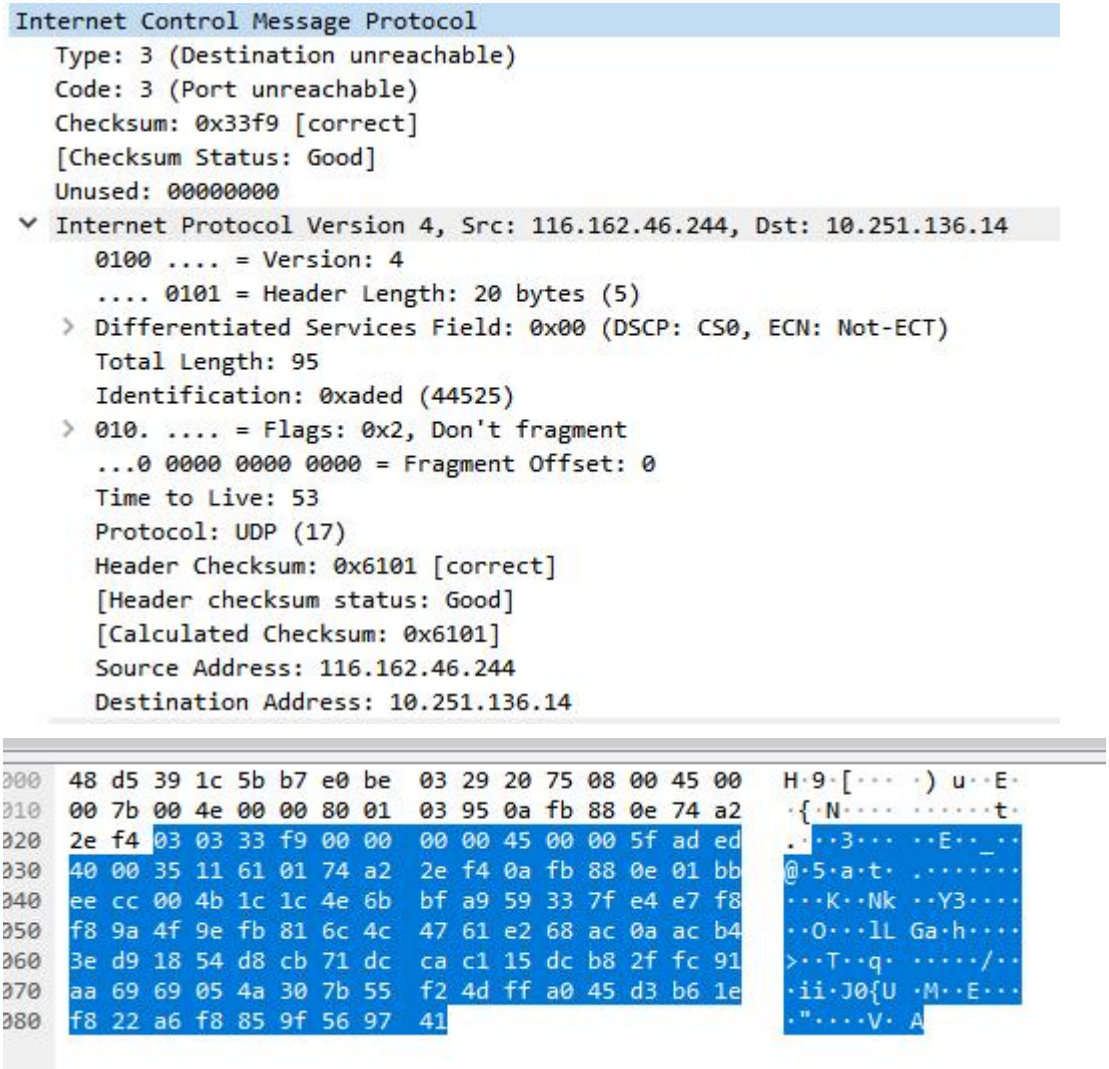


版本号（Version）：字段值为 4，对应 IPv4 协议，符合当前网络主流协议版本，正确。
首部长度（Header Length）：值为 20 bytes（二进制 0101），是 IPv4 头的标准最小长度（无选项 / 填充字段），正确。
总长度（Total Length）：值为 50，表示整个 IP 数据报（IP 头 + 数据）长度为 50 字节（IP 头 20 字节 + 后续数据 30 字节，符合 20+30=50 的计算逻辑），正确。
标识（Identification）：值为 0xe925（十进制 59685），是 IP 数据报的唯一标识，用于分片重组，格式合法，正确。

标志位（Flags）：值为 0x2（二进制 010），其中 “Don't Fragment（禁止分片）” 位为 1，符合当前无需分片的通信场景，正确。
片偏移（Fragment Offset）：值为 0，结合 “禁止分片” 标志位，说明该 IP 数据报是完整包，未被分片，正确。
TL（Time to Live）：值为 128，是 Windows 系统默认的 TTL 初始值，用于防止数据包循环转发，正确。
协议类型（Protocol）：值为 TCP（6），表示 IP 数据报内部封装的是 TCP 协议，与后续解析的 TCP 段一致，正确。
综上，该 IP 头部的所有字段均符合 IPv4 协议规范，无格式或逻辑错误

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。）



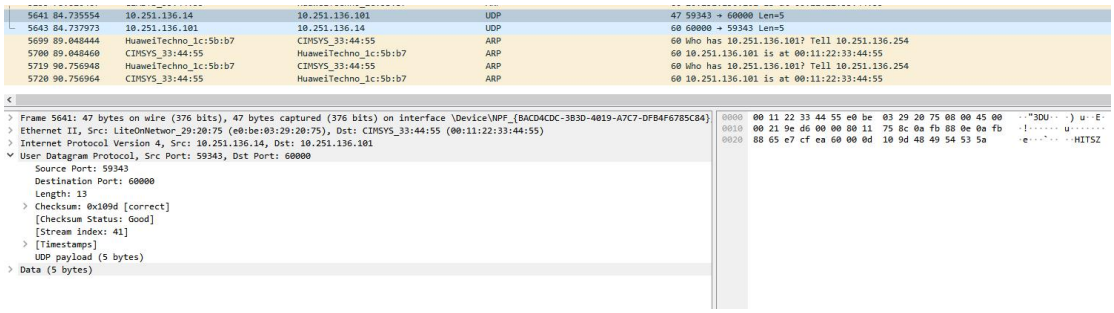
ICMP 类型（Type）：值为 3，表示“目标不可达”，符合 ICMP 协议规范。

代码（Code）：值为 3，表示“端口不可达”（细化了“目标不可达 的原因，即目的 IP 可达，但指定端口未开放），字段含义准确。
校验和（Checksum）：值为 0x33f9，状态显示 “Correct”，说明 ICMP 报文的完整性校验通过，字段合法。
2. 报文携带的信息
该 ICMP 差错报文携带了出错的原始 IP 数据报信息（ICMP 差错报文的标准格式）：
内嵌的原始 IP 头部显示:源 IP 是 116.162.46.244、目的 IP 是 10.251.136.14，协议类型为 UDP（17）（说明原始报文是 UDP 包）；
这些信息用于告知源端 “之前发送的 UDP 报文（目标端口）无法到达”，帮助源端定位通信故障。

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。）

wireShark 捕获的 UDP 数据包：

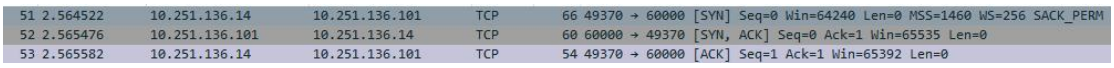


UDP 头部字段解析
源端口：59343（调试工具的发送端口）
目的端口：60000（协议栈的监听端口）
长度：13（UDP 头部 8 字节 + 载荷 5 字节，与实际数据量匹配）
校验和：0x109d [correct]（校验和状态为 “Good”，验证通过）
载荷数据：5 bytes（HITSZ）

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。）

TCP 连接建立过程



```
Transmission Control Protocol, Src Port: 49370, Dst Port: 60000, Seq: 0, Len: 0
    Source Port: 49370
    Destination Port: 60000
    [Stream index: 6]
  > [Conversation completeness: Incomplete, ESTABLISHED (7)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 1218435183
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x002 (SYN)
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0x816b [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NO
  > [Timestamps]
```

```
Transmission Control Protocol, Src Port: 60000, Dst Port: 49370, Seq: 0, Ack: 1, Len: 0
    Source Port: 60000
    Destination Port: 49370
    [Stream index: 6]
  > [Conversation completeness: Incomplete, ESTABLISHED (7)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 23438
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 1218435184
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x012 (SYN, ACK)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0x618f [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
```

```
Transmission Control Protocol, Src Port: 49370, Dst Port: 60000, Seq: 1, Ack: 1, Len: 0
    Source Port: 49370
    Destination Port: 60000
    [Stream index: 6]
  > [Conversation completeness: Incomplete, ESTABLISHED (7)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 1218435184
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 23439
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x010 (ACK)
    Window: 65392
    [Calculated window size: 65392]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0x621f [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
```

第一次握手（客户端→服务端）

源端口号：49370（客户端端口）
目的端口号：60000（服务端监听端口）
序列号（Sequence Number）：相对序号 0（原始序号 1218435183）→ 客户端初始序号
确认号（Acknowledgment number）：0→ 尚未确认任何数据
标志位（Flags）：0x002 (SYN)→ 仅 SYN 置 1，发起连接请求
其他字段：窗口大小 64240，携带 MSS 等选项
第二次握手（服务端→客户端）
源端口号：60000（服务端端口）
目的端口号：49370（客户端端口）
序列号（Sequence Number）：相对序号 0（原始序号 23438）→ 服务端初始序号
确认号（Acknowledgment number）：1→ 对应客户端第一次握手的 Seq=0+1，确认客户端请求
标志位（Flags）：0x012 (SYN, ACK)→ SYN（同步自身序号）+ACK（确认客户端请求）均置 1
其他字段：窗口大小 65535
第三次握手（客户端→服务端）
源端口号：49370（客户端端口）
目的端口号：60000（服务端端口）
序列号（Sequence Number）：相对序号 1（原始序号 1218435184）→ 客户端初始序号的下一个值
确认号（Acknowledgment number）：1→ 对应服务端第二次握手的 Seq=0+1，确认服务端响应
标志位（Flags）：0x010 (ACK)→ 仅 ACK 置 1，完成最终确认
其他字段：窗口大小 65392
二、连接状态转换
TCP 连接的状态随三次握手逐步变化：
第一次握手前：服务端处于 LISTEN（监听）状态，等待连接请求；客户端处于 CLOSED（关闭）状态。
第一次握手后：客户端发送 SYN，进入 SYN_SENT（同步已发送）状态。
第二次握手后：服务端收到 SYN 并回复 SYN+ACK，进入 SYN_RCVD（同步已接收）状态。
第三次握手后：客户端回复 ACK，双方均进入 ESTABLISHED（已建立）状态 —— 此时连接正式建立，可开始传输数据。

数据传输：

14988	351.166206	10.251.136.101	10.251.136.14	TCP	64	60000 → 49370 [ACK] Seq=1 Ack=11 Win=65535 Len=10
14989	351.206941	10.251.136.14	10.251.136.101	TCP	54	49370 → 60000 [ACK] Seq=11 Ack=11 Win=65382 Len=0

```
Transmission Control Protocol, Src Port: 60000, Dst Port: 49370, Seq: 1, Ack: 11, Len: 10
   Source Port: 60000
   Destination Port: 49370
   [Stream index: 6]
 > [Conversation completeness: Incomplete, DATA (15)]
   [TCP Segment Len: 10]
   Sequence Number: 1    (relative sequence number)
   Sequence Number (raw): 23439
   [Next Sequence Number: 11    (relative sequence number)]
   Acknowledgment Number: 11    (relative ack number)
   Acknowledgment number (raw): 1218435194
   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: 0x6680 [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > [Timestamps]
 > [SEQ/ACK analysis]
   TCP payload (10 bytes)
Data (10 bytes)
```

```
0000  e0 be 03 29 20 75 00 11  22 33 44 55 08 00 45 00   ···) u·· "3DU··E·
0010  00 32 00 cf 00 00 40 06  53 8e 0a fb 88 65 0a fb   ·2····@· S····e··
0020  88 0e ea 60 c0 da 00 00  5b 8f 48 9f d8 7a 50 10   ···`···· [·H··zP·
0030  ff ff 66 80 00 00 32 30  32 33 33 31 31 31 32 36   ··f···20 23311126
```

```
Transmission Control Protocol, Src Port: 49370, Dst Port: 60000, Seq: 11, Ack: 11, Len: 0
   Source Port: 49370
   Destination Port: 60000
   [Stream index: 6]
 > [Conversation completeness: Incomplete, DATA (15)]
   [TCP Segment Len: 0]
   Sequence Number: 11    (relative sequence number)
   Sequence Number (raw): 1218435194
   [Next Sequence Number: 11    (relative sequence number)]
   Acknowledgment Number: 11    (relative ack number)
   Acknowledgment number (raw): 23449
   0101 .... = Header Length: 20 bytes (5)
 > Flags: 0x010 (ACK)
   Window: 65382
   [Calculated window size: 65382]
   [Window size scaling factor: -2 (no window scaling used)]
   Checksum: 0x6215 [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > [Timestamps]
 > [SEQ/ACK analysis]
```

```
0000  00 11 22 33 44 55 e0 be  03 29 20 75 08 00 45 00   ··"3DU·· ·) u··E·
0010  00 28 ea 8f 40 00 80 06  e9 d6 0a fb 88 0e 0a fb   ·(··@··· ········
0020  88 65 c0 da ea 60 48 9f  d8 7a 00 00 5b 99 50 10   ·e···`H· ·z··[·P·
0030  ff 66 62 15 00 00                                 ·fb···
```

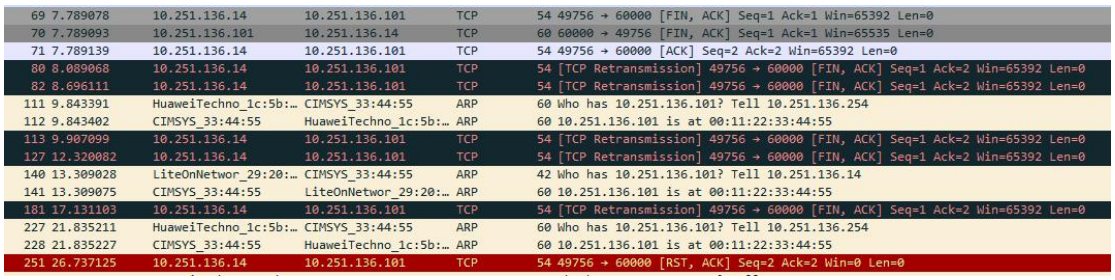
这组报文是 TCP 连接建立后（ESTABLISHED 状态）的双向数据交互，字段分析如下：

服务端→客户端的带数据报文

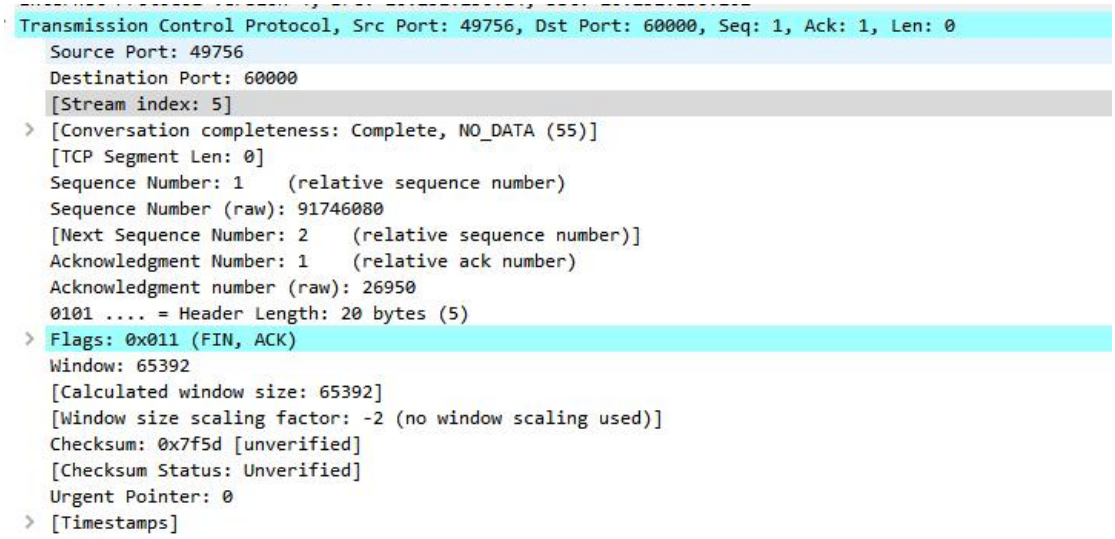
源 / 目的端口：源端口 60000（服务端），目的端口 49370（客户端）

序列号（Sequence Number）：相对序号 1（原始序号 23439）→ 基于服务端

初始序号 0 的下一个值
确认号（Acknowledgment number）：相对序号 11→ 确认客户端已发送到序号 10 的数据
标志位（Flags）：0x010 （ACK）→ 仅 ACK 置 1，同时携带数据（Len=10）
数据部分：携带 10 字节的 TCP payload（实际数据）
客户端→服务端的确认报文
源 / 目的端口：源端口 49370（客户端），目的端口 60000（服务端）
序列号（Sequence Number）：相对序号 11→ 对应客户端已发送到序号 10 的下一个值
确认号（Acknowledgment number）：相对序号 11→ 确认服务端已发送到序号 10 的数据
标志位（Flags）：0x010 （ACK）→ 仅 ACK 置 1，无数据（Len=0）
关闭过程：



69	7.789078	10.251.136.14	10.251.136.101	TCP	54 49756 → 60000 [FIN, ACK] Seq=1 Ack=1 Win=65392 Len=0
70	7.789093	10.251.136.101	10.251.136.14	TCP	60 60000 → 49756 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
71	7.789139	10.251.136.14	10.251.136.101	TCP	54 49756 → 60000 [ACK] Seq=2 Ack=2 Win=65392 Len=0
80	8.089068	10.251.136.14	10.251.136.101	TCP	54 [TCP Retransmission] 49756 → 60000 [FIN, ACK] Seq=1 Ack=2 Win=65392 Len=0
82	8.696111	10.251.136.14	10.251.136.101	TCP	54 [TCP Retransmission] 49756 → 60000 [FIN, ACK] Seq=1 Ack=2 Win=65392 Len=0
111	9.843391	HuaweiTechno_1c:5b:…	CIMSYS_33:44:55	ARP	60 Who has 10.251.136.101? Tell 10.251.136.254
112	9.843402	CIMSYS_33:44:55	HuaweiTechno_1c:5b:…	ARP	60 10.251.136.101 is at 00:11:22:33:44:55
113	9.907099	10.251.136.14	10.251.136.101	TCP	54 [TCP Retransmission] 49756 → 60000 [FIN, ACK] Seq=1 Ack=2 Win=65392 Len=0
127	12.320082	10.251.136.14	10.251.136.101	TCP	54 [TCP Retransmission] 49756 → 60000 [FIN, ACK] Seq=1 Ack=2 Win=65392 Len=0
140	13.309028	LiteOnNetwor_29:20:…	CIMSYS_33:44:55	ARP	42 Who has 10.251.136.101? Tell 10.251.136.14
141	13.309075	CIMSYS_33:44:55	LiteOnNetwor_29:20:…	ARP	60 10.251.136.101 is at 00:11:22:33:44:55
181	17.131103	10.251.136.14	10.251.136.101	TCP	54 [TCP Retransmission] 49756 → 60000 [FIN, ACK] Seq=1 Ack=2 Win=65392 Len=0
227	21.835211	HuaweiTechno_1c:5b:…	CIMSYS_33:44:55	ARP	60 Who has 10.251.136.101? Tell 10.251.136.254
228	21.835227	CIMSYS_33:44:55	HuaweiTechno_1c:5b:…	ARP	60 10.251.136.101 is at 00:11:22:33:44:55
251	26.737125	10.251.136.14	10.251.136.101	TCP	54 49756 → 60000 [RST, ACK] Seq=2 Ack=2 Win=0 Len=0



```
Transmission Control Protocol, Src Port: 49756, Dst Port: 60000, Seq: 1, Ack: 1, Len: 0
    Source Port: 49756
    Destination Port: 60000
    [Stream index: 5]
  > [Conversation completeness: Complete, NO_DATA (55)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 91746080
    [Next Sequence Number: 2    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 26950
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x011 (FIN, ACK)
    Window: 65392
    [Calculated window size: 65392]
    [Window size scaling factor: -2 (no window scaling used)]
    Checksum: 0x7f5d [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > [Timestamps]
```

```
Transmission Control Protocol, Src Port: 60000, Dst Port: 49756, Seq: 1, Ack: 1, Len: 0
   Source Port: 60000
   Destination Port: 49756
   [Stream index: 5]
 > [Conversation completeness: Complete, NO_DATA (55)]
   [TCP Segment Len: 0]
   Sequence Number: 1    (relative sequence number)
   Sequence Number (raw): 26950
   [Next Sequence Number: 2    (relative sequence number)]
   Acknowledgment Number: 1    (relative ack number)
   Acknowledgment number (raw): 91746080
   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: 0x7ece [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > [Timestamps]
```

```
Transmission Control Protocol, Src Port: 49756, Dst Port: 60000, Seq: 2, Ack: 2, Len: 0
   Source Port: 49756
   Destination Port: 60000
   [Stream index: 5]
 > [Conversation completeness: Complete, NO_DATA (55)]
   [TCP Segment Len: 0]
   Sequence Number: 2    (relative sequence number)
   Sequence Number (raw): 91746081
   [Next Sequence Number: 2    (relative sequence number)]
   Acknowledgment Number: 2    (relative ack number)
   Acknowledgment number (raw): 26951
   0101 .... = Header Length: 20 bytes (5)
 > Flags: 0x010 (ACK)
   Window: 65392
   [Calculated window size: 65392]
   [Window size scaling factor: -2 (no window scaling used)]
   Checksum: 0x7f5c [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > [Timestamps]
 > [SEQ/ACK analysis]
```

TCP 头部字段检查（对应关闭步骤）

第一次挥手：主动方发起关闭请求

源 / 目的端口：源端口 49756（主动关闭方）→ 目的端口 60000（被动关闭方）

序列号（Sequence Number）：相对序号 1（原始序号 91746080）→ 基于连接中的序号延续

确认号（Acknowledgment number）：相对序号 1→ 确认被动方之前的数据流

标志位（Flags）：0x011 (FIN, ACK)→ 同时携带 FIN（发起关闭）和 ACK（确认数据）

第二次 + 第三次挥手：被动方合并 “确认 + 发起关闭”

源 / 目的端口：源端口 60000（被动关闭方）→ 目的端口 49756（主动关闭方）

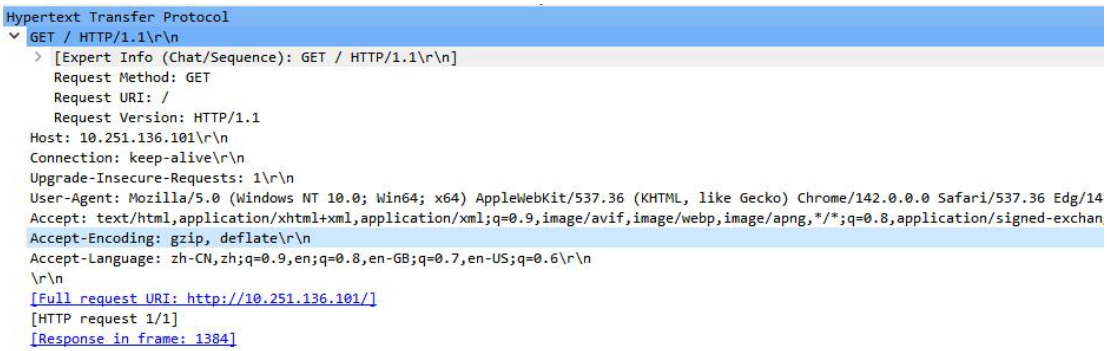
序列号（Sequence Number）：相对序号 1（原始序号 26950）→ 基于被动方

的序号延续
确认号（Acknowledgment number）：相对序号 1→ 确认主动方的 FIN（对应主动方 Seq=1）
标志位（Flags）：0x011（FIN, ACK）→ 被动方将 “确认主动方 FIN 的 ACK” 和 “自身发起的 FIN” 合并发送（这是 TCP 的优化机制，减少报文数量）
第四次挥手：主动方确认被动方的关闭请求
源 / 目的端口：源端口 49756（主动关闭方）→ 目的端口 60000（被动关闭方）
序列号（Sequence Number）：相对序号 2→ 对应主动方 FIN 的 Seq=1+1
确认号（Acknowledgment number）：相对序号 2→ 确认被动方的 FIN（对应被动方 Seq=1+1）
标志位（Flags）：0x010（ACK）→ 仅携带 ACK，完成最终确认
二、连接状态转换（对应 “合并版挥手”）
第一次挥手后：主动方（49756）发送 FIN，进入 FIN_WAIT_1 状态；
第二次 + 第三次挥手后：被动方（60000）回复 FIN+ACK，主动方收到后进入 FIN_WAIT_2 状态；被动方进入 LAST_ACK 状态；
第四次挥手后：主动方发送 ACK，进入 TIME_WAIT 状态（等待 2MSL 确保连接彻底关闭）；被动方收到 ACK 后进入 CLOSED 状态。
为什么看起来 “少了一条 FIN”？因为被动方把 “确认 ACK” 和 “自身 FIN” 合并成了一条 FIN+ACK 报文（对应步骤 2），这是 TCP 的常见优化

7. web 服务器实验结果及分析

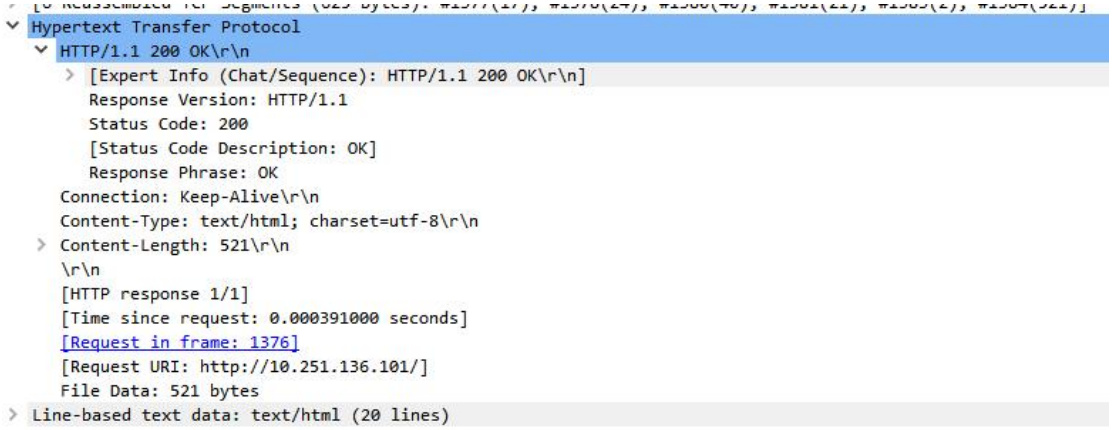
（展示 web 服务器的请求和响应过程截图，分析 HTTP 请求和响应的格式、内容。检查请求方法、请求 URL、请求头、响应状态码、响应头、响应体等部分。）

请求：



请求行（请求的核心标识）
请求方法：GET（获取资源的请求方法）
请求 URL：/（请求服务端的根路径资源）
请求版本：HTTP/1.1（使用 HTTP 1.1 协议版本）
请求头（请求的附加参数）
包含多组 “键：值” 格式的字段，用于告知服务端请求的细节：

Host: 10.251.136.101：指定请求的目标服务器地址；
Connection: keep-alive：要求保持长连接，避免频繁建立 / 关闭连接；
Upgrade-Insecure-Requests: 1：请求将 HTTP 连接升级为 HTTPS（更安全的加密连接）；
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)...：标识发起请求的客户端（这里是 Chrome 浏览器）；
Accept: text/html,application/xml...：告知服务端客户端能接收的内容类型；
Accept-Encoding: gzip, deflate：支持的压缩编码格式；
Accept-Language: zh-CN,zh;q=0.9...：客户端偏好的语言（中文优先）。
3. 其他信息
[Full request URI: http://10.251.136.101/]：完整的请求 URI（包含协议和主机）；
[HTTP request 1/1]：表示这是本次请求的唯一报文（HTTP 1.1 通常单报文完成请求）；
[Response in frame: 1384]：说明对应的响应报文在帧 1384 中）
相应：



关联的 HTTP 请求信息（从响应中反向获取）
请求方法：隐含为 GET（通常根路径请求用 GET 方法）
请求 URL：http://10.251.136.101/（对应 Request URI 字段）
HTTP 响应的格式与内容
1. 状态行（响应的核心标识）
响应版本：HTTP/1.1
响应状态码：200（表示请求成功）
状态码描述：OK
2. 响应头（响应的附加参数）
Connection: Keep-Alive：保持长连接，不立即关闭
Content-Type: text/html; charset=utf-8：响应内容是 HTML 文本，编码为 UTF-8
Content-Length: 521：响应体的大小为 521 字节
3. 响应体
类型：text/html（HTML 文本）
大小：521 字节（对应 Content-Length）

说明：当前截图显示 “Line-based text data: text/html (20 lines)”，表示响应体是 20 行的 HTML 内容（具体内容未完全展示）

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。）

在实验中遇到的困难主要是由于实验环境配置导致的。
比如在 udp server 检测中，发现在客户端发送报文，服务器一直没收到，后面通过排查发现是校验和的问题，然后再常见问题中找到了解决方法
还有 tcp server 检测中，也是一样的问题，导致客户端只有一次挥手，和服务器没法连接，后面发现也是和之前一样的校验和问题，一样的解决方式

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。）

通过本次实验，我深刻了解了计算机网络中各个协议栈的功能和实现方式，收获颇丰，同时也深感实验环境难配，出现了问题很难排查。
建议：可以把 windows 中需要关闭校验和自动填充 0 这个功能写到指导书中，不然还挺难排查的。