



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: 计算机 7 班
学生学号: 220110720
学生姓名: 刘睿
评阅教师:
报告成绩:

实验与创新实践教育中心制

2025 年 3 月

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 (Ethernet, 简称 Eth) 位于 TCP/IP 协议栈的数据链路层, 主要负责在局域网中完成数据帧的封装与解封装, 实现主机间的基本通信。在本实验中, 我们主要通过实现 `ethernet_out()` 和 `ethernet_in()` 函数来完成以太网协议的核心功能。

在发送数据时, 首先需要判断数据长度是否满足以太网帧的最小长度要求 (46 字节)。若不足, 需要使用 `buf_add_padding()` 函数对数据进行补齐。接着调用 `buf_add_header()` 为数据预留头部空间, 并填写以太网帧的各字段内容, 包括目的 MAC 地址、源 MAC 地址 (由系统变量 `net_if_mac` 提供) 和上层协议类型 (如 IPv4 协议为 0x0800)。最后, 调用 `driver_send()` 函数将封装后的数据帧通过驱动接口发送出去。

在接收数据时, 首先对接收到的数据帧进行长度校验, 若小于以太网头部长度的 (14 字节), 则说明帧不完整, 应直接丢弃。之后, 使用 `buf_remove_header()` 去除以太网帧头, 仅保留有效负载。最后, 根据协议类型字段调用 `net_in()` 函数, 将数据交由相应的上层协议 (如 ARP、IP 等) 进行处理。

以太网帧的结构包括目的地址 (6 字节)、源地址 (6 字节)、协议类型字段 (2 字节) 和有效数据部分 (46~1500 字节), 其结构简洁、开销小, 适合在局域网中高效传输。在实验中通过实际编码实现该协议的封装与解封装, 有助于加深对数据链路层通信机制的理解。

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。)

(1) ARP 请求处理逻辑

当需要向某一 IP 地址发送数据包时, 若本地 ARP 表中尚未记录该 IP 对应的 MAC 地址, 系统将构造并发送一份 ARP 请求报文。该报文采用广播方式 (目的 MAC 地址为 FF:FF:FF:FF:FF:FF), 在报文中填写目标 IP 地址, 并将目标 MAC 置为空 (全 0)。

发送流程如下: 首先检查 ARP 表中是否已有对应记录, 若无, 则将来自 IP 层待发送的数据包缓存; 随后构造 ARP 请求头部, 设置操作类型为 ARP_REQUEST; 最后调用 `ethernet_out()` 函数将该 ARP 请求报文广播发送至局域网。

(2) ARP 响应处理逻辑

收到 ARP 报文后, 若其为 ARP 响应类型, 且目标 IP 为本机地址, 则提取源 IP 和源 MAC 地址, 并调用 `map_set()` 函数更新 ARP 表; 接着检查是否缓存了发往该 IP 地址的数据包, 若有缓存, 则使用更新后的 MAC 地址将数据包通过 `ethernet_out()` 发送出去, 最后清除缓存项。

若收到的是 ARP 请求报文, 且请求目标 IP 与本机 IP 一致, 则构造一个 ARP 响应报文, 填入本机的 MAC 地址和 IP 地址, 并调用 `ethernet_out()` 将响应报文单播返回给请求源。

(3) ARP 表更新机制

实验中实现了一个支持超时机制的动态 ARP 表，使用 `map_t` 数据结构管理 IP 到 MAC 的映射关系。当收到 ARP 报文时，调用 `map_set()` 函数将新的映射写入表中，同时记录当前时间戳。查询时会调用 `map_entry_valid()` 函数判断表项是否仍然有效，若超过设定的超时时间 `ARP_TIMEOUT_SEC`，则该表项视为过期并删除。

通过这一机制，系统能够动态维护 ARP 映射表项的有效性，避免因 MAC 地址变化导致的通信错误。

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。)

1. IP 数据包的封装（发送端处理）

在发送数据时，IP 层负责将上层传下来的数据封装成 IP 数据报，主要步骤如下：

判断数据长度：如果数据长度大于最大传输单元（MTU 减去 IP 头部长度），则需要进行 IP 分片；

构造 IP 报文头部，设置版本、首部长度、标识、标志、偏移量、TTL、协议类型等字段；

计算校验和：对 IP 报头执行 16 位反码求和，填入校验和字段；

调用 ARP 模块发送：通过 `arp_out()` 函数将完整的 IP 报文发送到链路层。

2. IP 数据包的解封装（接收端处理）

接收到 IP 报文后，IP 层进行以下处理：

校验长度与版本号：确保为 IPv4，长度合理；

验证校验和：对接收到的 IP 报头重新计算校验和并比较；

验证目的地址：仅处理目标 IP 为本机的数据报；

移除填充和 IP 报头；

向上层传递数据：调用 `net_in()` 将数据传递给 TCP、UDP 等上层协议。

若无法识别协议类型，则调用 `icmp_unreachable()` 返回错误信息（该函数暂为空实现）。

3. IP 数据包分片处理

当 IP 数据报长度超出 MTU 限制时，需执行分片处理：

每片的最大数据长度为 MTU 减去 IP 头部长度，且必须是 8 字节的整数倍；

除最后一片外，其“更多分片（MF）”标志需置为 1；

所有分片的标识字段一致，用于后续在接收端重组；

每片设置正确的偏移值，用于表示其在原始数据报中的相对位置。

分片由 `ip_out()` 判断并分派给 `ip_fragment_out()` 实现。

4. 校验和计算

IP 校验和采用 16 位反码求和算法，仅作用于 IP 报头：

将报头按 16 位为单位分组相加；

如果有进位则加回最低位；

最后对结果取反，作为校验和。

校验函数在 `utils.c` 中实现，函数名为 `checksum16()`。

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。)

ICMP 请求与响应的处理流程

ICMP 报文主要分为两类：查询报文（如 Ping 请求和响应）和差错报文（如端口不可达）。本实验重点处理回显类 ICMP 报文。

1. 回显请求（Echo Request）的处理：

当接收到 IP 层传递上来的 ICMP 数据包时，首先需要检查报文的长度是否合法。如果报文的类型字段为 8（表示 Echo Request），则说明该报文是一个回显请求。程序应提取出请求报文中的标识符、序列号和数据部分，并将其原样复制，用于构造一个类型为 0（Echo Reply）的回显应答报文。然后调用发送函数，将构造好的应答报文发送回原始请求主机。

此流程通常由 `icmp_in()` 函数检测类型后，调用 `icmp_resp()` 函数实现。

2. 差错报文的构造（如协议不可达或端口不可达）：

当本机收到一个无法处理的 IP 数据包（例如，目标端口没有监听，或指定协议不支持），应调用 `icmp_unreachable()` 构造一个类型为 3 的 ICMP 差错报文。其中 `code` 字段值为 2 表示协议不可达，值为 3 表示端口不可达。数据部分需包含原始 IP 报文首部及其数据部分前 8 字节。构造完成后，通过 `ip_out()` 发送该差错报文。

ICMP 报文在网络故障诊断中的应用

ICMP 协议在网络运维与故障排查中具有重要作用，主要体现在以下几个方面：

1. ping 命令：

ping 使用 ICMP Echo Request 和 Echo Reply 报文来检测网络连通性。通过测量请求与应答之间的时间差可以估计主机间的延迟，判断目标主机是否可达，是最常用的连通性测试工具。

2. tracert 或 traceroute 命令：

该命令发送一系列具有递增 TTL 值的数据包。当数据包在传输途中 TTL 变为 0 时，中间路由器返回 ICMP 类型为 11 的“TTL 超时”报文。利用这一机制，可以依次探测出数据包经过的路由路径，用于分析网络路径及排查中间节点问题。

3. 差错报文提示网络异常：

若目标主机无法到达，或者传输过程中发生错误（如端口不可达），ICMP 差错报文能够及时通知发送端，使其做出相应处理，增强网络的可管理性与健壮性。

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 数据包封装过程中,首先需要构造 UDP 头部,包含源端口、目的端口、长度和校验和字段。然后将要发送的数据添加在头部后面,组成完整的 UDP 报文。报文的长度字段为头部和数据之和。接着,根据 IP 和 UDP 协议的要求,对报文计算校验和,最后将封装好的报文交由 IP 层发送。

在接收 UDP 报文时,首先检查报文长度是否小于 UDP 头部长度,若是则丢弃。接着解析头部字段,提取出源端口、目的端口和长度等信息。再对整个 UDP 报文进行校验和验证,如校验失败则丢弃该报文。若校验通过,将数据部分交由上层对应端口处理。

UDP 校验和采用与 IP 相同的算法。校验计算时,需要构造一个伪首部,伪首部包括源 IP、目的 IP、协议号和 UDP 长度。伪首部、UDP 头部和数据共同参与校验和的计算。接收端接收到 UDP 报文后,将使用相同方式验证校验和,以保证数据的完整性和正确性。

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。)

1. TCP 连接建立(三次握手)

TCP 的连接建立通过三次握手完成:

- (1) 客户端向服务端发送 SYN 报文,表示希望建立连接,携带初始序列号 $\text{seq} = x$;
- (2) 服务端收到 SYN 后,回复 SYN + ACK 报文,表示同意连接,并发送 $\text{seq} = y$, $\text{ack} = x+1$;
- (3) 客户端收到 SYN + ACK 后,再次发送 ACK 报文, $\text{seq} = x+1$, $\text{ack} = y+1$, 连接正式建立,双方进入 ESTABLISHED 状态。

在代码中,通过维护 `tcp_conn_t` 结构体中的 `state` 字段来跟踪连接状态,从 LISTEN 状态逐步过渡到 ESTABLISHED。

2. TCP 连接关闭(四次挥手)

连接断开通过四次挥手完成:

- (1) 客户端发送 FIN 报文,请求关闭连接;
- (2) 服务端收到后回复 ACK 报文,并进入 CLOSE_WAIT 状态;
- (3) 服务端准备好关闭后发送 FIN 报文;
- (4) 客户端收到 FIN 后回复 ACK 报文,进入 TIME_WAIT 状态,等待 2 倍 MSL 时间后关闭连接。

在实验中,服务端可被动响应 FIN 报文,状态机会依次从 CLOSE_WAIT 过渡到

LAST_ACK，最终转为 CLOSED 状态。

3. 数据确认机制

每个 TCP 报文都包含序列号 (seq) 和确认号 (ack):

序列号 seq 表示该报文中第一个字节的编号;

确认号 ack 表示期望接收的下一个字节编号。

接收方在收到有效数据后，会通过带有 ACK 标志位的报文进行确认。确认采用累计确认机制，即确认号代表之前所有字节已正确收到。

在 tcp_in() 函数中接收报文后，会根据 seq 和 ack 字段更新连接上下文的 tcp_conn_t.seq 和 tcp_conn_t.ack，并调用 tcp_out() 回复 ACK 报文。

4. 状态机管理

TCP 连接使用状态机管理连接生命周期，状态包括 LISTEN、SYN_RCVD、ESTABLISHED、FIN_WAIT 等。在 tcp.c 文件中通过对输入报文标志 (如 SYN、ACK、FIN) 的解析，进行对应的状态转换。

同时，所有连接记录保存在 tcp_conn_table 哈希表中，其键为 [源 IP, 源端口, 目标端口] 三元组，对应值为一个 tcp_conn_t 结构体，表示该连接的上下文信息。

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。)

1. 启动监听

Web 服务器启动后，会在指定端口 (如 80 或 60000) 上监听 TCP 连接请求。连接建立后，服务器进入就绪状态，等待客户端发送 HTTP 请求。

2. 解析请求

客户端 (如浏览器) 通过 TCP 连接发送 HTTP 请求报文，通常为 GET 请求。服务器收到报文后，首先从数据中提取请求行，解析出目标资源路径 (如 "/index.html")。

3. 查找文件并生成响应

根据请求的路径，服务器尝试在预设目录 (如 `static/`) 中查找对应的 HTML 文件:

如果文件存在，服务器读取文件内容，构造一个带有状态码 200 OK 的响应报文，包含 Content-Type、Content-Length 和数据正文;

如果文件不存在，服务器生成一个状态码为 404 Not Found 的响应报文，并返回一个简单的错误页面。

4. 发送响应数据

构造完响应报文后，通过 TCP 连接将 HTTP 响应数据发送给客户端。客户端收到后渲染网页内容。

5. 持久连接与关闭

由于使用的是 HTTP/1.1 协议，默认开启长连接 (keep-alive)。因此，客户端可以

复用已有的 TCP 连接请求多个资源（如图片、样式表等）。连接关闭通常由客户端发起，服务器被动响应 FIN 报文。

6. 函数实现说明

服务器核心逻辑在 http_respond() 函数中实现，流程如下：

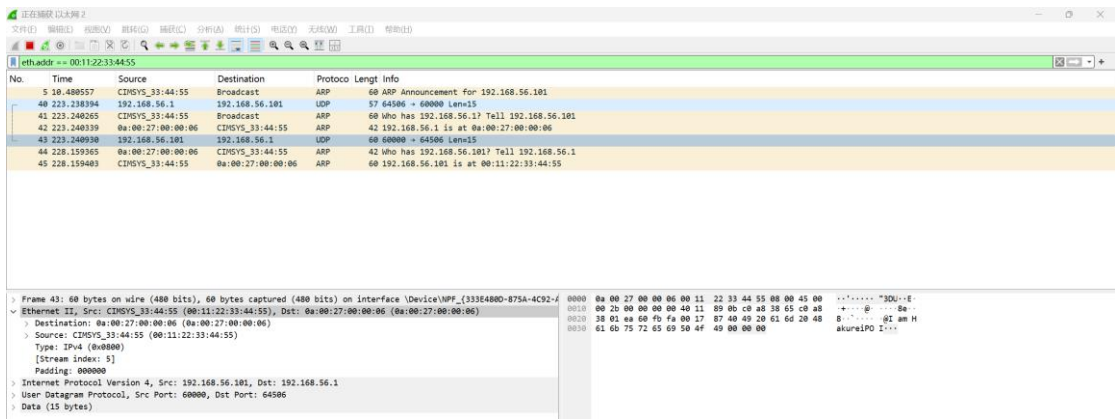
- 获取请求路径并进行字符串匹配；
- 判断文件是否存在，若不存在调用 send_404() 函数发送错误页面；
- 若存在，则读取文件并调用 send_response() 函数，返回 200 响应及文件内容。

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）



以第 43 号帧为例，该帧的结构信息如下：

目的 MAC 地址：0a:00:27:00:00:06

该地址为目的主机 192.168.56.1 对应的物理地址，是通过 ARP 协议获取到的，符合以太网通信要求。

源 MAC 地址：00:11:22:33:44:55

该地址为本机网卡的物理地址，正是实验中预设的 `NET_IF_MAC` 值，说明帧构造时正确设置了源地址字段。

协议类型字段：0x0800

该字段表示上层协议为 IPv4，符合 IP 层封装规范，说明 Ethernet 模块能够正确标识上层协议类型。

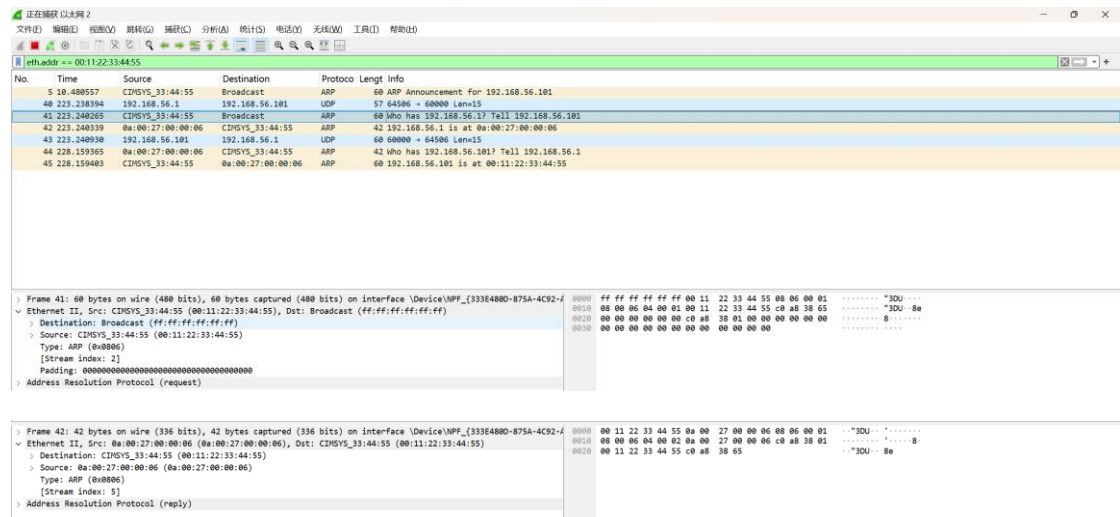
数据部分：

数据部分承载的是完整的 IP 数据报，进一步封装了 UDP 协议数据。UDP 的源端口为 60000，目的端口为 64506，Payload 长度为 15 字节，对应实际发送的字符串 "I am HakureiPOI"。十六进制内容中也可清晰看到 ASCII 字符串的编码，说明数据未被破坏，封装正确。

综上所述，该帧的以太网头部结构完整，字段设置正确，数据封装合理，说明`ethernet_out()`能够成功封装以太网帧，`ethernet_in()`能正确识别和解析帧结构，验证了 Ethernet 层收发流程的正确性。

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。)



在主机向目标发送 UDP 报文之前，会自动触发 ARP 协议解析过程。通过 Wireshark 抓包，成功捕获到了完整的 ARP 请求和响应帧，以第 41 帧和第 42 帧为例，具体分析如下：

ARP 请求帧（第 41 帧）：

该帧的源 MAC 地址为 00:11:22:33:44:55，对应本机自定义的网卡地址，源 IP 地址为 192.168.56.101，目标 IP 地址为 192.168.56.1，目的 MAC 地址为 ff:ff:ff:ff:ff:ff，表示广播。该请求帧用于询问局域网内谁拥有 IP 地址 192.168.56.1。

ARP 响应帧（第 42 帧）：

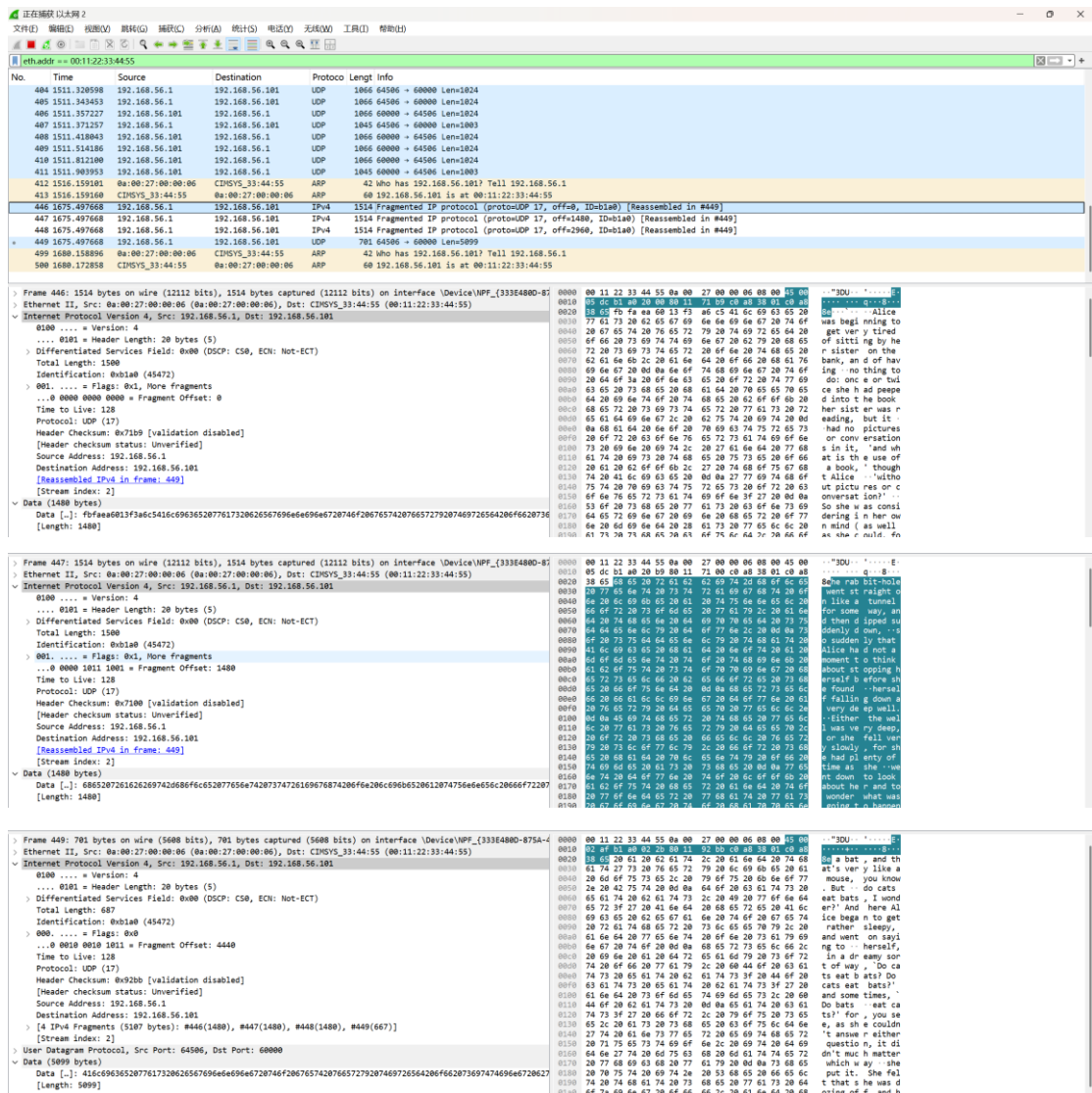
目标主机收到请求后回应，源 MAC 地址为 0a:00:27:00:00:06，对应 192.168.56.1，目标 MAC 地址为 00:11:22:33:44:55，即回复给发送方。该帧表明“192.168.56.1 的 MAC 地址是 0a:00:27:00:00:06”。

综上所述 ARP 请求与响应过程完整，字段设置正确，地址对应合理，说明 ARP 模

块能够正确构造广播请求、接收应答并解析目标主机 MAC 地址，满足地址解析协议的标准行为。

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。)



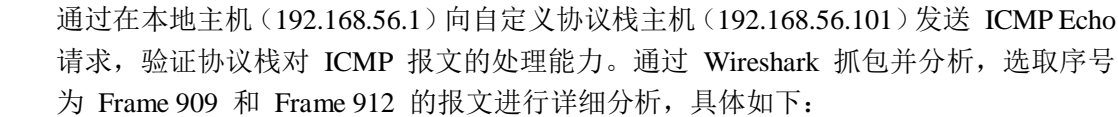
我们通过发送超长 UDP 报文(约 5099 字节), 触发了 IP 层的自动分片机制, 借助 Wireshark 抓包对分片报文的结构与字段进行分析, 验证协议实现的正确性

以第 446、447、448、449 号帧为例, 分析结果如下:

1. 版本号 (Version): 所有 IP 报文的版本字段均为 4, 说明使用的是 IPv4 协议, 符合预期。
2. 首部长度 (Header Length): 为 5, 对应 IP 报文头部长度的 20 字节, 未使用可选字段, 标准格式正确。
3. 总长度 (Total Length):
 - 第 446 帧: 1500 字节 (数据 1480 字节)
 - 第 447 帧: 1500 字节 (数据 1480 字节)
 - 第 448 帧: 1127 字节 (数据 1107 字节)表明分片过程中每片尽可能填满 MTU, 最后一片可能略短, 均合理。
4. 标识字段 (Identification): 0xb1a0 (十进制 45472), 三条分片报文均一致, 说明它们属于同一原始 IP 报文。
5. 标志位 (Flags) 与片偏移 (Fragment Offset):
 - 第 446 帧: Flags = 0x1 (MF = 1), Fragment Offset = 0 (首片)
 - 第 447 帧: Flags = 0x1 (MF = 1), Fragment Offset = 1480 / 8 = 185
 - 第 448 帧: Flags = 0x0 (MF = 0), Fragment Offset = 2960 / 8 = 370三帧构成完整的分片传输链条, 标志位和偏移值设置准确, 符合 RFC791 标准。
6. TTL (生存时间): 各分片 TTL 值为 128, 说明报文在网络中最大可跳数未被修改, 默认设置合理。
7. 协议类型 (Protocol)**: 字段值为 17, 表示上层为 UDP 协议, 匹配发送数据类型。
8. 源地址与目的地址: 源地址 192.168.56.1, 目的地址 192.168.56.101, 符合发送主机与接收主机的 IP 设置。
9. 重组情况: Wireshark 能够自动将三片 IP 报文重组为完整的 UDP 报文 (Frame 449), 数据长度为 5099 字节, 验证了 IP 层分片和重组功能的正确性。

通过分析可知, 实验成功触发了 IP 分片, 所有字段设置正确, 片偏移和标志位逻辑严谨, 重组过程完整无误, 说明协议栈中 IP 层的分片发送与重组接收模块实现正确, 符合 IP 协议规范和实际网络传输行为。

(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。)



数据部分为 32 字节，内容为 "abcdefghijklmnopqrstuvwxyzabcdefghi"

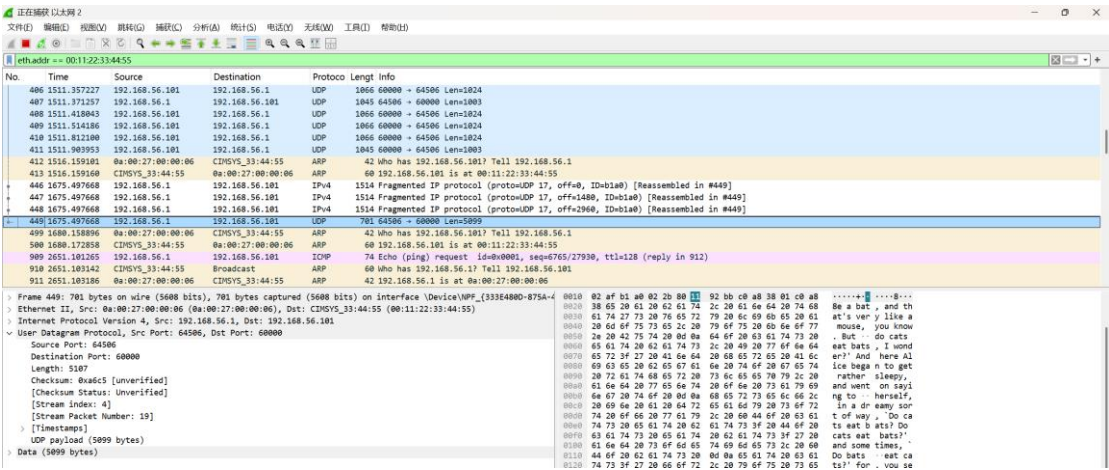
TTL: 64, 说明该报文由协议栈主机返回
ICMP 类型 (Type): 0, 代码 (Code): 0, 表示回显应答
校验和 (Checksum): 0x3aee, 状态为正确
标识符与序号与请求报文完全一致, 数据长度与内容也保持一致
响应时间为 3.644 毫秒, 显示回应及时且稳定

3. 数据校验: 请求和回应报文的 Identifier 与 Sequence Number 完全对应, 数据段未发生篡改或截断, 验证了协议栈能够正确识别 ICMP 请求, 并通过 `icmp_resp()` 准确生成回应。

综上, 协议栈的 ICMP 模块功能正常, 能够正确处理 ICMP Echo 请求并及时生成回应, 具备完整的 ICMP 报文封装与解析能力。

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。)



No.	Time	Source	Destination	Protocol	Length	Info
406	1511.377227	192.168.56.101	192.168.56.1	UDP	1066	60000 → 64506 Len=1024
407	1511.371257	192.168.56.1	192.168.56.101	UDP	1045	64506 → 60000 Len=1003
408	1511.418043	192.168.56.101	192.168.56.1	UDP	1066	60000 → 64506 Len=1024
409	1511.514186	192.168.56.101	192.168.56.1	UDP	1066	60000 → 64506 Len=1024
410	1511.812100	192.168.56.101	192.168.56.1	UDP	1066	60000 → 64506 Len=1024
411	1511.903953	192.168.56.101	192.168.56.1	UDP	1045	60000 → 64506 Len=1003
412	1516.155101	0a:00:27:00:00:06	CMISYS_33:44:55	ARP	42	Who has 192.168.56.101? Tell 192.168.56.1
413	1516.155160	CMISYS_33:44:55	0a:00:27:00:00:06	ARP	60	192.168.56.101 is at 00:11:22:33:44:55
446	1675.497668	192.168.56.1	192.168.56.101	IPv4	1514	Fragmented IP protocol (protocol 17, offset 0, ID=100) [Reassembled in #449]
447	1675.497668	192.168.56.1	192.168.56.101	IPv4	1514	Fragmented IP protocol (protocol 17, offset 1480, ID=100) [Reassembled in #449]
448	1675.497668	192.168.56.1	192.168.56.101	IPv4	1514	Fragmented IP protocol (protocol 17, offset 2960, ID=100) [Reassembled in #449]
449	1675.497668	192.168.56.1	192.168.56.1	UDP	791	64506 → 60000 Len=5099
499	1680.158896	0a:00:27:00:00:06	CMISYS_33:44:55	ARP	42	Who has 192.168.56.101? Tell 192.168.56.1
500	1680.172858	CMISYS_33:44:55	0a:00:27:00:00:06	ARP	60	192.168.56.101 is at 00:11:22:33:44:55
909	2651.101265	192.168.56.1	192.168.56.101	ICMP	74	Echo (ping) request id=0x0001, seq=0x527930, ttl=128 (reply in 912)
910	2651.103142	CMISYS_33:44:55	Broadcast	ARP	60	Who has 192.168.56.1? Tell 192.168.56.101
911	2651.103186	0a:00:27:00:00:06	CMISYS_33:44:55	ARP	42	192.168.56.1 is at 0a:00:27:00:00:06

Frame 449: 791 bytes on wire (6328 bits), 791 bytes captured (6328 bits) on interface Device\NPF_{333E4800-875A-4...} Ethernet II, Src: 0a:00:27:00:00:06 (0a:00:27:00:00:06), Dst: CMISYS_33:44:55 (00:11:22:33:44:55)
Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.101
User Datagram Protocol, Src Port: 64506, Dst Port: 60000
Source Port: 64506
Destination Port: 60000
Length: 5107
Checksum: 0xa6c5 [Unverified]
[Checksum Status: Unverified]
[Stream Index: 4]
[Stream Packet Number: 19]
[Timestamps]
UDP payload (5099 bytes)
Data (5099 bytes)

以 Frame 449 (之前发送的大量数据的重组) 为例, UDP 报文结构如下:

源端口号 (Source Port): 64506

目的端口号 (Destination Port): 60000

长度 (Length): 5107 字节, 包含 8 字节 UDP 头部和 5099 字节数据部分

校验和 (Checksum): 0xa6c5, Wireshark 显示校验状态为 unverified (因在抓包时未启用校验计算, 不影响协议栈测试)

综上所述, 实验表明协议栈已具备处理 UDP 报文的完整功能, 包括头部解析、载荷接收与分片重组, 整体符合协议设计预期。

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。)

三次握手

The following table summarizes the key packets from the Wireshark capture:

No.	Time	Source	Destination	Protocol	Length	Info
29	0.991889	192.168.56.1	192.168.56.101	TCP	64	47620 → 60000 [SYN] Seq=0 Window=65535 Len=0 MSS=1460 WS=256 SACK_PERM
32	0.994752	192.168.56.101	192.168.56.1	TCP	60	60000 → 47620 [SYN, ACK] Seq=0 Ack=1 Window=65535 Len=0
33	0.996609	192.168.56.1	192.168.56.101	TCP	54	47620 → 60000 [ACK] Seq=1 Ack=1 Window=65535 Len=0

Detailed analysis of the three-way handshake:

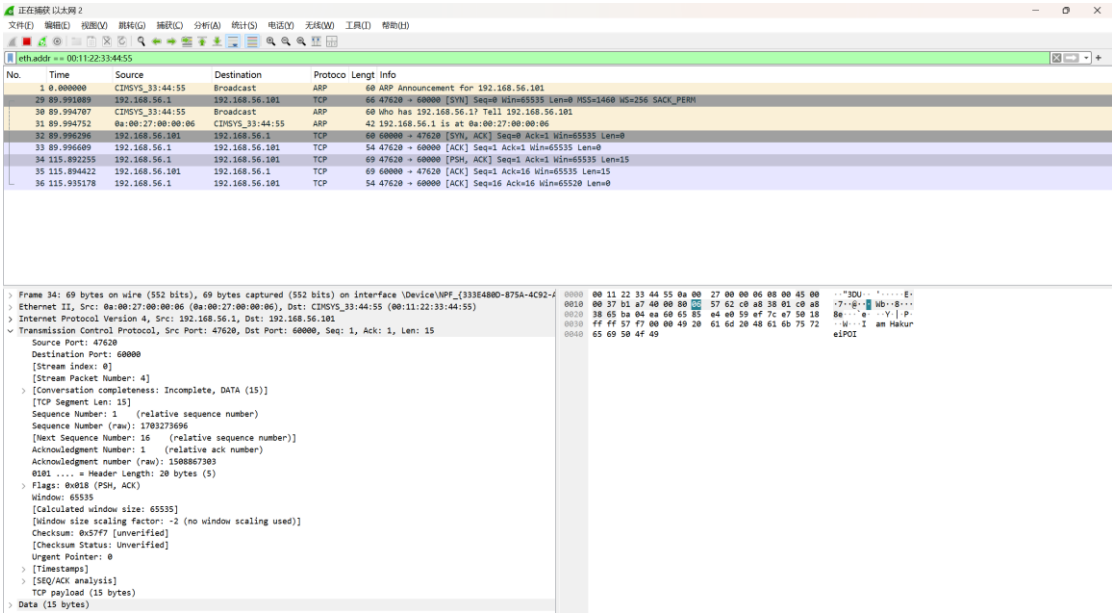
- Frame 29:** 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface \Device\NPF{333E4800-875A-4C92-4-00000000}. Ethernet II, Src: 08:00:27:00:00:06 (08:00:27:00:00:06), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55). Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.101. Transmission Control Protocol, Src Port: 47620, Dst Port: 60000, Seq: 0, Len: 0. Source Port: 47620, Destination Port: 60000. [Stream index: 0]. [Stream Packet Number: 1]. [Conversation completeness: Incomplete, DATA (15)]. [TCP Segment Len: 0]. Sequence Number: 0 (relative sequence number). [Next Sequence Number: 1 (relative sequence number)]. Acknowledgment Number: 0. Acknowledgment number (raw): 0. Window: 0. Header Length: 32 bytes (8). Flags: 0x002 (SYN). Window: 65535. [Calculated window size: 65535]. Checksum: 0x0000 [Unverified]. [Checksum Status: Unverified]. Urgent Pointer: 0. Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP). [Timestamps].
- Frame 32:** 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \Device\NPF{333E4800-875A-4C92-4-00000000}. Ethernet II, Src: 08:00:27:00:00:06 (08:00:27:00:00:06), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55). Internet Protocol Version 4, Src: 192.168.56.101, Dst: 192.168.56.1. Transmission Control Protocol, Src Port: 60000, Dst Port: 47620, Seq: 0, Ack: 1, Len: 0. Source Port: 60000, Destination Port: 47620. [Stream index: 0]. [Stream Packet Number: 2]. [Conversation completeness: Incomplete, DATA (15)]. [TCP Segment Len: 0]. Sequence Number: 0 (relative sequence number). [Next Sequence Number: 1 (relative sequence number)]. Acknowledgment Number: 1 (relative ack number). Acknowledgment number (raw): 1703273696. Window: 0. Header Length: 20 bytes (5). Flags: 0x012 (SYN, ACK). Window: 65535. [Calculated window size: 65535]. Checksum: 0xf879 [Unverified]. [Checksum Status: Unverified]. Urgent Pointer: 0. [Timestamps]. [SEQ/ACK analysis].
- Frame 33:** 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface \Device\NPF{333E4800-875A-4C92-4-00000000}. Ethernet II, Src: 08:00:27:00:00:06 (08:00:27:00:00:06), Dst: CIMSYS_33:44:55 (00:11:22:33:44:55). Internet Protocol Version 4, Src: 192.168.56.1, Dst: 192.168.56.101. Transmission Control Protocol, Src Port: 47620, Dst Port: 60000, Seq: 1, Ack: 1, Len: 0. Source Port: 47620, Destination Port: 60000. [Stream index: 0]. [Stream Packet Number: 3]. [Conversation completeness: Incomplete, DATA (15)]. [TCP Segment Len: 0]. Sequence Number: 1 (relative sequence number). [Next Sequence Number: 1 (relative sequence number)]. Acknowledgment Number: 1 (relative ack number). Acknowledgment number (raw): 1508867303. Window: 0. Header Length: 20 bytes (5). Flags: 0x010 (ACK). Window: 65535. [Calculated window size: 65535]. [Window size scaling factor: -2 (no window scaling used)]. Checksum: 0xf87a [Unverified]. [Checksum Status: Unverified]. Urgent Pointer: 0. [Timestamps]. [SEQ/ACK analysis].

报文 29: 192.168.56.1 向 192.168.56.101 发送 SYN 报文, 请求建立连接, 源端口 47620, 目的端口 60000, Seq=0, 窗口大小为 65535。

报文 32: 192.168.56.101 返回 SYN-ACK 报文, 确认收到请求, Seq=0, Ack=1, 表示已准备接收。

报文 33: 192.168.56.1 回复 ACK, Seq=1, Ack=1, 连接正式建立。

数据发送与确认



报文 34: 192.168.56.1 发送带有 PSH 标志的数据报文, Seq=1, 数据长度为 15 字节 (即字符串 “I am HakureiPOI”)。

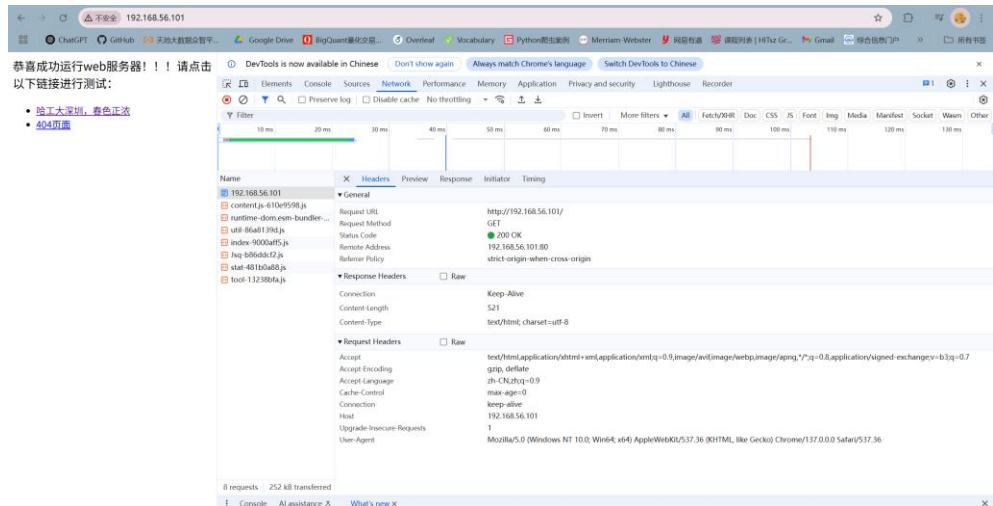
报文 35: 192.168.56.101 回复 ACK, 确认号 Ack=16, 表明已正确接收前 15 字节的数据。

报文 36: 192.168.56.1 再次发送 ACK 报文, 为后续交互作准备。

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。)

用 chrome 的开发者工具查看如下



由图可知，这是一次浏览器向本地服务器（192.168.56.101）发起的 HTTP GET 请求，成功返回网页内容

请求部分：

方法：GET

URL: http://192.168.56.101/

请求头：包含浏览器信息、支持的编码、语言等

响应部分：

状态码：200 OK（请求成功）

响应头：内容类型为 HTML，长度为 521 字节

响应体：显示“恭喜成功运行 web 服务器”，附带两个测试链接

综上，客户端成功访问本地 Web 服务器，页面正常返回，说明服务器已正常运行

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。）

在本次实验中，除了一些代码编写和调试环境配置上的常规问题外，主要遇到的问题是实验周期较长，导致在设计或调试过程中对先前实现细节的遗忘。这种情况在实验的后期尤其突出，前期为了满足某一阶段的测试需求而作出的设计简化，往往会与后续功能的完整实现产生冲突。

印象较深的问题出现在实现 TCP 协议功能时。当时使用 Wireshark 抓包时，发现客户端始终无法收到 TCP 数据报，而是频繁接收到 ICMP 协议不可达的差错报文。起初无法判断是哪一层出现了异常，于是在 IP 层、TCP 层等多个关键环节增加日志输出，逐步排查后发现问题根源在 IP 层的协议处理逻辑中：为了满足前期实验的简单需求，代码中只允许 ICMP 和 UDP 协议被处理，对于 TCP 协议则直接返回差错报文。该逻辑在前期实验中是有效的，但在后续 TCP 协议实现中显然不再适用。

最终通过修改 IP 层逻辑，将协议判断下沉到统一的协议分发接口 `net_in()` 中，并根据其返回值判断是否返回差错报文，从而解决了该问题，使得 TCP 报文可以被正确识别和处理，抓包数据也恢复正常。

```
src/ip.c
45 45 memcpy(src_ip, ip_hdr->src_ip, NET_IP_LEN);
46 46 uint8_t protocol = ip_hdr->protocol;
47 47
48 - // 若协议类型不支持，则返回 ICMP 协议不可达
49 - if (protocol != NET_PROTOCOL_ICMP && protocol != NET_PROTOCOL_UDP) {
50 -     icmp_unreachable(buf, src_ip, ICMP_CODE_PROTOCOL_UNREACH);
51 -     return;
52 - }
53 -
54 48 // 去除 IP 头部，准备向上层协议传递
55 49 buf_remove_header(buf, ip_hdr->hdr_len * IP_HDR_LEN_PER_BYTE);
56 50
57 51 // 调用统一接口，将数据交由上层协议处理
58 - net_in(buf, protocol, src_ip);
59 + int ret = net_in(buf, protocol, src_ip);
60 + if (ret == -1) {
61 +     // 若协议类型不支持，则返回 ICMP 协议不可达
62 +     icmp_unreachable(buf, src_ip, ICMP_CODE_PROTOCOL_UNREACH);
63 +     return;
64 + }
65 }
```

另一个遇到的问题是在实现 IP 协议时曾预留了对 ICMP 差错报文的处理逻辑。当时由于 ICMP 协议尚未实现，计划先调用一个空的 icmp_unreachable() 函数占位，待后续补全具体功能。但在后续真正开始实现 ICMP 功能时，发现数据包始终无法正常发送，调试了很久仍无法定位问题。

由于前后间隔较长，加上当时对 ICMP 报文格式和触发条件理解不系统，理论知识也不够扎实，导致排查方向一度偏离。最终经过回顾 IP 层实现逻辑，并对照调试输出，才意识到问题出在最初忘记在 IP 层调用 icmp_unreachable() 函数，导致虽然 ICMP 模块已实现，但从未被正确触发调用。

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。)

通过这次配置实验和协议栈的实现，我对计算机网络的理解有了很大提升。以前看教材资料的时候，很多协议流程和字段说明都只是停留在文字上，感觉很抽象，看完也记不住。但通过这次一步步写代码、调试、抓包，我真正理解了像“封装 header、发送数据包、拆包接收”等过程到底是怎么实现的，同时使用 wireshark 抓包看到自己发出的报文，让我对协议格式有了直观理解。总之实验过程非常有价值，感谢实验老师和助教的付出