# 第二次实验报告：使用Libpcap编程

**2112614 刘心源**

## 一、实验要求

1. 了解NPcap(Libpcap)的架构。
2. 学习NPcap(Libpcap)的设备列表获取方法、网卡设备打开方法，以及数据包捕获方法。
3. 通过NPcap(Libpcap)编程，实现本机的IP数据报捕获，显示捕获数据帧的源MAC地址和目的MAC地址，以及类型/长度字段的值。
4. 捕获的数据报不要求硬盘存储，但应以简单明了的方式在屏幕上显示。必显字段包括源MAC地址、目的MAC地址和类型/长度字段的值。
5. 编写的程序应结构清晰，具有较好的可读性。

## 二、实验目的

1. 学习网络数据包捕获方法
2. 初步掌握网络监听与分析技术的实现过程
3. 理解IP数据包校验和计算方法

## 三、实验原理

### Libpcap

Libpcap是一个跨平台的数据包捕获库；支持多个操作系统，包括macOS、Linux，BSD等；在Windows上的版本为`WinPcap`。

- **跨平台支持**：Libpcap 提供了一个在多种Unix-like操作系统上进行原始数据包捕获的功能强大的用户级库。它在 Linux、macOS 以及其他多种 Unix 类型的系统上都可以运行，而不仅仅是限于某个特定的平台或架构，为开发者提供了广泛的开发和部署选择。
- **高效的数据包过滤**：Libpcap 拥有一个高效的数据包过滤系统，它能够在内核级别对数据包进行过滤，极大地减小了需要在用户空间处理的数据包的数量。开发者可以使用 BPF（Berkeley Packet Filter）语法定义数据包过滤规则，以便只捕获对分析和处理特定问题相关的数据包。
- **数据包捕获机制**：Libpcap 提供了多种数据包捕获的机制，例如 `pcap_loop()` 和 `pcap_dispatch()`，它们分别提供了基于数据包数量和超时的数据包捕获。此外，还有 `pcap_next()` 和 `pcap_next_ex()` 函数，这些函数提供了更简单的基于单个数据包的捕获方法。
- **易于使用的 API**：Libpcap 提供了一套简单且功能强大的 API，开发者可以使用这些 API 快速地构建自己的网络监控或分析工具。Libpcap API 允许开发者执行如打开网络接口、编译和应用过滤器、捕获数据包等基本操作。
- **底层数据包处理**：Libpcap 处理了与底层网络硬件和操作系统交互所需的所有细节，如读取数据包、处理数据包等，使开发者能够专注于处理捕获的数据包，而无需担心底层的具体实现细节。
- **开源与社区支持**：Libpcap 是开源的，并且有一个活跃的开发和用户社区。因此，它能够支持最新的网络技术和协议，也能够在遇到问题时得到社区的帮助和支持。
- **扩展性和兼容性**：Libpcap 支持通过各种语言的绑定和包装器在多种编程语言中使用，如 Python、Perl、

Ruby 等。这为在多种场景和平台下开发网络工具提供了极大的便利和灵活性。同时，由于 Libpcap 的广泛使用，它在网络工具开发领域形成了一种事实上的标准，许多工具和库为其提供了支持或集成。

## Libpcap的安装

1. 在MacOS上使用homebrew：

```
brew install libpcap
```

2. 在代码中引入库：

```
#include <pcap.h>
```

## 数据包捕获函数

### 设备发现

- `pcap_findalldevs()`: 获取本地机器上所有网络设备的列表。

### 网络设备设置

- `pcap_open_live()`用于打开指定的网络设备，并且返回用于捕获网络数据包的数据包捕获描述字。对于此网络设备的操作都要基于此网络设备描述字。
- `pcap_lookupnet()`: 获取指定网络设备的网络号和掩码。

### 设置过滤器

- `pcap_compile()`: 将用户指定的过滤策略编译为过滤程序。
- `pcap_setfilter()`: 将编译后的过滤程序设置到 pcap 句柄。

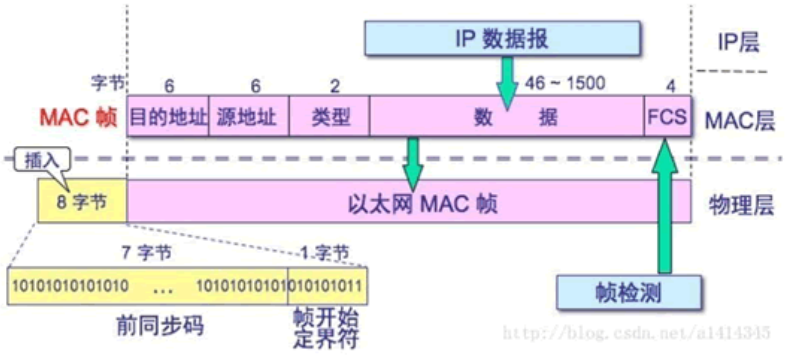
### 数据包捕获

- `pcap_loop()` 或 `pcap_dispatch()`: 捕获数据包并调用指定的回调函数进行处理。
- `pcap_next()` 和 `pcap_next_ex()`: 捕获单个数据包。

### 关闭网络设备

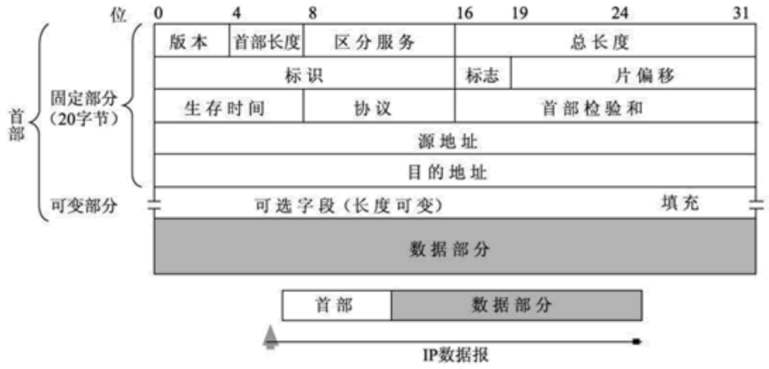
- `pcap_close()`: 关闭已打开的网络设备并释放资源。

## 以太网帧的结构




1. **前同步码 (Preamble) 和开始帧定界符 (Start Frame Delimiter, SFD)**
    - **前导序列**：用来使接收端的适配器在接收 MAC 帧时能够迅速调整时钟频率，使它和发送端的频率相同。前同步码为 7 个字节，1 和 0 交替。
    - **开始帧定界符**：帧的起始符，为 1 个字节，通常是 `10101011`。前 6 位1和0交替，最后的两个连续的1表示告诉接收端适配器：“帧信息要来了，准备接收”。
2. **目的地址 (Destination Address, DA)**：6 字节
    - 接收帧的网络适配器的物理地址（MAC 地址）。作用是当网卡接收到一个数据帧时，首先会检查该帧的目的地址，是否与当前适配器的物理地址相同，如果相同，就会进一步处理；如果不同，则直接丢弃。
3. **源地址 (Source Address, SA)**：6 字节
    - 发送帧的网络适配器的物理地址（MAC 地址）
4. **类型/长度 (Type/Length)**：2 字节
    - 上层协议的类型。由于上层协议众多，所以在处理数据的时候必须设置该字段，标识数据交付哪个协议处理。
    - 例如，字段为 0x0800 时，表示将数据交付给 IP 协议。
    - 如果值是 1500 以下，则表示“长度”（帧的数据部分的长度）。
    - 如果值是 1536 以上，则表示“类型”（指定网络层协议的类型，如 IPv4 或 IPv6）。
5. **数据和填充 (Data and Padding)**：46-1500 字节
    - **数据**：这部分包含上层协议（如 IP）的数据。也称为效载荷，表示交付给上层的数据。以太网帧数据长度最小为 46 字节，最大为 1500 字节。
    - **填充**：如果数据字段的长度少于 46 字节，那么会使用填充字节来确保帧的总长度至少为 64 字节。
6. **帧校验序列 (Frame Check Sequence, FCS)**：4 字节
    - 这是一个 CRC（循环冗余校验）值，用于错误检测。接收端会计算接收到的帧的 FCS，并与这个字段中的值进行比较，以检查帧是否在传输过程中出错。如果两个值不相同，则表示传输过程中发生了数据丢失或改变。这时，就需要重新传输这一帧。
    - 一个简单的计算过程：
        - 假设原始数据是：101001
        - 使用生产多项式G(x)(收发双方事先约定)，比如G(x) = x^3 + x^2+ 1
        - 构造被除数：原始数据+生成多项式最高次项个0，即101001000
        - 除数就是生成多项式的系数，也就是1101
        - 两数相除得到余数，并且进行补位(补到与生成多项式最高次项一致)。注意⚠️：这里的除法上下两行数使用的是异或运算
        - 使用上面的运算得到校验码001，添加到原始数据之后得到的最终发送数据为 `101001001`
        - 假设收到的数据是101001001，接收方对于该数据做除法，除数依然是之前的1101，若余数为0则表示没有出错，余数不为0则出现错误。

## IP数据包的首部结构

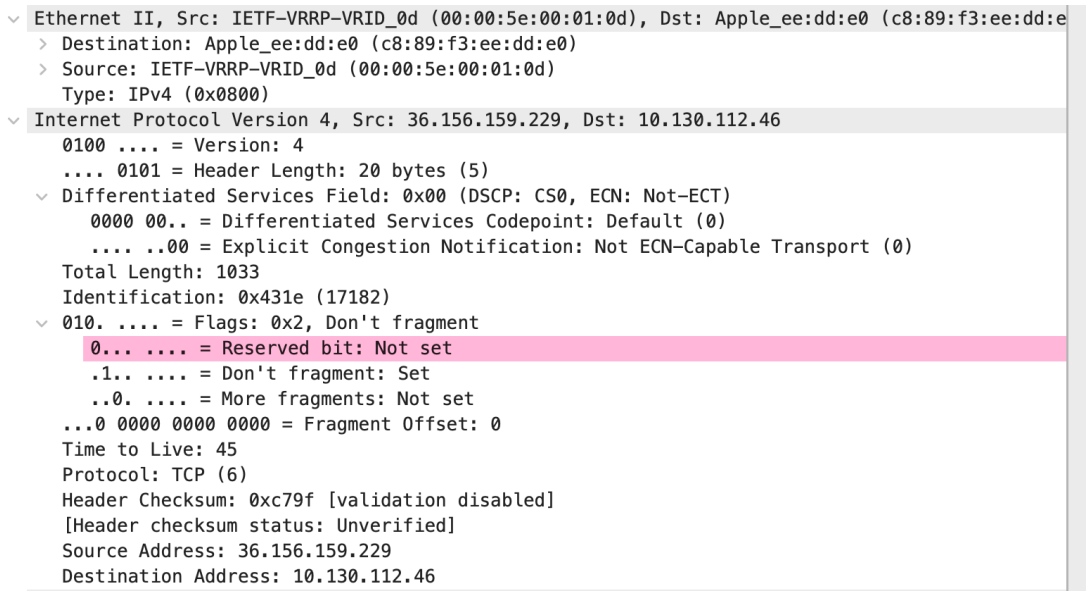



IP 报头的最小长度为 20 字节，以最常用的IPv4结构为例：

1. **版本 (Version)**：4 bits
    - 表示 IP 协议的版本。通信双方使用的 IP 协议版本必须一致。IPv4 的版本字段值为 4。
2. **首部长度 (Header Length/IHL)**：4 bits
    - 表示 IP 首部的长度。单位是 32 bits（4 字节）。占 4 位，可表示的最大十进制数值是 15。因此，当 IP 的首部长度为 1111 时（即十进制的 15），首部长度就达到 60 字节。
3. **区分服务 (Type of Service/TOS)**：8 bits
    - 也被称为服务类型，用于定义服务质量，包括优先级、延迟、吞吐量等。
4. **总长度 (Total Length)**：16 bits
    - 包括首部和数据的总长度。单位是字节。总长度字段为 16 位，因此数据报的最大长度为 2^16-1=65535 字节。
5. **标识 (Identification)**：16 bits
    - 用来标识数据报，占 16 位。IP 协议在存储器中维持一个计数器。每产生一个数据报，计数器就加 1，并将此值赋给标识字段。当数据报的长度超过网络的 MTU，而必须分片时，这个标识字段的值就被复制到所有的数据报的标识字段中。具有相同的标识字段值的分片报文会被重组成原来的数据报。
6. **标志 (Flags)**：3 bits
    - 包括“不分片”和“更多片”标志。占 3 位。第一位未使用，其值为 0。第二位称为 DF（不分片），表示是否允许分片。取值为 0 时，表示允许分片；取值为 1 时，表示不允许分片。第三位称为 MF（更多分片），表示是否还有分片正在传输，设置为 0 时，表示没有更多分片需要发送，或数据报没有分片。
7. **片偏移 (Fragment Offset)**：13 bits
    - 用于指示片在原始数据报中的位置。当报文被分片后，该字段标记该分片在原报文中的相对位置。片偏移以 8 个字节为偏移单位。所以，除了最后一个分片，其他分片的偏移值都是 8 字节（64 位）的整数倍。
8. **生存时间 (Time to Live/TTL)**：8 bits
    - 限制数据报的生存时间。该字段由发出数据报的源主机设置。其目的是防止无法交付的数据报无限制地在网络中传输，从而消耗网络资源。
    - 路由器在转发数据报之前，先把 TTL 值减 1。若 TTL 值减少到 0，则丢弃这个数据报，不再转发。因此，TTL 指明数据报在网络中最多可经过多少个路由器。TTL 的最大数值为 255。若把 TTL 的初始值设为 1，则表示这个数据报只能在本局域网中传送。
9. **协议 (Protocol)**：8 bits

- 表示该数据报文所携带的数据所使用的协议类型。该字段可以方便目的主机的 IP 层知道按照什么协议来处理数据部分。不同的协议有专门不同的协议号。例如，TCP 的协议号为 6，UDP 的协议号为 17，ICMP 的协议号为 1。

10. **首部校验和 (Header Checksum)**：16 bits
    - 用于检测首部是否在传输过程中出错。数据报每经过一个路由器，首部的字段都可能发生变化（如 TTL），所以需要重新校验。而数据部分不发生变化，所以不用重新生成校验值。
11. **源 IP 地址 (Source IP Address)**：32 bits
    - 发送者的 IP 地址。
12. **目的 IP 地址 (Destination IP Address)**：32 bits
    - 接收者的 IP 地址。
13. **选项 (Options)**：可变长度
    - 包括一些可选的参数，如记录路由、时间戳等。
14. **数据 (Data)**
    - IP 数据报的数据部分，包含上层协议的数据。
15. **填充**
    - 由于可选字段中的长度不是固定的，使用若干个 0 填充该字段，可以保证整个报头的长度是 32 位的整数倍。




## 首部校验和的计算方法

**1. 划分段**

- 在计算首部校验和之前，需要先将首部校验和字段设置为0。
- 将IP首部的每个字段（包括版本，首部长度，服务类型等）划分为16位（2字节）的段。如果IP首部包含选项字段，那么这些选项字段也要包括在内。
- 如果首部的长度不是16位的整数倍，通常会在尾部添加填充零。

**2. 求和**

- 将所有16位的段相加。在这个过程中，会忽略任何进位。

**3. 处理进位**

- 如果步骤2中的加法产生了进位，那么将进位的值加到结果中。

**4. 取反**

- 将步骤3得到的结果取反（按位取反）。

**5. 放入首部**

- 将步骤4得到的结果放入IP首部的“首部校验和”字段。

**6.在接收端处**

- 接收端会进行类似的计算，然后将计算得到的值与接收到的首部校验和字段的值进行比较。
- 如果两个值相同，那么首部被认为是没有错误的。
- 如果两个值不同，那么首部被认为是有错误的，并且该数据包会被丢弃。

注意：当IP数据报通过路由器时，由于某些字段（例如生存时间TTL）可能会发生变化，因此可能需要重新计算和验证首部校验和。

# 四、关键代码

**结构体定义**

```
Packet.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pcap.h>
/*
* 以太头
*/
struct ether_header
{
    uint8_t Ethernet_Dhost[6]; // 目的地址
    uint8_t Ethernet_Shost[6]; // 源地址
    uint16_t Ethernet_Type;    // 以太网类型
};

/*
* IP头
*/
struct ip_header // IP首部
{
    uint8_t Ver_HLen;       // 8位版本+首部长度
```

```
    uint8_t TOS;            // 8位服务类型
    uint16_t TotalLen;      // 16位总长度
    uint16_t ID;            // 16位标识
    uint16_t Flag_Segment; // 16位标志3+片偏移13
    uint8_t TTL;            // 8位生存时间
    uint8_t Protocol;       // 8位协议
    uint16_t Checksum;      // 16位首部校验和
    uint32_t SrcIP;         // 32位源IP地址
    uint32_t DstIP;         // 32位目的IP地址
};

void analysis_ip(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char
*packet);
void showIPPacket(struct ip_header* iphdr);
void analysis_ether(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char
*packet);
void showEtherPacket(struct ether_header* ethhdr);

void packet_handler(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char
*packet);
```

在`Packet.c`文件中主要定义了IP头和以太头的结构体。

### checksum(struct ip_header* iphdr)

```
uint16_t checksum(struct ip_header* iphdr){
    uint32_t sum = 0;
    uint16_t* ptr = (uint16_t*)iphdr;
    iphdr->Checksum = 0;//将校验和字段置为0
    // 将IP首部中的每16位相加
    for(int i = 0; i < 10; i++){
        sum += ntohs(*ptr);//将大端字节序转换为小端字节序
        ptr++;
    }
    // 把高16位和低16位相加
    while(sum >> 16){
        sum = (sum >> 16) + (sum & 0xffff);//sum>>16高 sum&0xffff低
    }
    //返回校验和     ~sum表示取反
    return (uint16_t)(~sum);
}
```

计算校验和，将`struct ip_header`结构体中的每16位进行相加，然后将相加的结果相加，直到结果不再超过16位。最后需要将结果取反，以得到最终的校验和。这是因为在计算校验和时，所有的16位值都是以二进制补码的形式进行相加的。而校验和的取反操作可以将校验和的二进制补码形式转换为反码形式，以便接收方进行校验。

### void print_binary(uint16_t value, int bits)

```
void print_binary(uint16_t value, int bits) {
    for (int i = bits - 1; i >= 0; --i) {
        printf("%d", (value >> i) & 1);
    }
    printf("\n");
}
```

这段代码实现了一个将一个16位的无符号整数转换为二进制字符串的函数。

`value >> i` 表示将 `value` 右移 `i` 位，得到前 `i` 位的值。`(value >> i) & 1` 表示将右移后的结果与 1 进行按位与操作，得到第 `i` 位的值。如果第 `i` 位的值为 1，则输出字符 '1'，否则输出字符 '0'。最后，将所有的字符拼接起来，得到一个二进制字符串。

### void analysis_ip(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char *packet)

这个函数用来解析IP数据包。从输入参数`struct ip_header* iphdr`这个指针所指向的IP数据包中依次输出 IP 数据包的各个字段的值，包括版本号、首部长度、服务类型等。其中，版本号和首部长度需要从 `Ver_HLen` 字段中解析出来，服务类型需要从 `TOS` 字段中解析出来。

函数会根据服务类型的值输出不同的提示信息，表示数据包的优先级。最后，函数会输出源 IP 地址和目的 IP 地址的值。

```
void analysis_ip(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char
*packet){
    struct ip_header *iphdr;
    iphdr = (struct ip_header *)(packet + 14); // 获取IP头，14 是因为以太网帧头部通常是 14 字
节
    printf("============开始解析IP层数据包========== \n");
    printf("版本号: %d\n", (iphdr->Ver_HLen >> 4));
    printf("首部长度: %d\n", (iphdr->Ver_HLen & 0x0F) * 4);
    printf("服务类型: %d\n", (int)iphdr->TOS);
    switch ((int)iphdr->TOS >> 5) {
    case 0:
        printf("优先级: Routine,数据包不需要特殊处理。\n");
        break;
    case 1:
        printf("优先级: Priority,数据包不需要特殊处理。\n");
        break;
    case 2:
        printf("优先级: Immediate,数据包需要立即处理。\n");
        break;
    case 3:
        printf("优先级: Flash,数据包需要快速处理。\n");
        break;
```

```
    case 4:
        printf("优先级：Flash Override,数据包需要立即处理，并覆盖其他数据包的处理。\n");
        break;
    case 5:
        printf("优先级：CRITIC/ECP,数据包是关键数据或网络控制数据，需要最高优先级处理。\n");
        break;
    case 6:
        printf("优先级：Internetwork Control,数据包是网络控制数据，需要高优先级处理。\n");
        break;
    case 7:
        printf("优先级：Network Control,数据包是网络控制数据，需要最高优先级处理。\n");
        break;
    }
    printf("总长度：%d\n", ntohs(iphdr->TotalLen));
    printf("标识：%d\n", ntohs(iphdr->ID));
    printf("标志：");
    print_binary(ntohs(iphdr->Flag_Segment) >> 13, 3);
    printf("对应标志：保留位、DF、MF。\n");
    printf("片偏移：%d\n", (ntohs(iphdr->Flag_Segment) & 0x1FFF)); // 0x1FFF = 0001 1111
1111 1111，取后13位
    printf("生存时间：%d\n", (int)iphdr->TTL);
    printf("协议编号：%d\n", (int)iphdr->Protocol);
    switch ((int)iphdr->Protocol) {
    case 1:
        printf("协议类型：ICMP\n");
        break;
    case 2:
        printf("协议类型：IGMP\n");
        break;
    case 6:
        printf("协议类型：TCP\n");
        break;
    case 17:
        printf("协议类型：UDP\n");
        break;
    case 58:
        printf("协议类型：ICMPv6\n");
        break;
    case 89:
        printf("协议类型：OSPF\n");
        break;
    case 132:
        printf("协议类型：SCTP\n");
        break;
    case 255:
        printf("协议类型：RAW\n");
        break;
    }
    printf("首部校验和：%d\n", ntohs(iphdr->Checksum));
```

```
    printf("源IP地址: %s\n", inet_ntoa(*(struct in_addr *)&iphdr->SrcIP));
    printf("目的IP地址: %s\n", inet_ntoa(*(struct in_addr *)&iphdr->DstIP));
    printf("============完成IP层数据包解析======== \n\n");
}
```

**void analysis_ether(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char *packet)**

```
void analysis_ether(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char
*packet){
    struct ether_header *ethhdr;
    ethhdr = (struct ether_header *)packet; // 获取以太网帧头部
        uint16_t type = ntohs(ethhdr->Ethernet_Type);
    printf("============开始解析以太网帧数据包======== \n");
    printf("目的MAC地址: %02x:%02x:%02x:%02x:%02x:%02x\n",
            ethhdr->Ethernet_Dhost[0], ethhdr->Ethernet_Dhost[1], ethhdr-
>Ethernet_Dhost[2],
            ethhdr->Ethernet_Dhost[3], ethhdr->Ethernet_Dhost[4], ethhdr-
>Ethernet_Dhost[5]);
    printf("源MAC地址: %02x:%02x:%02x:%02x:%02x:%02x\n",
            ethhdr->Ethernet_Shost[0], ethhdr->Ethernet_Shost[1], ethhdr-
>Ethernet_Shost[2],
            ethhdr->Ethernet_Shost[3], ethhdr->Ethernet_Shost[4], ethhdr-
>Ethernet_Shost[5]);
    printf("以太网类型: %04x\n", type);
    switch (type) {
    case 0x0800:
        printf("以太网类型: IP\n");
        analysis_ip(user_data, pkthdr, packet);
        break;
    case 0x0806:
        printf("以太网类型: ARP\n");
        break;
    case 0x8035:
        printf("以太网类型: RARP\n");// RARP 数据包 (反向地址解析协议)
        break;
    case 0x86DD:
        printf("以太网类型: IPv6\n");
        break;
    default:
        printf("以太网类型: 其他\n");
        break;
    }
    printf("============完成以太网帧数据包解析======== \n\n");
}
```

**回调函数：void packet_handler(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char *packet)**

参数解释：

1. `u_char *user_data`：

   用户定义的数据。这个参数可以用于将额外的数据传递给回调函数。当需要在回调函数中访问某些全局数据时会用到。

2. `const struct pcap_pkthdr *pkthdr`：

   数据包头部信息。这个结构包含了关于捕获到的数据包的一些基本信息，如数据包的长度和时间戳。通过这个参数可以获取数据包的捕获时间和数据包的实际长度和捕获长度。

3. `const u_char *packet`：

   数据包的内容。这个参数指向捕获到的数据包的内容。可以通过这个参数来访问和解析数据包的内容，包括以太网头部、IP 头部、TCP 头部等。

```
void packet_handler(u_char *user_data, const struct pcap_pkthdr *pkthdr, const u_char
*packet) {
    // 定义以太网和IP头的最小总长度
    const unsigned int MIN_PACKET_LEN = 14 + 20;
    printf("---------------捕获到一个数据包---------------\n");

    // 检查数据包长度
    if (pkthdr->len < MIN_PACKET_LEN)
    {
        fprintf(stderr, "Packet is too short (%d bytes), unable to parse\n", pkthdr-
>len);
        return;
    }
    analysis_ether(user_data, pkthdr, packet);
}
```

回调函数（Callback Function）是一种在编程中广泛应用的机制，它允许将一个函数作为参数传递给另一个函数，并在该函数内部在适当的时间执行这个传递的函数。这种机制提供了很高的灵活性，并能使得代码结构更加清晰和模块化。

1. **定义和基本概念**：
   - 回调函数是一种将函数作为参数传递给另一函数的方法。在这种情况下，传递的函数（回调函数）将在主函数（接收回调函数的函数）内部的某个点被调用。
2. **应用场景**：
   - **异步编程**：回调函数在异步编程中尤为常见，例如处理网络请求、定时任务、或事件处理等场景。
   - **高阶函数**：在许多编程语言中，回调函数用于实现高阶函数，例如数组的 map、filter 和 reduce 方法。
3. **优势**：
   - **模块化**：通过回调，可以将复杂的逻辑分解为更小、更易于管理的部分。
   - **灵活性**：回调提供了一种方式，使得函数可以在不同的时间点和上下文中执行特定的代码，增加了代码的灵活性。

但是要注意⚠️：**回调地狱**：如果过度使用或不正确使用回调函数，可能会导致代码难以理解和维护，这种情况通常被称为 "回调地狱"（Callback Hell）

### main函数

按照实验原理中的“数据包捕获函数“部分可以得出：

```
char errbuf[PCAP_ERRBUF_SIZE];//用于存储任何与pcap函数相关的错误消息
pcap_t *handle;// pcap_t是一个结构体，用于存储抓包的会话信息

// 获取本地机器上所有网络设备的列表
pcap_if_t *alldevs;// pcap_if_t是一个结构体，用于存储所有可用的网络设备列表
if (pcap_findalldevs(&alldevs, errbuf) == -1) {
    //pcap_findalldevs 函数获取本地机器上所有可用的网络设备列表，并将其存储在 alldevs 中
    printf("Error finding devices: %s\n", errbuf);
    return 1;
}
if (alldevs == NULL) {
    printf("No devices found.\n");
    return 1;
}
char *dev;
dev = alldevs->name;

/**
 * 使用 pcap_open_live 函数打开指定的网络设备并开始监听。
 * BUFSIZ 是捕获的最大字节数
 * 1 表示将设备设置为混杂模式，1000 是读取超时（以毫秒为单位）。
*/
handle = pcap_open_live(dev, BUFSIZ, 1, 1000, errbuf);
if (handle == NULL) {
    printf("Error opening device: %s\n", errbuf);
    return 1;
}
//这是使用wireshark的测试用例
// handle = pcap_open_offline("test.pcap", errbuf);
// if(handle == NULL){
//     printf("Error opening file: %s\n", errbuf);
//     return 1;
// }

// 检查数据链路层是否为以太网
if (pcap_datalink(handle) != DLT_EN10MB) {
    printf("Device does not provide Ethernet headers.\n");
    pcap_close(handle);
    return 1;
}
```

```
    /** 捕获数据包并调用指定的回调函数packet_handler进行处理。
     *  pcap_t *p，该结构包含了捕获数据包所需的所有信息
     * int cnt：整数，指定了要捕获的数据包数量。如果设置为负数，pcap_loop 将会无限循环，直到发生错误或者调用 pcap_breakloop。
     * pcap_handler callback:回调函数，它会在每个捕获到的数据包上被调用。这个回调函数必须具有 pcap_handler 类型的签名
     *
     */
    pcap_loop(handle, 0, packet_handler, NULL);

    // 关闭已打开的网络设备并释放资源
    pcap_close(handle);
    pcap_freealldevs(alldevs);
    return 0;
```

## 五、结果验证

```
--------------捕获到一个数据包--------------
===========开始解析以太网帧数据包========
目的MAC地址： c8:89:f3:ee:dd:e0
源MAC地址： 00:00:5e:00:01:08
以太网类型： 0800
以太网类型： IP
===========开始解析IP层数据包==========
版本号： 4
首部长度： 20
服务类型： 0
优先级： Routine,数据包不需要特殊处理。
总长度： 52
标识： 2568
标志： 010
对应标志：保留位、DF、MF。
片偏移： 0
生存时间： 54
协议编号： 6
协议类型： TCP
首部校验和： 52154
源IP地址： 223.166.17.64
目的IP地址： 10.136.115.147
===========完成IP层数据包解析========

===========完成以太网帧数据包解析========
```

```
Ethernet II, Src: IETF-VRRP-VRID_08 (00:00:5e:00:01:08), Dst: Apple_ee:dd:e0 (c8:89:f3:ee:dd:e0)
  Destination: Apple_ee:dd:e0 (c8:89:f3:ee:dd:e0)
    Address: Apple_ee:dd:e0 (c8:89:f3:ee:dd:e0)
    .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
    .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  Source: IETF-VRRP-VRID_08 (00:00:5e:00:01:08)
    Address: IETF-VRRP-VRID_08 (00:00:5e:00:01:08)
    .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
    .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 223.166.17.64, Dst: 10.136.115.147
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    0000 00.. = Differentiated Services Codepoint: Default (0)
    .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
  Total Length: 52
  Identification: 0x0a08 (2568)
  010. .... = Flags: 0x2, Don't fragment
    0... .... = Reserved bit: Not set
    .1.. .... = Don't fragment: Set
    ..0. .... = More fragments: Not set
  ...0 0000 0000 0000 = Fragment Offset: 0
  Time to Live: 54
  Protocol: TCP (6)
  Header Checksum: 0xcbba [validation disabled]
  [Header checksum status: Unverified]
  Source Address: 223.166.17.64
  Destination Address: 10.136.115.147
```

注意首部校验和：$(CBBA)_{16}=(12 \times 16^3) + (11 \times 16^2) + (11 \times 16^1) + (10 \times 16^0) = (52154)_{10}$，是正确的。

## 六、其他补充

**数据链路类型**

```
#define DLT_NULL  0 /* BSD loopback encapsulation */
#define DLT_EN10MB  1 /* Ethernet (10Mb) */
#define DLT_EN3MB 2 /* Experimental Ethernet (3Mb) */
#define DLT_AX25  3 /* Amateur Radio AX.25 */
#define DLT_PRONET  4 /* Proteon ProNET Token Ring */
#define DLT_CHAOS 5 /* Chaos */
#define DLT_IEEE802 6 /* 802.5 Token Ring */
#define DLT_ARCNET  7 /* ARCNET, with BSD-style header */
#define DLT_SLIP  8 /* Serial Line IP */
#define DLT_PPP   9 /* Point-to-point Protocol */
#define DLT_FDDI  10  /* FDDI */
```

这段代码定义了一组宏（macros），这些宏与网络数据包捕获库（例如 pcap）中的数据链路类型相关。数据链路类型是指网络层协议在网络上通信时所使用的底层协议。在网络分析或数据包捕获的上下文中，数据链路类型用于指明数据包的格式和内容。

每个宏都代表了一种特定的数据链路类型，并为其分配了一个唯一的数字值。这些数字值通常在程序中用于识别和处理不同类型的数据链路。下面是这些宏的解释：

1. **DLT_NULL (0)**：BSD loopback encapsulation，用于循环回接测试，不涉及实际的网络接口。
2. **DLT_EN10MB (1)**：10Mb Ethernet，标准的以太网协议，是最常见的局域网技术。
3. **DLT_EN3MB (2)**：Experimental 3Mb Ethernet，一个实验性的以太网协议，速率为3Mbps。
4. **DLT_AX25 (3)**：Amateur Radio AX.25，无线电通信中使用的协议。
5. **DLT_PRONET (4)**：Proteon ProNET Token Ring，Proteon公司的令牌环网络协议。
6. **DLT_CHAOS (5)**：Chaos，一个早期的网络协议。
7. **DLT_IEEE802 (6)**：802.5 Token Ring，IEEE定义的令牌环网络协议。
8. **DLT_ARCNET (7)**：ARCNET，另一种早期的网络协议，使用BSD风格的头部。
9. **DLT_SLIP (8)**：Serial Line IP，一种通过串行线路传送IP数据包的简单协议。
10. **DLT_PPP (9)**：Point-to-point Protocol，一种用于点对点连接传送网络数据包的协议。
11. **DLT_FDDI (10)**：FDDI (Fiber Distributed Data Interface)，一种基于光纤的高速网络协议。