目录

实验指导书：IP 数据报的转发及收发

1. 实验目的

- 了解原始套接字的基本概念和使用方法。
- 掌握路由器进行 IP 数据报转发的基本原理。
- 实现于原始套接字的 IP 数据报的发送和接收
- 实现基于原始套接字的 IP 数据报转发，包括 AF_INET 和 AF_PACKET 原始套接字的应用。

2. 实验环境

- **操作系统**：Linux
- **编程语言**：C
- **工具**：GCC 编译器、tcpdump、Wireshark

3. 实验内容

(1) 使用虚拟机实现多主机间的 UDP 数据报收发及转发

利用虚拟机搭建实验环境，掌握 Linux 下的 Socket 网络编程。

选做 1：改进程序，示例程序只实现了一个数据包（携带 1 条消息）的发、转、收过程，要求实现每条消息由**控制台输入**，并且**不限制**发送消息的数目。

(2) 基于单网口主机的 IP 数据转发及收发

在局域网中，**模拟** IP 数据报的路由转发过程。通过原始套接字实现了完整的数据封装过程，实现了 UDP 头部、IP 头部、MAC 帧头部的构造。

选做 2：扩展实验的网络规模，由原始方案中 3 台主机增加到**不少于 5 台**主机，共同完成 IP 数据报转发及收发过程，要求采用**转发表**改进示例程序，增加程序通用性。

(3) 基于双网口主机的路由转发

构造了静态路由表，并实现了不同子网间的 IP 数据报查表转发过程。

选做 3：通过**完善路由表**，改进示例程序实现**双向传输**。

3.1 路由器进行 IP 数据报转发的基本原理

- **路由器的工作原理**：
  - 路由器通过查找路由表来决定数据包的转发路径。
  - 路由表包含网络地址、子网掩码、网关地址和网络接口等信息。
  - 数据包到达路由器时，路由器解析 IP 头部，查找路由表，确定下一跳地

址，并将数据包转发到相应的网络接口。

3.2 示例程序

让我们详细说明如何捕获数据报并查找路由表以确定下一跳。

- **捕获数据报**

要捕获数据报，我们需要使用原始套接字。以下是如何创建和使用原始套接字来捕获 IP 数据报的详细步骤：

- **创建原始套接字**：

```
int sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
if (sockfd < 0) {
    perror("Socket creation failed");
    return 1;
}
```

- **捕获数据报： 使用 recvfrom() 函数从网络接口捕获数据报。**

```
struct sockaddr saddr;
unsigned char *buffer = (unsigned char *)malloc(BUFFER_SIZE);
int data_size = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, &saddr, (socklen_t
*)sizeof(saddr));
if (data_size < 0) {
    perror("Recvfrom error");
    return 1;
}
```

- **查找路由表**

在转发数据报之前，我们需要查找路由表以确定下一跳路由器。以下是如何实现这一过程的步骤：

- **解析 IP 头部：**

```
struct iphdr *ip_header = (struct iphdr *)(buffer + sizeof(struct ethhdr));
```

- **查找路由表： 这里假设我们有一个简单的路由表结构和查找函数。**

```
struct route_entry {
    uint32_t dest;
    uint32_t gateway; //下一跳
    uint32_t netmask;
    char interface[IFNAMSIZ]; //接口
};

struct route_entry *lookup_route(uint32_t dest_ip) {
```

```
    // 假设我们有一个全局的路由表数组 route_table 和条目数 route_table_size
    for (int i = 0; i < route_table_size; i++) {
        if ((dest_ip & route_table[i].netmask) == (route_table[i].dest &
route_table[i].netmask)) {
            return &route_table[i];
        }
    }
    return NULL; // 未找到匹配的路由
}
```

获取下一跳:

```
struct route_entry *route = lookup_route(ip_header->daddr);
if (route == NULL) {
    fprintf(stderr, "No route to host\n");
    return 1;
}
```

- **转发数据报**

修改数据报: 修改 TTL 并重新计算校验和。

```
            // 修改 TTL
            ip_header->ttl -= 1;
            ip_header->check = 0;
            ip_header->check = checksum((unsigned short *)ip_header, ip_header->ihl * 4);

            // 发送数据包到目的主机
            struct ifreq ifr,ifr_mac;
            struct sockaddr_ll dest;

            // 获取网卡接口索引
            memset(&ifr, 0, sizeof(ifr));
            snprintf(ifr.ifr_name, sizeof(ifr.ifr_name), route->interface);
            if (ioctl(sockfd, SIOCGIFINDEX, &ifr) < 0) {
                perror("ioctl");
                return 1;
            }

            // 获取网卡接口 MAC 地址
            memset(&ifr_mac, 0, sizeof(ifr_mac));
            snprintf(ifr_mac.ifr_name, sizeof(ifr_mac.ifr_name), route->interface);
            if (ioctl(sockfd, SIOCGIFHWADDR, &ifr_mac) < 0) {
                perror("ioctl");
                return 1;
            }
```

```
            // 设置目标 MAC 地址（假设目标地址已知,此处做了简化处理，实际上，如果存在“下一跳”，应该利用 ARP 协议获得 route->gateway 的 MAC 地址，如果是“直接交付”的话，也应使用 ARP 协议获得目的主机的 MAC 地址。）
            unsigned char target_mac[ETH_ALEN] = {0x00, 0x0c, 0x29, 0x48, 0xd3, 0xf7}; // 替换为实际的目标 MAC 地址

            memset(&dest, 0, sizeof(dest));
            dest.sll_ifindex = ifr.ifr_ifindex;
            dest.sll_halen = ETH_ALEN;
            memcpy(dest.sll_addr, target_mac, ETH_ALEN);

            // 构造新的以太网帧头
            memcpy(eth_header->h_dest, target_mac, ETH_ALEN); // 目标 MAC 地址
            memcpy(eth_header->h_source, ifr_mac.ifr_hwaddr.sa_data, ETH_ALEN); // 源 MAC 地址
            eth_header->h_proto = htons(ETH_P_IP); // 以太网类型为 IP

            printf("Interface name: %s, index: %d\n", ifr.ifr_name, ifr.ifr_ifindex);
            if (sendto(sockfd, buffer, data_size, 0, (struct sockaddr *)&dest, sizeof(dest)) < 0) {
                perror("Sendto error");
                return 1;
            }
```

4. 实验步骤

4.1 使用虚拟机实现多主机间的 UDP 数据报收发及转发

采用 UDP 原始套接字，将 UDP 数据报由 A 主机发送给 B 主机，再由 B 主机转发给 C 主机，完成了一条消息的传送。注意，此实现方案，UDP 数据报的传输过程中，目的 IP 地址发生了变化，因此属于代理转发的场景。

此实验内容目的在于通过多主机的数据收发，获得更接近于实际的开发环境。

4.1.1 实验环境

使用虚拟机软件（如 VirtualBox 或 VMware）在 Windows 笔记本电脑上创建 Linux 虚拟机，从而模拟出你所需的网络环境。以下是一个简单的步骤指南：

1. **安装虚拟机软件**：下载并安装 VirtualBox 或 VMware Workstation Player。
2. **下载 Linux 发行版**：选择一个适合的 Linux 发行版（如 Ubuntu 或 Debian），并下载 ISO 文件。
3. **创建虚拟机**（客户机）：在虚拟机软件中创建一个新的虚拟机，并指定下载的

Linux ISO 文件作为引导盘。

4. **配置网络**：为虚拟机配置网络设置，使其可以与其他主机进行通信。可以选择“桥接模式”（Bridged Adapter），这样确保虚拟机和物理机都连接到同一个局域网，这样虚拟机就可以像独立的设备一样连接到网络。

5. **设置路由转发(可选)**：在虚拟机中安装和配置路由软件（如 iptables 或 firewall），以实现路由转发功能。注意，具体是否开户 Linux 主机的路由转发功能，视应用场景而定。

6. **部署实验环境**：在 Linux 虚拟机中安装所需的网络编程实验软件和工具。

验证网络连接：

1. **获取 IP 地址：**
   - 在 windows 物理机上打开命令提示符或终端，使用命令 ipconfig 查看物理机的 IP 地址。
   - 在虚拟机中打开终端，使用以下命令查看虚拟机的 IP 地址。虚拟机使用“桥接模式”时，一般会通过 DHCP 协议获得与物理机同一子网的 IP 地址。

```
Ip a
```

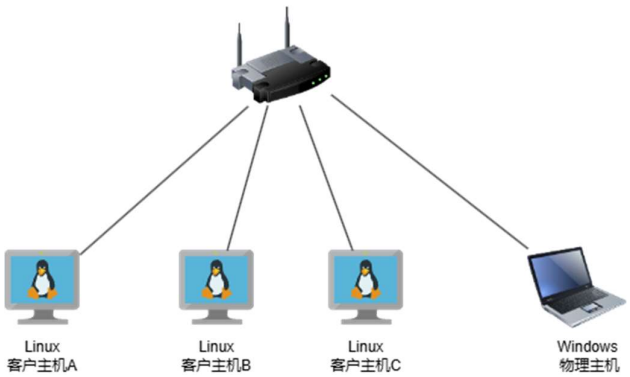
2. **测试连接：**
   - 在虚拟机中，使用 ping 命令测试是否能连接到物理机：

```
ping 物理机的 IP 地址
```

   - 在物理机中，使用 ping 命令测试是否能连接到虚拟机：

```
ping 虚拟机的 IP 地址
```

在创建一台 Linux 虚拟机之后，通过虚拟机克隆可以得到另两台 Linux 虚拟机，这样得到了下图所示的实验环境。



三台 Linux 虚拟机（客户机）与物理主机上处于同一子网中。

4.1.2 示例程序

1. 准备工作

- 安装必要的软件包：sudo apt-get install gcc
- 确认网络配置正常，确保主机间能正常通信
- **注意：Linux 主机要确保防火墙没有阻止 UDP 流量**

2. 发送 IP 数据报

- 创建一个文件 send_ip.c
- 编写以下代码以实现 IP 数据报的发送功能：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main() {
    int sockfd;
    struct sockaddr_in dest_addr;
    char *message = "Hello, this is a UDP datagram!";
    int port = 12345;  // 目标端口号

    // 创建 UDP 套接字
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
        perror("socket");
        return 1;
    }

    // 目标地址
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(port);
    dest_addr.sin_addr.s_addr = inet_addr("目标 IP 地址");

    // 发送数据报
    if (sendto(sockfd, message, strlen(message), 0, (struct sockaddr *)&dest_addr,
sizeof(dest_addr)) < 0) {
        perror("sendto");
        return 1;
    }

    printf("Datagram sent.\n");
    close(sockfd);
    return 0;
```

```
}
```

- 使用 GCC 编译并运行程序：gcc -o send_ip send_ip.c，运行：./send_ip

3. 转发 IP 数据报

- 创建一个文件 forward_ip.c
- 编写以下代码以实现 IP 数据报的转发功能：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main() {
    int sockfd;
    struct sockaddr_in src_addr, dest_addr, my_addr;
    char buffer[1024];
    socklen_t addr_len;
    int src_port = 12345;  // 原始端口号
    int dest_port = 54321;  // 目标端口号（接收程序的端口号）

    // 创建 UDP 套接字
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
        perror("socket");
        return 1;
    }

    // 本地地址
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(src_port);
    my_addr.sin_addr.s_addr = INADDR_ANY;

    // 绑定套接字到本地地址
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr)) < 0) {
        perror("bind");
        return 1;
    }

    // 接收数据报
    addr_len = sizeof(src_addr);
    if (recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)&src_addr,
&addr_len) < 0) {
```

```
        perror("recvfrom");
        return 1;
    }

    printf("Datagram received: %s\n", buffer);

    // 修改目标地址为接收程序主机的 IP 地址
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(dest_port);
    dest_addr.sin_addr.s_addr = inet_addr("接收程序的 IP 地址");  // 替换为接收程序主机的实际 IP 地址

    // 发送数据报
    if (sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr *)&dest_addr, sizeof(dest_addr)) < 0) {
        perror("sendto");
        return 1;
    }

    printf("Datagram forwarded.\n");
    close(sockfd);
    return 0;
}
```

- 防火墙开放相应的 UDP 端口

```
sudo ufw allow 12345/udp
```

- 使用 GCC 编译并运行程序：gcc -o forward_ip forward_ip.c，然后运行：./forward_ip

4. 接收 IP 数据报

- 创建一个文件 recv_ip.c
- 编写以下代码以实现 IP 数据报的接收功能：

```
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <sys/socket.h>

int main() {
    int sockfd;
    struct sockaddr_in src_addr, my_addr;
    char buffer[1024];
```

```
    socklen_t addr_len;
    int port = 54321;  // 修改后的接收端口号

    // 创建 UDP 套接字
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
        perror("socket");
        return 1;
    }

    // 本地地址
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(port);
    my_addr.sin_addr.s_addr = INADDR_ANY;

    // 绑定套接字到本地地址
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr)) < 0) {
        perror("bind");
        return 1;
    }

    // 接收数据报
    addr_len = sizeof(src_addr);
    if (recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)&src_addr,
&addr_len) < 0) {
        perror("recvfrom");
        return 1;
    }

    printf("Datagram received: %s\n", buffer);
    close(sockfd);
    return 0;
}
```

- 防火墙开放相应的 UDP 端口

```
sudo ufw allow 54321/udp
```

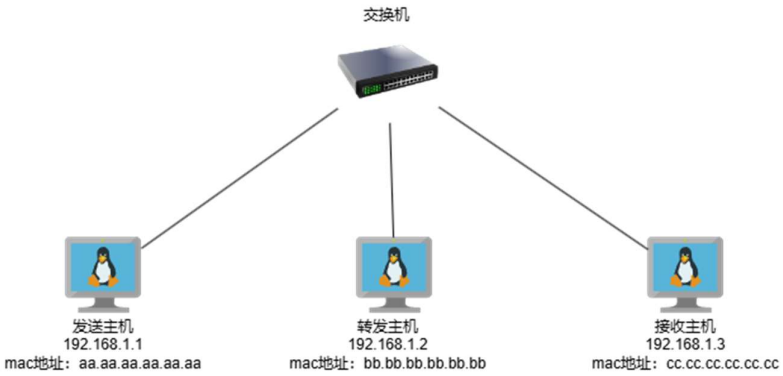
- 使用 GCC 编译并运行程序：gcc -o recv_ip recv_ip.c，然后运行：./recv_ip

4.2 基于单网口主机的 IP 数据报转发及收发

4.2.1 实验环境

路由主机只有一个网络接口，用于转发源主机和目的主机之间的数据包。

网络拓扑



说明

1. **源主机**（192.168.1.1/24）：发送数据包到目的主机（192.168.1.3）。
2. **转发主机**（192.168.1.2/24）：接收并转发数据包。
3. **目的主机**（192.168.1.3/24）：接收数据包。

注：这个实验环境，可采用与实验内容 1 相同的实验环境。

这个实验使用了一个特殊的处理技巧，因为源主机（192.168.1.1）发送的 IP 数据包的目的地址为接收主机（192.18.1.3），由于二者处于一个子网中，这个 IP 数据报将直接交付给目的主机，因而正常不会经由转发主机（192.18.1.2）来进行转发，鉴于此种情况，我们将直接处理 IP 数据报的以太网帧封装，将 IP 数据报交给转发主机。

在发送主机上，首先构造出 IP 数据报，其头部源 IP 地址为 192.168.1.1，目的 IP 地址为 192.168.1.3，然后构造用于封装该 IP 数据报的以太网帧，该帧头部的源 MAC 地址为 aa.aa.aa.aa.aa.aa，但其目的 MAC 地址为转发主机的 MAC 地址 bb.bb.bb.bb.bb.bb（而不是接收主机的 MAC 地址 cc.cc.cc.cc.cc.cc），通过这种方法，将去往接收主机(192.168.1.3)的 IP 数据报交给转发主机(192.168.1.2)进行转发。

实际上路由器转发的过程与此类似，在将 IP 数据报封装到链路层的 MAC 帧时，通过源 MAC 地址和目的 MAC 地址的变化，来实现 IP 数据报从源主机到目的主机及传输路径上各路由器间的跳转传输。

数据包处理流程

1. **源主机发送数据包**：
    - 源主机发送的数据包通过交换机到达路由器。
    - 数据包的 IP 头部源地址为 192.168.1.2，目的地址为 192.168.1.3。
    - 数据包在封装到下层的以太网帧的时候，进行了特殊处理，即构造的以太网帧的目的 MAC 地址为路由主机 192.168.1.2 的 MAC 地址 bb.bb.bb.bb.bb.bb，以使数据报被交给路由主机，进而转发到目的主机。

2. **路由器处理数据包**：
    - 路由器接收到数据包后，解析 IP 头部信息。
    - 修改 TTL（生存时间）字段，并重新计算校验和。
    - 根据路由表(示例程序未实现路由表)决定将数据包转发到目的主机。
3. **目的主机接收数据包**：
    - 目的主机接收到数据包并处理。

4.2.2 三个主机的示例程序

1. 发送主机程序（源主机）

这个程序将构造一个以太网帧，其中 IP 头的源地址和目的地址分别为 192.168.1.2 和 192.168.1.3，以太网头的目的 MAC 地址为路由主机的 MAC 地址。

示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/ether.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <linux/if_packet.h>

#define DEST_MAC0 0x00
#define DEST_MAC1 0x0c
#define DEST_MAC2 0x29
#define DEST_MAC3 0x3e
#define DEST_MAC4 0x1e
#define DEST_MAC5 0x4c

#define ETHER_TYPE 0x0800
#define BUFFER_SIZE 1518
#define UDP_SRC_PORT 12345
#define UDP_DST_PORT 12345

unsigned short checksum(void *b, int len) {
```

```
    unsigned short *buf = b;
    unsigned int sum = 0;
    unsigned short result;

    for (sum = 0; len > 1; len -= 2)
        sum += *buf++;
    if (len == 1)
        sum += *(unsigned char *)buf;
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    result = ~sum;
    return result;
}

int main() {
    int sockfd;
    struct ifreq if_idx, if_mac;
    struct sockaddr_ll socket_address;
    char buffer[BUFFER_SIZE];

    // 创建原始套接字
    if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1) {
        perror("socket");
        return 1;
    }

    // 获取接口索引
    memset(&if_idx, 0, sizeof(struct ifreq));
    strncpy(if_idx.ifr_name, "eth0", IFNAMSIZ-1);
    if (ioctl(sockfd, SIOCGIFINDEX, &if_idx) < 0) {
        perror("SIOCGIFINDEX");
        return 1;
    }

    // 获取接口 MAC 地址
    memset(&if_mac, 0, sizeof(struct ifreq));
    strncpy(if_mac.ifr_name, "eth0", IFNAMSIZ-1);
    if (ioctl(sockfd, SIOCGIFHWADDR, &if_mac) < 0) {
        perror("SIOCGIFHWADDR");
        return 1;
    }

    // 构造以太网头
    struct ether_header *eh = (struct ether_header *) buffer;
```

```
    eh->ether_shost[0] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[0];
    eh->ether_shost[1] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[1];
    eh->ether_shost[2] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[2];
    eh->ether_shost[3] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[3];
    eh->ether_shost[4] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[4];
    eh->ether_shost[5] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[5];
    eh->ether_dhost[0] = DEST_MAC0;
    eh->ether_dhost[1] = DEST_MAC1;
    eh->ether_dhost[2] = DEST_MAC2;
    eh->ether_dhost[3] = DEST_MAC3;
    eh->ether_dhost[4] = DEST_MAC4;
    eh->ether_dhost[5] = DEST_MAC5;
    eh->ether_type = htons(ETHER_TYPE);

    // 构造 IP 头
    struct iphdr *iph = (struct iphdr *) (buffer + sizeof(struct ether_header));
    iph->ihl = 5;
    iph->version = 4;
    iph->tos = 0;
    iph->tot_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr) + strlen("Hello,
this is a test message."));
    iph->id = htonl(54321);
    iph->frag_off = 0;
    iph->ttl = 255;
    iph->protocol = IPPROTO_UDP;
    iph->check = 0;
    iph->saddr = inet_addr("192.168.1.2");
    iph->daddr = inet_addr("192.168.1.3");
    iph->check = checksum((unsigned short *)iph, sizeof(struct iphdr));

    // 构造 UDP 头
    struct udphdr *udph = (struct udphdr *) (buffer + sizeof(struct ether_header) +
sizeof(struct iphdr));
    udph->source = htons(UDP_SRC_PORT);
    udph->dest = htons(UDP_DST_PORT);
    udph->len = htons(sizeof(struct udphdr) + strlen("Hello, this is a test message."));
    udph->check = 0; // UDP 校验和可选

    // 填充数据
    char *data = (char *) (buffer + sizeof(struct ether_header) + sizeof(struct iphdr) +
sizeof(struct udphdr));
    strcpy(data, "Hello, this is a test message.");

    // 设置 socket 地址结构
```

```
    socket_address.sll_ifindex = if_idx.ifr_ifindex;
    socket_address.sll_halen = ETH_ALEN;
    socket_address.sll_addr[0] = DEST_MAC0;
    socket_address.sll_addr[1] = DEST_MAC1;
    socket_address.sll_addr[2] = DEST_MAC2;
    socket_address.sll_addr[3] = DEST_MAC3;
    socket_address.sll_addr[4] = DEST_MAC4;
    socket_address.sll_addr[5] = DEST_MAC5;

    // 发送数据包
    if (sendto(sockfd, buffer, sizeof(struct ether_header) + sizeof(struct iphdr) +
sizeof(struct udphdr) + strlen("Hello, this is a test message."), 0, (struct
sockaddr*)&socket_address, sizeof(struct sockaddr_ll)) < 0) {
        perror("sendto");
        return 1;
    }

    close(sockfd);
    return 0;
}
```

说明

1. **创建原始套接字**：使用 socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))创建一个原始套接字。
2. **获取接口索引**：使用 ioctl 获取网络接口的索引。
3. **获取接口 MAC 地址**：使用 ioctl 获取网络接口的 MAC 地址。
4. **构造以太网头**：设置源 MAC 地址和目的 MAC 地址（路由主机的 MAC 地址）。
5. **构造 IP 头**：设置源 IP 地址和目的 IP 地址。
6. **构造 UDP 头**：设置源端口地址和目的端口地址。
7. **设置 socket 地址结构**：配置发送数据包的目标地址。
8. **发送数据包**：使用 sendto 函数发送数据包。

这个程序将构造并发送一个以太网帧，其中包含指定的 IP 头和以太网头。

2. 路由转发程序（中间主机）

这个程序将捕获数据包，修改 TTL 并转发到目的主机：

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>
#include <netinet/ether.h>
#include <sys/socket.h>
#include <unistd.h>
#include <linux/if_packet.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <time.h>

#define BUFFER_SIZE 65536

unsigned short checksum(void *b, int len) {
    unsigned short *buf = b;
    unsigned int sum = 0;
    unsigned short result;

    for (sum = 0; len > 1; len -= 2)
        sum += *buf++;
    if (len == 1)
        sum += *(unsigned char *)buf;
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    result = ~sum;
    return result;
}

int main() {
    int sockfd;
    struct sockaddr saddr;
    unsigned char *buffer = (unsigned char *)malloc(BUFFER_SIZE);

    sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    while (1) {
        int saddr_len = sizeof(saddr);
```

```
        int data_size = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, &saddr, (socklen_t
*)&saddr_len);
        if (data_size < 0) {
            perror("Recvfrom error");
            return 1;
        }

        struct ethhdr *eth_header = (struct ethhdr *)buffer;
        struct iphdr *ip_header = (struct iphdr *)(buffer + sizeof(struct ethhdr));
        char src_ip[INET_ADDRSTRLEN];
        char dest_ip[INET_ADDRSTRLEN];

        inet_ntop(AF_INET, &(ip_header->saddr), src_ip, INET_ADDRSTRLEN);
        inet_ntop(AF_INET, &(ip_header->daddr), dest_ip, INET_ADDRSTRLEN);

        if (strcmp(src_ip, "192.168.1.1") == 0 && strcmp(dest_ip, "192.168.1.3") == 0) {
            // 获取当前系统时间
            time_t rawtime;
            struct tm *timeinfo;
            char time_str[100];

            time(&rawtime);
            timeinfo = localtime(&rawtime);

            // 格式化时间字符串
            strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", timeinfo);

            // 打印信息
            printf("[%s] Captured packet from %s to %s\n", time_str, src_ip, dest_ip);

            // 修改 TTL
            ip_header->ttl -= 1;
            ip_header->check = 0;
            ip_header->check = checksum((unsigned short *)ip_header, ip_header->ihl * 4);

            // 发送数据包到目的主机
            struct ifreq ifr,ifr_mac;
            struct sockaddr_ll dest;

            // 获取网卡接口索引
            memset(&ifr, 0, sizeof(ifr));
            snprintf(ifr.ifr_name, sizeof(ifr.ifr_name), "eth0");
            if (ioctl(sockfd, SIOCGIFINDEX, &ifr) < 0) {
                perror("ioctl");
```

```
                return 1;
            }

            // 获取网卡接口 MAC 地址
            memset(&ifr_mac, 0, sizeof(ifr_mac));
            snprintf(ifr_mac.ifr_name, sizeof(ifr_mac.ifr_name), "eth0");
            if (ioctl(sockfd, SIOCGIFHWADDR, &ifr_mac) < 0) {
                perror("ioctl");
                return 1;
            }

            // 设置目标 MAC 地址（假设目标地址已知）
            unsigned char target_mac[ETH_ALEN] = {0x00, 0x0c, 0x29, 0x48, 0xd3, 0xf7}; //
替换为实际的目标 MAC 地址

            memset(&dest, 0, sizeof(dest));
            dest.sll_ifindex = ifr.ifr_ifindex;
            dest.sll_halen = ETH_ALEN;
            memcpy(dest.sll_addr, target_mac, ETH_ALEN);

            // 构造新的以太网帧头
            memcpy(eth_header->h_dest, target_mac, ETH_ALEN); // 目标 MAC 地址
            memcpy(eth_header->h_source, ifr_mac.ifr_hwaddr.sa_data, ETH_ALEN); // 源 MAC
地址
            eth_header->h_proto = htons(ETH_P_IP); // 以太网类型为 IP

            printf("Interface name: %s, index: %d\n", ifr.ifr_name, ifr.ifr_ifindex);
            if (sendto(sockfd, buffer, data_size, 0, (struct sockaddr *)&dest,
sizeof(dest)) < 0) {
                perror("Sendto error");
                return 1;
            }
            printf("Datagram forwarded.\n");
        } else {
            printf("Ignored packet from %s to %s\n", src_ip, dest_ip);
        }
    }

    close(sockfd);
    free(buffer);
    return 0;
}
```

3. 接收主机程序（目的主机）

这个程序将接收数据包并打印内容：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>

#define PORT 12345

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    char buffer[1024];

    // 创建 UDP 套接字
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    // 绑定套接字到端口
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        return 1;
    }

    // 接收数据包
    int recv_len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr
*)&client_addr, &addr_len);
    if (recv_len < 0) {
        perror("Recvfrom failed");
        return 1;
    }
```

```
    buffer[recv_len] = '\0';
    printf("Received message: %s\n", buffer);

    close(sockfd);
    return 0;
}
```

运行步骤（注意三个程序的启动顺序）

1. 在源主机上运行发送程序。
2. 在路由器上运行转发程序。
3. 在目的主机上运行接收程序。

这样，源主机发送的数据包将通过单臂路由器转发到目的主机，目的主机将接收到并打印数据包的内容。

4.3 基于双网口主机的路由转发 实验五、六

4.3.1 实验环境

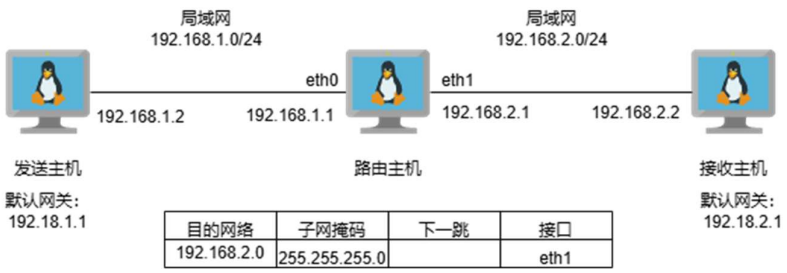
展示源主机发送数据包到路由器，再由路由器转发数据包到目的主机的过程，可以按照以下步骤进行:

1. 准备设备和软件

- **三台计算机**：一台作为源主机，一台作为路由器，一台作为目的主机。
- **网络交换机**：用于连接这些计算机。
- **操作系统**：建议使用 Linux（如 Debian）来方便使用原始套接字。
- **网络工具**：如 Wireshark（Tshark，命令行界面的 Wireshark），用于捕获和分析网络数据包。

2. 配置网络

示例网络拓扑图



1. **连接设备**：
    - 将源主机、路由主机和目的主机通过交换机互相连接，形成示例网络拓扑结构。
    - 确保每台设备的相应接口都有一个唯一的 IP 地址。
2. **配置 IP 地址**：
    - 源主机：192.168.1.2/24
    - 路由主机（两个接口）：
        - 接口 1（连接源主机）：192.168.1.1/24
        - 接口 2（连接目的主机）：192.168.2.1/24
    - 目的主机：192.168.2.2/24

3. 编写和部署转发程序

在路由器上编写并运行转发程序。确保程序能够捕获数据包、查找路由表并将数据包转发到下一跳。

4. 配置路由表

在源主机和目的主机上配置静态路由，以确保数据包能够正确转发：

- **源主机**：

```
sudo ip route add 192.168.2.0/24 via 192.168.1.1
```

- **路由器**： 路由器上已经有两个接口，分别连接源主机和目的主机，不需要额外配置。
- **目的主机**：

```
sudo ip route add 192.168.1.0/24 via 192.168.2.1
```

5. 测试网络连接

1. **在源主机上发送数据包**： 使用 ping 命令或编写一个简单的程序发送数据包到目的主机。

```
ping 192.168.2.2
```

2. **捕获和分析数据包**： 使用 Wireshark 在路由器上捕获数据包，验证数据包是否被正确转发。

6. 验证转发过程

1. **检查源主机**： 确认源主机发送的数据包到达路由器。

2. **检查路由器**： 确认路由器捕获数据包，修改 TTL 并转发到目的主机。
3. **检查目的主机**： 确认目的主机接收到数据包并返回响应。

4.3.2 基于虚拟机的实验环境搭建

配置双网口虚拟机的步骤可能会因虚拟机软件和操作系统而有所不同。以下是一些常见的步骤：

1. **创建虚拟机：**首先，确保你已经安装了虚拟机软件（如 VMware、VirtualBox 等）并创建了一个新的虚拟机。
2. **添加第二个网口：**在虚拟机设置中，添加一个新的网络接口。通常可以在“网络”或“硬件”选项卡中找到这个选项。
3. **配置网络：**为新的网口配置网络设置，例如 IP 地址、子网掩码、网关等。确保这些设置与你的物理网络环境兼容。
4. **启动虚拟机：**启动虚拟机并检查新的网口是否正常工作。你可以使用命令行工具（如 ipconfig 或 ifconfig）来查看网络配置。
5. **测试连接：**确保虚拟机可以通过新的网口访问网络资源，例如浏览网页或 ping 其他设备。

注意：一个双网口主机，拥有两个网卡，通常这两个网络接口**分别属于不同的子网**。如果这两个网络接口，都接入一个网络时，要注意配置，避免网络冲突**。**

1. 配置物理主机

确保所有物理主机（笔记本电脑）连接到同一个 Wi-Fi 网络。

2. 安装虚拟机软件

在每台笔记本电脑上安装 VirtualBox 或 VMware，并创建相应的 Linux 虚拟机。

启动虚拟机，并安装你选择的 Linux 发行版（如 Ubuntu 或 Debian）。

3. 配置虚拟机网络

将每个虚拟机的网络适配器设置为**桥接模式**（Bridged Adapter），确保虚拟机通过 Wi-Fi 连接到同一网络中。

4. 设置双网口虚拟机

在其中一台笔记本电脑上创建一个具有双网口的 Linux 虚拟机，配置如下：

- **网络适配器 1（eth0）**：桥接模式，连接到 192.168.1.0/24 网段。
- **网络适配器 2（eth1）**：桥接模式，连接到 192.168.2.0/24 网段。

配置网络接口

- **使用以下命令检查并配置网络接口：**

```
sudo nano /etc/network/interfaces
```

- **添加以下配置：**

```
auto eth0
iface eth0 inet static
    address 192.168.1.1
    netmask 255.255.255.0
    # 不需要配置 gateway

auto eth1
iface eth1 inet static
    address 192.168.2.1
    netmask 255.255.255.0
    # 不需要配置 gateway
```

5. 配置客户端主机

客户端主机 1

- 设置虚拟机的 IP 地址在 192.168.1.0/24 网段。
- 设置默认网关为双网口虚拟机的 eth0 IP 地址：

```
sudo route add default gw 192.168.1.1
```

客户端主机 2

- 设置虚拟机的 IP 地址在 192.168.2.0/24 网段。
- 设置默认网关为双网口虚拟机的 eth1 IP 地址：

```
sudo route add default gw 192.168.2.1
```

4.3.3 三台主机的程序示例

1. 发送主机程序（源主机）

这个程序将发送一个简单的 UDP 数据包到路由器：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
```

```c
#include <unistd.h>

#define DEST_IP "192.168.2.2"
#define DEST_PORT 12345
#define MESSAGE "Hello, this is a test message."

int main() {
    int sockfd;
    struct sockaddr_in dest_addr;

    // 创建 UDP 套接字
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    // 设置目的地址
    memset(&dest_addr, 0, sizeof(dest_addr));
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(DEST_PORT);
    inet_pton(AF_INET, DEST_IP, &dest_addr.sin_addr);

    // 发送数据包
    if (sendto(sockfd, MESSAGE, strlen(MESSAGE), 0, (struct sockaddr *)&dest_addr,
sizeof(dest_addr)) < 0) {
        perror("Sendto failed");
        return 1;
    }

    printf("Message sent to %s:%d\n", DEST_IP, DEST_PORT);

    close(sockfd);
    return 0;
}
```

2. 路由转发程序（中间主机）

这个程序将捕获数据包，修改 TTL 并转发到目的主机：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>
```

```c
#include <sys/socket.h>
#include <unistd.h>
#include <net/if.h>

#define BUFFER_SIZE 65536

struct route_entry {
    uint32_t dest;
    uint32_t gateway;
    uint32_t netmask;
    char interface[IFNAMSIZ];
};

struct route_entry route_table[] = {
    {inet_addr("192.168.2.0"), inet_addr("192.168.2.1"), inet_addr("255.255.255.0"),
"eth1"}
};
int route_table_size = sizeof(route_table) / sizeof(route_table[0]);

unsigned short checksum(unsigned short *buf, int nwords) {
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (unsigned short)(~sum);
}

struct route_entry *lookup_route(uint32_t dest_ip) {
    for (int i = 0; i < route_table_size; i++) {
        if ((dest_ip & route_table[i].netmask) == (route_table[i].dest &
route_table[i].netmask)) {
            return &route_table[i];
        }
    }
    return NULL;
}

int main() {
    int sockfd;
    struct sockaddr saddr;
    unsigned char *buffer = (unsigned char *)malloc(BUFFER_SIZE);

    sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
```

```
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    while (1) {
        int data_size = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, &saddr, (socklen_t
*)sizeof(saddr));
        if (data_size < 0) {
            perror("Recvfrom error");
            return 1;
        }

        struct iphdr *ip_header = (struct iphdr *)(buffer + sizeof(struct ethhdr));
        printf("Captured packet from %s to %s\n", inet_ntoa(*(struct in_addr
*)&ip_header->saddr), inet_ntoa(*(struct in_addr *)&ip_header->daddr));

        struct route_entry *route = lookup_route(ip_header->daddr);
        if (route == NULL) {
            fprintf(stderr, "No route to host\n");
            continue;
        }

        // 修改 TTL
        ip_header->ttl -= 1;
        ip_header->check = 0;
        ip_header->check = checksum((unsigned short *)ip_header, ip_header->ihl * 4);

        // 发送数据包到目的主机
        struct ifreq ifr,ifr_mac;
        struct sockaddr_ll dest;

        // 获取网卡接口索引
        memset(&ifr, 0, sizeof(ifr));
        snprintf(ifr.ifr_name, sizeof(ifr.ifr_name), route->interface);
        if (ioctl(sockfd, SIOCGIFINDEX, &ifr) < 0) {
            perror("ioctl");
            return 1;
        }

        // 获取网卡接口 MAC 地址
        memset(&ifr_mac, 0, sizeof(ifr_mac));
        snprintf(ifr_mac.ifr_name, sizeof(ifr_mac.ifr_name), route->interface);
        if (ioctl(sockfd, SIOCGIFHWADDR, &ifr_mac) < 0) {
```

```
            perror("ioctl");
            return 1;
        }

        // 设置目标 MAC 地址（假设目标地址已知,此处做了简化处理，实际上，如果查找路由表后，存在"下一跳"，应该利用 ARP 协议获得 route->gateway 的 MAC 地址，如果是"直接交付"的话，也应使用 ARP 协议获得目的主机的 MAC 地址。）
            unsigned char target_mac[ETH_ALEN] = {0x00, 0x0c, 0x29, 0x48, 0xd3, 0xf7}; // 替换为实际的目标 MAC 地址

        memset(&dest, 0, sizeof(dest));
        dest.sll_ifindex = ifr.ifr_ifindex;
        dest.sll_halen = ETH_ALEN;
        memcpy(dest.sll_addr, target_mac, ETH_ALEN);

        // 构造新的以太网帧头
        memcpy(eth_header->h_dest, target_mac, ETH_ALEN); // 目标 MAC 地址
        memcpy(eth_header->h_source, ifr_mac.ifr_hwaddr.sa_data, ETH_ALEN); // 源 MAC 地址
        eth_header->h_proto = htons(ETH_P_IP); // 以太网类型为 IP

        printf("Interface name: %s, index: %d\n", ifr.ifr_name, ifr.ifr_ifindex);
        if (sendto(sockfd, buffer, data_size, 0, (struct sockaddr *)&dest, sizeof(dest)) < 0) {
                perror("Sendto error");
                return 1;
            }
    }

    close(sockfd);
    free(buffer);
    return 0;
}
```

3. 接收主机程序（目的主机）

这个程序将接收数据包并打印内容：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>

#define PORT 12345
```

```c
int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    char buffer[1024];

    // 创建 UDP 套接字
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return 1;
    }

    // 绑定套接字到端口
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        return 1;
    }

    // 接收数据包
    int recv_len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr
*)&client_addr, &addr_len);
    if (recv_len < 0) {
        perror("Recvfrom failed");
        return 1;
    }

    buffer[recv_len] = '\0';
    printf("Received message: %s\n", buffer);

    close(sockfd);
    return 0;
}
```

运行步骤（注意执行顺序）

1. 在源主机上运行发送程序。
2. 在路由器上运行转发程序。
3. 在目的主机上运行接收程序。

这样，源主机发送的数据包将通过路由器转发到目的主机，目的主机将接收到并打印数据包的内容。

5. 实验方式

每位同学独立上机实验，记录并分析每个步骤中的程序输出，思考数据报在网络中传输的路径和延迟。

6. 实验报告

实验报告应包括以下内容：

(1) **实验目的**：简要描述本次实验的目的。

(2) **实验环境**：详细描述实验环境，包括操作系统版本、编程语言和编译器版本、使用的工具等。

(3) **实验内容**：逐项描述实验内容的实现过程，包括代码实现、运行结果、遇到的问题及解决方法。

(4) **实验结果分析**：对实验结果进行分析，包括数据包转发的成功率、性能和准确性。

(5) **实验总结**：总结本次实验的收获和体会，包括对原始套接字和路由器转发机制的理解和掌握，实验过程中遇到的主要问题和解决方法，对网络编程的进一步认识和思考。

(6) **参考文献**：列出在实验过程中参考的文献和资料，包括书籍、论文、在线教程等。

附录. 原始套接字介绍

1. 概述

原始套接字（Raw Socket）原始套接字是一种特殊的套接字类型，也是一种强大的网络编程工具，允许应用程序直接访问网络层（IP 层）数据包或链路层（数据链路层）的数据包，绕过操作系统提供的传输层接口。它不同于标准的 TCP 或 UDP 套接字，能够处理和发送自定义的网络协议数据包。这种套接字通常用于实现新的协议或对现有协议进行低级别的操作。

2. 定义与用途

- **定义**：原始套接字是直接基于网络层（如 IP）的。当使用原始套接字发送数据时，应用程序负责构建完整的协议头。

- **用途**：原始套接字常用于构造和发送自定义的 IP 包，如在 ping、traceroute 等工具中，它们使用 ICMP 协议构建消息[1][2]。

3. 创建原始套接字

创建原始套接字的函数是 socket()，它的原型如下：

int socket(int domain, int type, int protocol);

type 参数指定了套接字的类型。常见的套接字类型包括：

- **SOCK_RAW**：原始套接字，允许直接访问底层协议（如 IP、ICMP、TCP、UDP 等）。
- **SOCK_STREAM**：流套接字，提供面向连接的、可靠的字节流服务（如 TCP）。
- **SOCK_DGRAM**：数据报套接字，提供无连接的、不可靠的数据报服务（如 UDP）。

当指定 type 为 SOCK_RAW 时，创建的即为原始套接字。创建 SOCK_RAW 原始套接字时，socket() 函数的 domain 和 protocol 参数有多种可能的取值，每种取值对应不同的用途。以下是详细的说明：

domain 参数

domain 参数指定了套接字使用的协议族。常见的取值包括：

- **AF_INET**：IPv4 协议族，用于处理 IPv4 地址。

  用途：用于创建处理 IPv4 数据包的原始套接字。例如，创建一个用于发送和接收 ICMP（如 ping 命令）的套接字。

  int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

- **AF_INET6**：IPv6 协议族，用于处理 IPv6 地址。

  用途：用于创建处理 IPv6 数据包的原始套接字。例如，创建一个用于发送和接收 IPv6 ICMP（如 ping6 命令）的套接字。

  int sockfd = socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);

- **AF_PACKET**：数据链路层协议族，用于直接访问网络接口层的数据包（如以太网帧）。

  用途：用于捕获和发送链路层数据包。例如，创建一个用于捕获所有以太网帧的套接字。

  int sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

- **AF_UNIX**：本地通信协议族，用于同一主机上的进程间通信。

  用途：虽然不常见，但可以用于本地进程间通信的原始套接字。

  int sockfd = socket(AF_UNIX, SOCK_RAW, 0);

protocol 参数

protocol 参数指定了使用的具体协议。常见的取值包括：

- **IPPROTO_ICMP**：ICMP 协议，通常用于实现 ping 工具。

  用途：用于发送和接收 ICMP 数据包。

  int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

- **IPPROTO_TCP**：TCP 协议，提供可靠的、面向连接的服务。

  用途：用于发送和接收 TCP 数据包。

  int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);

- **IPPROTO_UDP**：UDP 协议，提供无连接的、不可靠的服务。

  用途：用于发送和接收 UDP 数据包。

  int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);

- **IPPROTO_RAW**：原始 IP 协议，允许应用程序自己构建 IP 头部。

  用途：用于发送（但不能接收）自定义的 IP 数据包。如果需要接收所有 IP 协议的数据包，建议使用 AF_PACKET 套接字与 ETH_P_IP 协议类型。

  int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

- **IPPROTO_ICMPV6**：ICMPv6 协议，通常用于实现 IPv6 的 ping 工具。

  用途：用于发送和接收 ICMPv6 数据包。

  int sockfd = socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);

示例代码

以下是一个创建 ICMP 原始套接字的示例代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```c
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockfd < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    printf("ICMP raw socket created successfully.\n");
    close(sockfd);
    return 0;
}
```

4. 特权需求

由于原始套接字允许直接访问底层协议，并可能被用于伪造数据包，所以它通常需要特殊权限（如 root 权限）[1][2]。

5. 工作方式

原始套接字发送数据时，需要提供完整的传输层头部（如 TCP、UDP 或 ICMP 头部），甚至是 IP 协议的头部。这给了我们控制头部字段的能力，例如伪造源 IP 地址。当使用原始套接字接收数据时，会得到底层协议的完整头部。

- **数据包接收过程**

  当网络接口接收到一个数据包时，数据包会经过以下处理流程：

  1. **网卡驱动**：网卡驱动首先接收到数据包，并将其传递给内核的网络子系统。
  2. **软中断处理**：在软中断上下文中，数据包被传递给 netif_receive_skb()函数进行处理。
  3. **协议类型匹配**：系统会检查数据包的协议类型，并匹配是否有注册的原始套接字。如果匹配成功，数据包会被克隆并交给相应的原始套接字处理。
  4. **协议栈处理**：即使数据包被克隆并交给原始套接字，原始数据包仍会继续在协议栈中进行后续处理[2]。

- **数据包发送过程**

  通过原始套接字发送数据包的过程如下：

  1. **应用层构造数据包**：应用程序通过原始套接字构造并发送数据包，包括自定义的 IP 头部和数据部分。

2. **套接字层处理**：数据包通过 inet_sendmsg()和 raw_sendmsg()函数进行处理。
3. **路由和邻居子系统**：数据包经过路由和邻居子系统的处理，确定发送路径和下一跳。
4. **网卡驱动发送**：最终，数据包被传递给网卡驱动进行实际的发送操作[2]。

- **设置 IP_HDRINCL 选项**

通过设置 IP_HDRINCL 选项，应用程序可以完全控制 IP 头部的内容，包括源 IP 地址、目的 IP 地址、协议类型和校验和等字段。这对于需要自定义 IP 头部的高级网络编程任务非常重要。

6. 用途与限制

1. **用途**：原始套接字经常被用于网络诊断工具（如 ping 和 traceroute）、网络攻击和防御、以及某些类型的网络测试[12]。
2. **限制**：大多数操作系统默认会处理某些协议，这可能会导致原始套接字不能接收到这些协议的数据包。例如，操作系统可能会自动处理 ICMP 回显请求和回显应答，这意味着原始套接字可能无法看到这些数据包[12]。

7. 注意事项

- **错误使用**：由于原始套接字跳过了常规的协议处理，错误的使用可能导致不可预期的网络行为。
- **操作系统保护**：操作系统可能提供了某种形式的保护，以防止滥用原始套接字，例如对其使用进行限制[12]。

8. 跨平台差异

不同的操作系统可能对原始套接字的实现和行为有所不同。例如，Windows 和 Linux 在处理和访问原始套接字时存在细微差别[12]。

9. 数据的分流过程

当数据包到达主机，进入系统协议栈时，协议栈会根据数据包的协议类型和目的地址将其分流到相应的协议实体，包括原始套接字。具体来说，数据包会被复制一份并传递给匹配的原始套接字。以下是详细的过程：

1. **数据包到达网络接口**： 数据包首先到达网络接口（如以太网卡），并被传递到内核的网络协议栈。
2. **链路层处理**： 在链路层，数据包的以太网帧头部被解析，确定数据包的类型（如 IPv4、ARP 等）。
    - 如果存在 AF_PACKET 原始套接字，系统会检查数据包的链路层协议类型

（如 ETH_P_ALL、ETH_P_IP 等）。如果匹配成功，数据包会被克隆一份并传递给相应的 AF_PACKET 原始套接字。

- 处理过程：数据包被传递给 packet_rcv()函数进行处理。packet_rcv()函数会将数据包传递给所有匹配的 AF_PACKET 原始套接字。

3. **网络层处理**： 在网络层，数据包的 IP 头部被解析，确定数据包的协议类型（如 ICMP、TCP、UDP 等）和目的地址。
   - 如果存在 AF_INET 原始套接字，系统会检查数据包的网络层协议类型（如 IPPROTO_TCP、IPPROTO_UDP、IPPROTO_ICMP 等）。如果匹配成功，数据包会被克隆一份并传递给相应的 AF_INET 原始套接字。
   - 处理过程：数据包被传递给 ip_local_deliver_finish()函数进行处理。ip_local_deliver_finish()函数会将数据包传递给所有匹配的 AF_INET 原始套接字。
4. **传输层处理**： 在传输层，数据包的传输层头部（如 TCP、UDP 头部）被解析，确定数据包的源端口和目的端口。
5. 如果数据包的协议类型和端口号匹配某个标准套接字（如 TCP 或 UDP 套接字），数据包会被传递到该套接字。
6. **BPF 过滤器处理**： 如果原始套接字附加了 BPF 过滤器，数据包会先经过 BPF 过滤器的检查，只有通过过滤器的数据包才会被传递到原始套接字[2]。

示例代码

以下是如何创建 AF_PACKET 和 AF_INET 原始套接字的示例代码：

AF_PACKET 原始套接字

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ether.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

int main() {
    int sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    if (sockfd < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }
```

```
    char buffer[2048];
    while (1) {
        int len = recvfrom(sockfd, buffer, sizeof(buffer), 0, NULL, NULL);
        if (len > 0) {
            printf("Received packet of length %d\n", len);
        }
    }

    close(sockfd);
    return 0;
}
```

AF_INET 原始套接字

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

int main() {
    int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockfd < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    char buffer[2048];
    while (1) {
        int len = recvfrom(sockfd, buffer, sizeof(buffer), 0, NULL, NULL);
        if (len > 0) {
            struct iphdr *ip = (struct iphdr *)buffer;
            struct icmphdr *icmp = (struct icmphdr *)(buffer + (ip->ihl << 2));
            printf("Received ICMP packet of length %d\n", len);
        }
    }

    close(sockfd);
    return 0;
}
```

总之，当数据包到达链路层时，如果存在 AF_PACKET 原始套接字，数据包会被克隆并传递

给匹配的 AF_PACKET 套接字进行处理。同样地，当数据包到达网络层时，如果存在 AF_INET 原始套接字，数据包会被克隆并传递给匹配的 AF_INET 套接字进行处理。这种机制确保了原始套接字能够接收到所有匹配的数据包，同时数据包仍会继续在协议栈中进行后续处理。