《计算机网络实验》实验报告

实验名称：**RAW SOCKET** 编程与以太网帧分析基础

姓名：胡育玮

学号：171860574

邮箱：yeevee@qq.com

班级：17 级计算机科学与技术系 2 班

一、实验目的

1、编写自己的抓包程序。在本机分别运行 ping 程序和用浏览器浏览网页，使用该抓包程序记录相应的数据包，截图并做简要分析。

2、编写自己的 ping 程序。利用 Raw Socket 封装和发送以太网帧的功能，实现 ICMP 包的发送和接收，并输出结果。分别运行你的 ping 和系统的 ping 程序，截图并作简单比较。

3、了解 Ethernet、ICMP、ARP 等格式的数据包的结构

4、了解 Raw Socket 编程的基本知识，掌握编写 ping 程序的基本步骤

二、数据结构说明

（1）抓包程序

该程序利用 socket 套接字实现网络数据包的抓取以及以太网、IP、ARP、ICMP 包的解析（按照 wireshark 的呈现形式进行输出）。使用的数据结构如下：

①**Ethernet**

在此程序中，我使用 Linux 系统自带的 Ethernet 结构。

在 linux 系统中，使用 **struct ethhdr** 结构体来表示以太

网帧的头部。这个 struct ethhdr 结构体需通过#include <linux/if_ether.h>来使用。其结构和字段含义如下：

```
struct ethhdr
{
    unsigned char h_dest[ETH_ALEN]; //目的MAC地址

    unsigned char h_source[ETH_ALEN]; //源MAC地址

    __u16 h_proto ; //网络层所使用的协议类型
}__attribute__((packed))  //用于告诉编译器不要对这个结构体中的缝隙部分进行填充操作;
```

图 1

所使用的宏如下：

```
#define  ETH_P_IP 0x0800 //IP协议
#define  ETH_P_ARP 0x0806  //地址解析协议(Address Resolution Protocol)
#define  ETH_P_RARP 0x8035  //返向地址解析协议(Reverse Address Resolution Protocol)
#define  ETH_P_IPV6 0x86DD  //IPV6协议
```

图 2

②**ARP**

在此程序中，我使用 Linux 系统自带的 ARP 结构。

在 linux 系统中，使用 **struct ethhdr** 结构体来表示 ARP 帧。这个 struct ethhdr 结构体需通过#include <net/if_arp.h>，#include <netinet/if_ether.h>来使用。其结构和使用的宏及其含义如下：

```
struct arphdr
  {
    unsigned short int ar_hrd;      /* Format of hardware address.  */
    unsigned short int ar_pro;      /* Format of protocol address.  */
    unsigned char ar_hln;       /* Length of hardware address.  */
    unsigned char ar_pln;       /* Length of protocol address.  */
    unsigned short int ar_op;       /* ARP opcode (command).  */
#if 0
```

图 3

```
struct  ether_arp {
    struct  arphdr ea_hdr;      /* fixed-size header */
    u_int8_t arp_sha[ETH_ALEN]; /* sender hardware address */
    u_int8_t arp_spa[4];        /* sender protocol address */
    u_int8_t arp_tha[ETH_ALEN]; /* target hardware address */
    u_int8_t arp_tpa[4];        /* target protocol address */
};
#define arp_hrd ea_hdr.ar_hrd
#define arp_pro ea_hdr.ar_pro
#define arp_hln ea_hdr.ar_hln
#define arp_pln ea_hdr.ar_pln
#define arp_op  ea_hdr.ar_op
```

图 4

③IP

在此程序中，我使用 Linux 系统自带的 IP 结构。

在 linux 系统中，使用 **struct iphdr** 结构体来表示 IP 帧的头部。这个 struct iphdr 结构体需通过#include <netinet/ip.h> 来使用。其结构和使用的宏如下：

```
struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;       //header length
    unsigned int version:4;   //version of IP protocol: usually IPv4
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int8_t tos;       //type of service
    u_int16_t tot_len;  //Total Length
    u_int16_t id;       //Identification
    u_int16_t frag_off; //Flags
    u_int8_t ttl;       //Time to live
    u_int8_t protocol;  //lower protocol that the IP frame use: TCP, UDP, ICMP, etc.
    u_int16_t check;    //Header Checksum
    u_int32_t saddr;    //Source
    u_int32_t daddr;    //Destination
    /*The options start here. */
  };
```

图 5

④ **ICMP**

在此程序中，我使用 Linux 系统自带的 ICMP 结构。

在 linux 系统中，使用 **struct icmphdr** 结构体来表示 IP 帧的头部。这个 struct icmphdr 结构体需通过#include <netinet/ip_icmp.h>来使用。其结构如下：

```
struct icmphdr
{
  u_int8_t type;        /* message type */
  u_int8_t code;        /* type sub-code */
  u_int16_t checksum;   //checksum
  union
  {
    struct
    {
      u_int16_t id;
      u_int16_t sequence;
    } echo;          /* echo datagram */
    u_int32_t   gateway;    /* gateway address */
    struct
    {
      u_int16_t __glibc_reserved;
      u_int16_t mtu;
    } frag;          /* path mtu discovery */
  } un;
};
```

图 6

（2）**ping 程序**

该程序通过自己构造一个 icmp 包并将其填充，来实现 ping 操作。使用的数据结构如下：

①**timeval**：使用 Linux 系统自带的 timeval 结构体：

```
struct timeval
{
__time_t tv_sec;          /* Seconds. */
__suseconds_t tv_usec;    /* Microseconds. */
};
```

其中，tv_sec 为 Epoch 到创建 struct timeval 时的秒数，tv_usec 秒后面余出的微秒数。使用该结构体的作用是：**记录发送 ping 包的时间和接受到包的时间，计算这两者的差便得到 RTT 值。**

②**sorkaddr_in**：使用 Linux 系统自带的 sorkaddr_in 结构体：

```
struct sockaddr_in
{
    sa_family_t         sin_family;     //地址族（Address Family）
    uint16_t            sin_port;       //16 位 TCP/UDP 端口号
    struct in_addr      sin_addr;       //32 位 IP 地址
    char                sin_zero[8];    //不使用
};
```

该结构体中提到的另一个结构体in_addr定义如下，它用来存放32位IP地址。

```
struct in_addr
{
    In_addr_t           s_addr;         //32 位 IPv4 地址
};
```

sockaddr_in 在头文件<netinet/in.h>或<arpa/inet.h>中定义。本程序中用作 ping 的目的地址等的数据结构。

③**hosten**：使用 Linux 系统自带的 hosten 结构体:

```
struct hostent
{
    char *h_name;       /* 主机的官方域名 */
    char **h_aliases;   /* 一个以NULL结尾的主机别名数组 */
    int h_addrtype;     /* 返回的地址类型，在Internet环境下为AF-INET */
    int h_length;       /* 地址的字节长度 */
    char **h_addr_list; /* 一个以0结尾的数组，包含该主机的所有地址*/
};
#define h_addr h_addr_list[0]  /*在h-addr-list中的第一个地址*/
```

该函数的作用是用来保存 IP 地址，主要应用于**gethostbyname()**函数。该函数将在后面介绍。

④**IP**：使用 Linux 系统自带的 IP 结构体:

```
struct ip
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ip_hl:4;       /* header length */
    unsigned int ip_v:4;        /* version */
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;        /* version */
    unsigned int ip_hl:4;       /* header length */
#endif
    u_int8_t ip_tos;            /* type of service */
    u_short ip_len;         /* total length */
    u_short ip_id;          /* identification */
    u_short ip_off;         /* fragment offset field */
#define IP_RF 0x8000            /* reserved fragment flag */
#define IP_DF 0x4000            /* dont fragment flag */
#define IP_MF 0x2000            /* more fragments flag */
#define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
    u_int8_t ip_ttl;            /* time to live */
    u_int8_t ip_p;          /* protocol */
    u_short ip_sum;         /* checksum */
    struct in_addr ip_src, ip_dst;  /* source and dest address */
  };
```

本程序中使用该结构的目的是：读取接收到的 IP 数据包的头部的长度，以去掉 IP 头部，让指针直接指向收到的数据包的 ICMP 部分并进行后续的操作。

⑤**ICMP**：使用 Linux 系统自带的 ICMP 结构体:

```
struct icmp
{
  u_int8_t  icmp_type;  /* type of message, see below */
  u_int8_t  icmp_code;  /* type sub code */
  u_int16_t icmp_cksum; /* ones complement checksum of struct */
```

```
#define icmp_pptr    icmp_hun.ih_pptr
#define icmp_gwaddr  icmp_hun.ih_gwaddr
#define icmp_id      icmp_hun.ih_idseq.icd_id
#define icmp_seq     icmp_hun.ih_idseq.icd_seq
#define icmp_void    icmp_hun.ih_void
```

本结构体过长，故不展示全部内容。本程序中仅需手动填充该结构体的 icmp_type, icmp_code, icmp_cksum, icmp_id, icmp_sep，icmp_data 这 6 个字段，其余字段无需理会。

三、程序设计的思路以及运行流程

（1）抓包程序

①设立缓存区 buf，建立套接字

由于在建立套接字后将开始接收数据包，因此在建立套接字前应设立缓存区 buf，以存放收到的数据包：

```
#define BUFSIZE 2048

//static void hex_print(const u_char *buf, int len,
static int echo_eth(const u_char *buf);
static int echo_ip(const u_char *buf);
static int echo_arp(const u_char *buf);
//static int echo_tcp(const u_char *buf);
//static int echo_udp(const u_char *buf);
static int echo_icmp(const u_char *buf);


int main()
{
    int socketret;
    int n;
    char buf[BUFSIZE];
```

图 7：buf 数组及其大小：

②然后建立套接字，若建立失败则输出错误信息，成功则进行下一步处理：

```
//建立PACKET套接字
if ((socketret = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0)
{  //建立成功则开始循环抓包，否则输出错误信息并退出
    perror("socket");
    exit(1);
}
```

图 8：建立成功则开始循环抓包。

建立套接字的语句格式为：

int sockid= socket(family, type, protocol)

本程序中建立套接字使用 **ETH_P_ALL** 作为 Protocol 参数，以抓取所有类型的包。

套接字建立完成后，开始循环抓包：

```
//循环抓包
while (1)
{
    if ((n = recv(socketret, buf, BUFSIZE, 0)) > 0)
    {
        printf("==========================start==========================\n");

        echo_eth(buf);  //resolve headers
        printf("\n==========================end==========================\n\n\n\n");
    }
}

//hex_print(buf, n, 0);  //print hex numbers first
return 0;
```

图 9：循环抓包

不断监测 recv()的返回值，（**该函数的返回值代表成功接收到的字节数目，若为负数代表没有收到字节**），若大于 0 则代表收到了包，便开始下一步的包解析工作：

如图 9，将缓存区 buf 的首地址传给 echo_eth()函数来展开包的解析工作。这个函数的作用是解析收到的包的以太网头部字段并进行上层协议的分析：

```
//以太网抓包：解析以太网头部字段
static int echo_eth(const u_char *buf)
{
    struct ethhdr *eth = (struct ethhdr *) buf;
    printf("\n==========================Ethernet==========================");
    printf("\nDestination: %02X:%02X:%02X:%02X:%02X:%02X",
          HWADDR(eth->h_dest));

    printf("\nSource: %02X:%02X:%02X:%02X:%02X:%02X",
          HWADDR(eth->h_source));
```

图 10：以太网头部字段解析

③如图，该解析函数使用了 ethhdr 结构体来表示 buf 数组，以方便读取 buf 中的内容。**这种技巧在后面的解析中也会不断使用。**

使用了这个结构体后，便可利用之来读取 buf 内的数据，并以我们希望的方式来打印这些数据。图 10 中显示了打印数据包的源 MAC 地址和目的 MAC 地址的代码（两个 printf()函数），这两行代码都使用了宏（图中划红线处）来“展开”printf()的参数，以方便阅读。本程序中会出现的各个这样的宏如下：

```
#define HWADDR(addr) ((unsigned char *)&addr)[0], ((unsigned char *)&addr)[1], ((unsigned char *)&addr)[2], ((unsigned char *)&addr)[3], ((unsigned char *)&addr)[4], ((unsigned char *)&addr)[5]

/*
*     Display an IP address in readable format.
*/
#define NIPQUAD(addr) ((unsigned char *)&addr)[0], ((unsigned char *)&addr)[1], ((unsigned char *)&addr)[2], ((unsigned char *)&addr)[3]


#define HIPQUAD(addr) ((unsigned char *)&addr)[3], ((unsigned char *)&addr)[2], ((unsigned char *)&addr)[1], ((unsigned char *)&addr)[0]


#define PWORD(addr) ((unsigned char *)&addr)[0], ((unsigned char *)&addr)[1]
```

图 10：宏

其他以太网头部字段的解析：

```
printf("\nType: ");

/*
if (((unsigned char *)&(eth->h_proto))[0] ==
    printf("IPv4 (0x%02X%02X)", ((unsigned ch

else if (((unsigned char *)&(eth->h_proto))[0
    printf("ARP (0x%02X%02X)", ((unsigned cha
else if (((unsigned char *)&(eth->h_proto))[0
    printf("RARP (0x%02X%02X)", ((unsigned ch
*/
if (ntohs(eth->h_proto) == ETH_P_IP)
    printf("IPv4 (0x0800)");
else if (ntohs(eth->h_proto) == ETH_P_ARP)
    printf("ARP (0x0806)");
else if (ntohs(eth->h_proto) == ETH_P_RARP)
    printf("RARP (0x8035)");
```

图 10：协议类型字段

如上图，协议类型字段的解析使用了宏 ETH_P_IP 等来判断协议的值。同时也使用了 ntohs()函数。这个函数的含义是“network to host short”，作用是将收到的网络字节转换为接收主机的数据表示方式（大端或小端）。**由于在网络中数据按照大端方式传输而在 x86 体系下数据以小端方式表示，故需要将收到的数据进行转换成小端**。ntohs()函数就是干这种事的，它能将 16 比特（short 类型）的数据转换为小端（机器内使用的端的类型）并返回。类似的函数还有 ntohl(), htons(), htonl(). 其中 n 和 h 代表 network 和 host, l 代表 long 类型（32 位，在 x86 体系下）。对于单个字节的数据无需进行这种转换，故没有 ntohc()（c 代表 char 类型）这种函数。

之后就是以太网的上层协议的识别部分：

```
//根据协议类型：选择IP或ARP协议
switch (ntohs(eth->h_proto))  //ntohs：如果主机与网络使用的字节的大小端方式不同，颠倒过来
{
case ETH_P_IP:
    return echo_ip((u_char *)(eth+1)); //
case ETH_P_ARP:
    return echo_arp((u_char *)(eth+1)); //直接加1：利用指针的加“1”在实际上移动整个结构长度的特性，直接让指针指向数据区，去掉包头
default:
    return -1;
}
return 0;
```

图 11：协议类型字段

根据 eht->h_protocol 的值来决定下一步要解析哪种包。注意划线的部分，这个操作直接去除了以太网包头，将携带的以太网数据区的首地址传给相应的解析函数。

④ARP 包的解析：

解析方式与以太网包头的解析方式如出一辙，都是利用相应的结构体来“解读”一个字节数组，从而直接取出所要的数据：

```
//ARP协议
static int echo_arp(const u_char *buf)
{
    struct ether_arp *arph = (struct ether_arp *) buf;
    printf("\n\n=========================ARP=========================");

    //printf("\nHardware Type: %d", ntohs(arph->arp_hrd));
    printf("\nHardware Type: ");
    if (ntohs(arph->arp_hrd) == 1)
        printf("Ethernet (1)");

    //printf("\nProtocol Type: %02X-%02X", PWORD(arph->arp_pro));
    printf("\nProtocol Type: ");
    /*
```

图 12：ARP 解析（1）。使用了 ether_arp 结构体

```
printf("\nProtocol Type: ");
/*
if (((unsigned char *)&(arph->arp_pro))[0] == 0x
    printf("IPv4 (0x%02X%02X)", ((unsigned char

else if (((unsigned char *)&(arph->arp_pro))[0]
    printf("ARP (0x%02X%02X)", ((unsigned char *
else if (((unsigned char *)&(arph->arp_pro))[0]
    printf("RARP (0x%02X%02X)", ((unsigned char
*/
if (ntohs(arph->arp_pro) == ETH_P_IP)
    printf("IPv4 (0x0800)");
else if (ntohs(arph->arp_pro) == ETH_P_ARP)
    printf("ARP (0x0806)");
else if (ntohs(arph->arp_pro) == ETH_P_RARP)
    printf("RARP (0x8035)");
```

图 13：ARP 解析（2）：协议类型解析

```
printf("\nHardware Size: %d", arph->arp_hln);

printf("\nProtocol Size: %d", arph->arp_pln);

//ARP请求为1，ARP响应为2，RARP请求为3，RARP响应为4
//printf("\nOpcode: %d", ntohs(arph->arp_op));
printf("\nOpcode: ");
if (ntohs(arph->arp_op) == 1)
    printf("ARP request (1)");
else if (ntohs(arph->arp_op) == 2)
    printf("ARP response (2)");
else if (ntohs(arph->arp_op) == 3)
    printf("RARP request (3)");
else if (ntohs(arph->arp_op) == 4)
    printf("RARP response (4)");

printf("\nSender MAC address: %02X:%02X:%02X:%02X:%02X:%02X",
    HWADDR(arph->arp_sha));

printf("\nSender IP address: %d.%d.%d.%d", NIPQUAD(arph->arp_spa));

printf("\nTarget MAC address: %02X:%02X:%02X:%02X:%02X:%02X",
    HWADDR(arph->arp_tha));

printf("\nTarget IP address: %d.%d.%d.%d", NIPQUAD(arph->arp_tpa));
```

图 14：ARP 解析（3）：其他字段的解析。同样使用了各种宏来展开参数，以及 ntohs()

⑤IP 包头各字段的解析：

与 ARP 包的解析相同：使用了 iphdr 结构体类型来解析包头：

```
//IP协议
static int echo_ip(const u_char *buf)
{ //#include <netinet/ip.h>
    struct iphdr *iph = (struct iphdr *) buf;
    printf("\n\n==========================IP==========================");

    printf("\nVersion: %d", iph->version);
    printf("\nHeader Length: %d bytes (%d)", iph->ihl * 4, iph->ihl);
    printf("\nTOS: 0x%02X", iph->tos);
    printf("\nTotal Length: %d", ntohs(iph->tot_len));
    printf("\nIdentification: 0x%04x (%d)", ntohs(iph->id), ntohs(iph->id));


    //((unsigned char *)&addr)[0], ((unsigned char *)&addr)[1]
    //printf("\nFlags: 0x%02X%02X", PWORD(iph->frag_off));
    printf("\nFlags: 0x%02X%02X", ((unsigned char *)&(iph->frag_off))[0], ((unsigned char *)&(iph->frag_off))[1]);
    printf("\nTime to live: %d", iph->ttl);
```

```
//TCP 协议的协议号为6，UDP 协议的协议号为17。
//printf("\nProtocol: %d", iph->protocol);
printf("\nProtocol: ");
switch (iph->protocol)
{
    case IPPROTO_TCP:
        printf("TCP (%d)", IPPROTO_TCP);
        break;

    case IPPROTO_UDP:
        printf("UDP (%d)", IPPROTO_UDP);
        break;

    case IPPROTO_ICMP:
        printf("ICMP (%d)", IPPROTO_ICMP);
        break;

    default: ;
}
```

图 15：IP 解析。第二张图解析了 IP 包中的数据段中的上层协议的类型并打印

```
printf("\nHeader Checksum: 0x%02X%02X", PWORD(iph->check));

printf("\nSource: %d.%d.%d.%d", NIPQUAD(iph->saddr));
printf("\nDestination: %d.%d.%d.%d", NIPQUAD(iph->daddr));

//printf("\nIPPROTO_TCP: %d", IPPROTO_TCP);
//printf("\nIPPROTO_UDP: %d", IPPROTO_UDP);
//printf("\nIPPROTO_ICMP: %d", IPPROTO_ICMP);

//选择TCP或UDP
switch (iph->protocol)
{
case IPPROTO_ICMP:
      return echo_icmp((u_char *)&iph[1]);


default:
      return -1;
}
```

图 16：IP 解析。因为本程序只解析 ICMP 报文，故监测到 TCP 或 UDP 包时不会进一步解析，只会进一步解析 ICMP 报文

⑥ICMP 报文解析：

```
static int echo_icmp(const u_char *buf)
{
    struct icmphdr *icmpph = (struct icmphdr*) buf;
    printf("\n\n=========================ICMP=========================");
    printf("\nType: %d", ntohs(icmpph->type));
    printf("\nCode: %d", ntohs(icmpph->code));
    printf("\nChecksum: 0x%04x", ntohs(icmpph->checksum));
    printf("\nIdentifier: %d", ntohs(icmpph->un.echo.id));
    printf("\nSequence Number: %d", ntohs(icmpph->un.echo.sequence));
    //printf("\n时间戳: %d", ntohl(icmpph->timestamp));
```

图 17：ICMP 报文解析。使用了 icmphdr 结构体类型

（2）ping 程序

该程序的总逻辑如下：**填充 ICMP 包；发送 ICMP 包；接收包；解析包并打印相关信息**。相关的全局变量和宏的定义如下：

```
#define PACKET_SIZE          4096  //数据包的长度
//#define MAX_WAIT_TIME        5
#define MAX_NO_PACKETS       4    //发送数据的次数:可以自己设置，也可以像真实的p
#define DATA_LEN             56  //数据长度


char sendpacket[PACKET_SIZE];    //发送ICMP数据报缓存
char recvpacket[PACKET_SIZE];    //接收ICMP数据报缓存

int nsend = 0;//发送数据报时的序号，即pack（）中的pack_no参数
//int nreceived = 0;


struct timeval send_time;  //发送数据包的时间。
struct timeval recv_time;  //接收数据包时的时间。减去send_time，则可以得到RTT
struct sockaddr_in from;   //存储recvfrom时对方的socket数据结构
```

①填充 ICMP 包

使用我们自己写的 pack()函数来填充 ICMP 包：

```
int pack(int pack_no)
{
    //int i, packsize;
    int packsize;
    struct icmp *icmp;     //使用了系统自带的ICMP结构体
    struct timeval *tval;  //使用了系统自带的timeval结构

    icmp = (struct icmp*)sendpacket; //封装ICMP包头
    icmp->icmp_type = ICMP_ECHO;   //设置type为ICMP_ECHO，发送数据包
    icmp->icmp_code = 0;  //ECHO
    icmp->icmp_cksum = 0;    //校验值 :后面再重新计算，因为要计算整个包的检验和
    icmp->icmp_seq = pack_no;  //包序号
    icmp->icmp_id = 2;     //ID，使用PID唯一标识来标识？？？并没有必要,自己随便设置一个就行了
    packsize = 8 + DATA_LEN; //包头长度(8)+数据长度(DATA_LEN)

    gettimeofday(&send_time, NULL);
    strcpy((char*)&icmp->icmp_data, "random data 233");

    icmp->icmp_cksum = cal_chksum( (unsigned short *)icmp, packsize); //计算检验和:整个包的检验和!
    return packsize;
```

如上图，填充 ICMP 包的 **type, code, cksum, seq, id, data** 字段。其中 type 字段设为 ICMP_ECHO，用来发送 ping 数据包；检验和字段应先设为 0，后续计算检验和时要计算整个包（包括数据字段）的检验和。包序号字段填上 pack_no 参数，每发送完一个数据包这个值（nsend）会自增 1，这样每个发送的包的序号是递增的。**ID 字段可以自己填写一个固定值。然后调用 gettimeofday() 来获取当前发送时间到 send_time 中。**

数据字段我填上了“random data 233”，这个数据字段可以随便填写。最后计算整个包的检验和并填充。检验和计算函数如下：

```
unsigned short cal_chksum(unsigned short *addr, int len)  //计算校验和
{
    int nleft = len;
    int sum = 0;
    unsigned short *w = addr;
    unsigned short answer = 0;

    //加法运算
    while (nleft>1)
    {
        sum += *w++; //sum每次加w指向的地址的值，然后w指向下一个short数
        nleft-=2;    //每次处理2个字节，故减2
    }

    if (nleft == 1)
    {   //如果剩下1个字节未处理：把它也加起来
        *(unsigned char *)(&answer)=*(unsigned char *)w;
        sum+=answer;
    }
    sum = (sum >> 16) + (sum & 0xffff);  //sum本为32位数，这里将其高16位与低16位相加：就是加上所有的溢出值!
    sum += (sum >> 16);  //然后再加上其高16位：把溢出的值也加起来

    //取反后输出
    answer = ~sum;

    return answer;  //最后取反
}
```

②发送包：使用自己编写的 send_packet()函数来发送：

```
/*发送数据报*/
void send_packet(int sockfd, struct sockaddr_in dest_addr)
{
    int packetsize;
    packetsize = pack(nsend); //打包
    /*这里我们只需要发送ICMP数据报就行，ip相关的头部由内核自己填充*/
    if (sendto(sockfd, sendpacket, packetsize, 0, (struct sockaddr *)&dest_addr, sizeof(dest_addr)) < 0)
    {
        perror("sendto error");
        return;
    }
}
```

如图，调用 pack()来填充要发送的包，然后调用 sendto()来发送。**只需发送 ICMP 数据报即可，IP 相关的头部由内核自己填充。**

③接收包：

```
while (1)
{

cvfrom()用来接收远程主机经指定的socket传来的数据,并把数据传到由参数buf指向的内存空间,参数len为可接收数据的最大长度.参数flag
recv().参数from用来指定欲传送的网络地址,结构sockaddr请参考bind()函数.参数fromlen为sockaddr的结构长度. 返回值:成功则返回接
 */

    //不断等待，直到收到包
    if ((n = recvfrom(sockfd, recvpacket, sizeof(recvpacket), 0, (struct sockaddr *)&from, &fromlen)) < 0)
    {
        if (errno != EINTR)
        {
            perror("recvfrom error");
            return;
        }
    }

    else
        break;
}
```

不断等待，直到 recvfrom()接收到数据包。收到的数据包存在 recvpacket 数组中。

```
    gettimeofday(&recv_time, NULL);
    if (unpack(recvpacket, n) == -1)
    {//进行解包
        fprintf(stderr, "%s\n", "Unreachable");
    }
}
```

然后获取接收到包的时间，存到 recv_time 结构体中，然后调用 unpack()进行包的解析以及相关信息的打印。

④解析包

```
int unpack(char *buf, int len)
{

    /*
    struct timeval {
          time_t       tv_sec;     //seconds
          suseconds_t   tv_usec;   //microseconds
    };

    */


    int i, iphdrlen;
    struct ip *ip;       //此数据结构可以在netinet/ip.h中查看
    struct icmp *icmp;  //此数据结构在netinet/ip_icmp.h中查看
    //struct timeval *tvsend;
    double rtt;          //RTT(Round Trip Time): 一个连接的往返时间，即数据发送时刻到接收到确认的时刻的差值;

    ip = (struct ip *)buf;      //ip头部
    iphdrlen = ip->ip_hl * 4;  // ip_hl占4位，各位占4个单位长度，因此乘以4
    icmp = (struct icmp *)(buf + iphdrlen);  //icmp段的地址 :IP指针移动头部的长度后达到
    len -= iphdrlen;                         //ICMP包长度: IP包总长度减IP包头部长度
```

如图，首先利用 IP 结构来去掉收到的 IP 包的包头，获取 ICMP 部分。

```
if (len < 8)                //ICMP固定长度为8，小于即错误
{
    printf("ICMP packets\'s length is less than 8!\n");
    return -1;
}


if ( icmp->icmp_type == ICMP_ECHOREPLY && icmp->icmp_id == 2 )  //判断是不是想要的包
{
    time differ(&recv time, &send time);
    rtt = ((double)recv time.tv sec) * 1000.0 + ((double)recv time.tv usec) / 1000.0;
    //ttl 生存时间和往返时间 rtt（单位是毫秒）

    //打印提示信息
    printf("%d byte from %s: icmp_seq = %u ttl = %d rtt = %.3f ms\n",
    len, inet_ntoa(from.sin_addr), icmp->icmp_seq, ip->ip_ttl, rtt);
}


else
{
    printf("icmp->icmp_id: %d\n", icmp->icmp_id);
    return -1;
}
```

如图，若 ICMP 包长度小于 8 则发生错误。然后根据 icmp_type 和 icmp_id 来判断收到的包是不是想要的 ping 后的回应。**接着利用前面已经设置好的 recv_time 和 send_time 的值，调用 time_differ()来计算时间差，把时间差存到 recv_time 中**。由于 timeval 结构体中的两个量一个是秒另一个是微秒，而 ping 程序显示的 RTT 应为毫秒形式，故上图计算 RTT 时进行了转换。

最后打印出相关信息：接收的 ICMP 包长度；源地址；包序号；TTL（调用 IP 结构来获取 TTL）；RTT。

计算时间差的函数 time_differ()：

```
void time_differ(struct timeval *out, struct timeval *in)   //计算接收时间与发送时间的差值
{   //如果接收时间小于发送时间，进行借位。
    out->tv_usec -= in->tv_usec; //微秒
    out->tv_sec -= in->tv_sec; //秒

    if (out->tv_usec < 0)
    {
        out->tv_sec--;
        out->tv_usec += 1000000;
    }
```

如果相减后微秒值为负，则借用 1 秒来补回微秒值。

至此，完整的发送-解析流程已结束。下面简要介绍主函数：

```
int sockfd;
struct sockaddr_in dest_addr;    //ping的目的地址数据结构
in_addr_t inaddr = 0;            //存储ip地址
```

```
if ( (sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) ) < 0) //创建套接字:接收ICMP数据包
{
    perror("socket error");
    exit(1);
}
```

主函数的主要作用之一就是创建套接字，用来发送数据。

```
struct hostent *host;
if ((inaddr = inet addr(argv[1])) == INADDR NONE)
{   //判断是域名还是IP地址:若是域名，则用gethostbyname()获取IP存到host里
    if ((host = gethostbyname(argv[1])) == NULL)
    {
        perror("gethostbyname error");
        exit(1);
    }


    memcpy( (char *)&(dest_addr.sin_addr), host->h_addr, host->h_length);  //将得到的IP地址拷贝
}


else
    memcpy( (char *)&(dest_addr.sin_addr), (char *)&inaddr, sizeof(inaddr));
```

如上图，另一作用是判断用户输入的要 ping 的地址是一个域名还是 IP 地址。将用户在命令行键入的地址（argv[1]）通过 inet_addr()转换网络字节顺序的整型值，存到变量 inaddr 中；若转换失败则说明用户键入的是域名而非 IP 地址，此时使用 gethostbyname()来把域名转换为 IP 地址，并存放到 host 结构体中，然后把 host 结构体中的 IP 地址赋值给 dest_addr.sin_addr。如果用户键入的就是 IP 地址，则可直接把 inaddr 结构中保存的赋给 dest_addr.sin_addr。

```
while (1)
{
    nsend++;
    send_packet(sockfd, dest_addr);     //发送数据报
    recv_packet(sockfd);                //接收数据报

    sleep(1);
}

//statistics(0);//输出最后的状态汇总信息

close(sockfd);
return 0;
```

然后进入循环发送和接收模式。将 socket 号和 dest_addr 传给发送函数，然后使用接收函数接收并解析数据包，再 sleep 1 秒种，以控制发送和接收的节奏。不断循环，直至用户按下 ctrl + z 停止为止。

四、运行结果截图

ping：外网和内网：

```
fusion@fusion-virtual-machine:~/桌面/netlab2$ sudo ./ping www.baidu.com
[sudo] fusion 的密码:
PING www.baidu.com(119.75.217.109): 56 bytes data in ICMP packets.
64 byte from 119.75.217.109: icmp_seq = 1 ttl = 128 rtt = 39.329 ms
64 byte from 119.75.217.109: icmp_seq = 2 ttl = 128 rtt = 34.031 ms
64 byte from 119.75.217.109: icmp_seq = 3 ttl = 128 rtt = 34.954 ms
64 byte from 119.75.217.109: icmp_seq = 4 ttl = 128 rtt = 34.074 ms
64 byte from 119.75.217.109: icmp_seq = 5 ttl = 128 rtt = 35.060 ms
64 byte from 119.75.217.109: icmp_seq = 6 ttl = 128 rtt = 37.168 ms
64 byte from 119.75.217.109: icmp_seq = 7 ttl = 128 rtt = 34.733 ms
64 byte from 119.75.217.109: icmp_seq = 8 ttl = 128 rtt = 33.916 ms
64 byte from 119.75.217.109: icmp_seq = 9 ttl = 128 rtt = 34.750 ms
64 byte from 119.75.217.109: icmp_seq = 10 ttl = 128 rtt = 38.866 ms
```

```
fusion@fusion-virtual-machine:~/桌面/netlab2$ sudo ./ping 192.168.2.2
PING 192.168.2.2(192.168.2.2): 56 bytes data in ICMP packets.
64 byte from 192.168.2.2: icmp_seq = 1 ttl = 64 rtt = 0.901 ms
64 byte from 192.168.2.2: icmp_seq = 2 ttl = 64 rtt = 0.569 ms
64 byte from 192.168.2.2: icmp_seq = 3 ttl = 64 rtt = 0.372 ms
64 byte from 192.168.2.2: icmp_seq = 4 ttl = 64 rtt = 0.479 ms
64 byte from 192.168.2.2: icmp_seq = 5 ttl = 64 rtt = 1.239 ms
64 byte from 192.168.2.2: icmp_seq = 6 ttl = 64 rtt = 0.309 ms
64 byte from 192.168.2.2: icmp_seq = 7 ttl = 64 rtt = 0.473 ms
64 byte from 192.168.2.2: icmp_seq = 8 ttl = 64 rtt = 0.507 ms
```

抓包：

```
==========start==========

==========Ethernet==========
Destination: 00:0C:29:48:69:BB
Source: 00:0C:29:B2:64:1F
Type: ARP (0x0806)

==========ARP==========
Hardware Type: Ethernet (1)
Protocol Type: IPv4 (0x0800)
Hardware Size: 6
Protocol Size: 4
Opcode: ARP request (1)
Sender MAC address: 00:0C:29:B2:64:1F
Sender IP address: 192.168.2.2
Target MAC address: 00:00:00:00:00:00
Target IP address: 192.168.2.1
==========end==========
```

```
==========start==========

==========Ethernet==========
Destination: 00:0C:29:B2:64:1F
Source: 00:0C:29:48:69:BB
Type: ARP (0x0806)

==========ARP==========
Hardware Type: Ethernet (1)
Protocol Type: IPv4 (0x0800)
Hardware Size: 6
Protocol Size: 4
Opcode: ARP response (2)
Sender MAC address: 00:0C:29:48:69:BB
Sender IP address: 192.168.2.1
Target MAC address: 00:0C:29:B2:64:1F
Target IP address: 192.168.2.2
==========end==========
```

上面 2 个图为 ARP 的 request 和 response

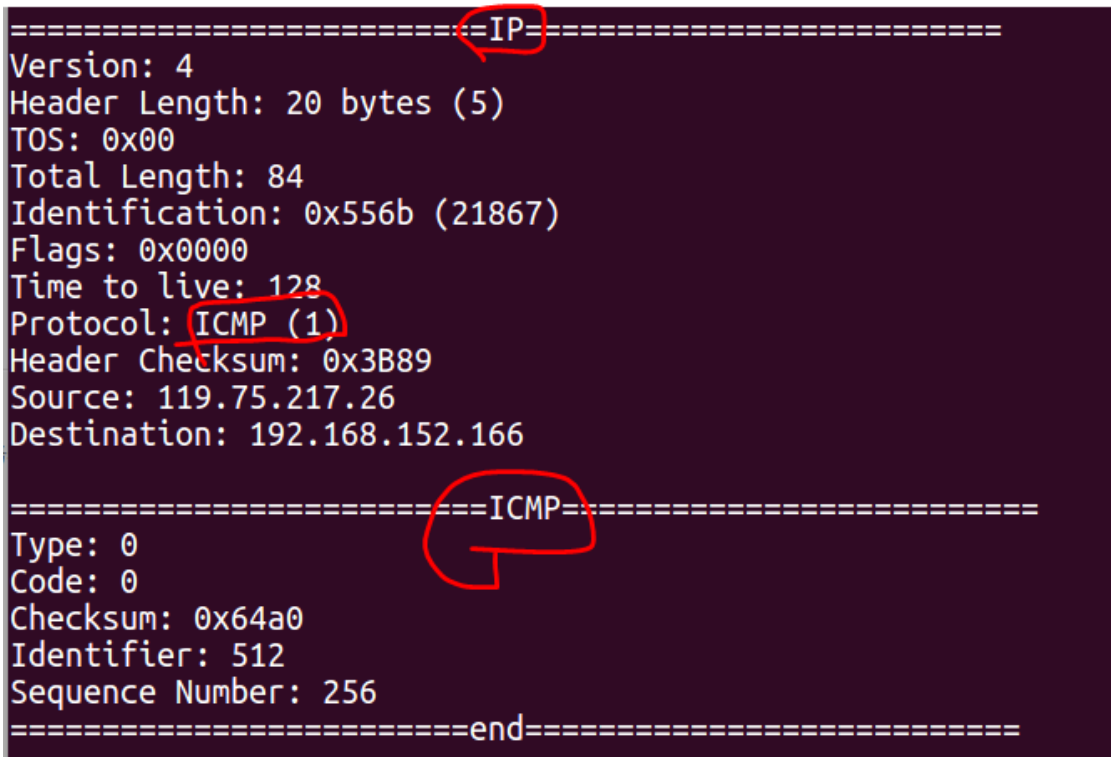
```
==========start==========

==========Ethernet==========
Destination: 00:0C:29:48:69:CF
Source: 00:50:56:E3:67:77
Type: IPv4 (0x0800)

==========IP==========
Version: 4
Header Length: 20 bytes (5)
TOS: 0x00
Total Length: 71
Identification: 0x555c (21852)
Flags: 0x0000
Time to live: 128
Protocol: TCP (6)
Header Checksum: 0xFEE7
Source: 52.34.88.252
Destination: 192.168.152.166
==========end==========
```

```
==========Ethernet==========
Destination: 00:00:00:00:00:00
Source: 00:00:00:00:00:00
Type: IPv4 (0x0800)

==========IP==========
Version: 4
Header Length: 20 bytes (5)
TOS: 0x00
Total Length: 69
Identification: 0x418a (16778)
Flags: 0x4000
Time to live: 64
Protocol: UDP (17)
Header Checksum: 0xFA1B
Source: 127.0.0.1
Destination: 127.0.1.1
==========end==========
```



上面 3 图为 IP 和 ICMP

五、相关参考资料

1. http://doc.okbase.net/javawebsoa/archive/117175.html

struct ethhdr 结构体的内容

2. https://www.cnblogs.com/orlion/p/6104204.html

ntohs()等函数的作用

3. https://blog.csdn.net/will130/article/details/53326740/

sockaddr_in 的内容

4. https://blog.csdn.net/wtk870424/article/details/5129211

Hosten 的内容

5.

https://blog.csdn.net/z_ml118/article/details/78285997?tdsourcetag=s_pctim_aiomsg#t13

https://blog.csdn.net/qq_38353700/article/details/78606632

Ping 的实现

6. https://blog.csdn.net/xiaopangzi313/article/details/9699657

抓包程序参考

六、代码个人创新以及思考

①我参考的 ping 样例程序中使用了进程这一概念，但我检查代码后发现，参考程序使用进程的唯一作用就是用进程号来填充 ICMP 报文的 icmp_id 字段。这其实在我们这个简单的 Ping 程序中是没有必要的，这个 icmp_id 字段只需要填入一个固定值即可。在收到 ICMP 包后，解析函数会判断收到的包是否是想要的对 Ping 的回应，这时需要利用 icmp_id 字段。因此只需要填入一个固定的值即可，没必要使用进程。

所以在我的程序里我去掉了进程相关的内容。

②我参考的 ping 样例程序中，在填写 ICMP 包时，将发送时间信息（send_time 结构体内容）填入 icmp->icmp_data 字段作为 ICMP 包携带的信息。然后在收到回应时，记录接收的时间，然后把收到的 ICMP 包的数据区内记录的发送时间提取出来，用接收时间减去发送时间得到 RTT 值。这也是没有必要的。数据区可以填写的内容是随意的；没必要把发送时间记录进数据区而又在接收时提取出这个时间。只需要把 send_time 作为全局变量，在发送时记录时间到 send_time 中，就可以实现完全一样的功能。

③我参考的 ping 样例程序中，运行起来后，接收的所有包的 RTT 的值都在 1000ms 以上，略微超过 1000ms. 这与系统的 ping 程序的 rtt 值大相径庭。经仔细检查，发现原因在于如下的循环中的代码：

```
while (1)
{
    nsend++;
    send_packet(sockfd, dest_addr);     //发送数据报
    sleep(1);
    recv_packet(sockfd);                //接收数据报


}
```

如上图，原代码在发包-收包循环中，每发送一个包，为

了控制节奏而不至于程序运行过快，会立刻 sleep 1 秒。而由上面的分析可知，在发送时记录发送时间，在接收时记录接收时间，这中间夹杂了一个 1 秒的 sleep, 因此 rtt 肯定超过一秒。解决方法就是，把 sleep 挪到 recv_packet()的下面：

```
while (1)
{
    nsend++;
    send_packet(sockfd, dest_addr);     //发送数据报

    recv_packet(sockfd);                //接收数据报
    sleep(1);

}
```

这样既控制了程序运行节奏，又能够解决上述问题。

七、应用场景创新

本次实验实现了抓包和 ping 两个程序，这两个程序一个只接收包，另一个可以更改和发送包。如果把这两个结合，便可以设计出一个更改正常的网络数据包的程序：若我的电脑作为无线热点，设备 A 经由我的电脑发送数据包给设备 B，那么我的电脑上可以运行这个程序来截获 A 发送的包，修改其中的部分内容再发送给 B，造成破坏性的后果。