《计算机网络协议开发》实验报告

第五/六次实验
IPv4 收发和转发实验

姓名：元玉慧
学号：101220151
10 级 计算机 系 4 班
邮箱：njucsyyh@gmail.com
时间：2013/04/12

一、 实验目的

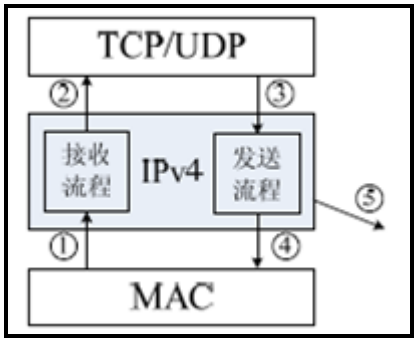
实验 5 通过编程，了解 IPv4 协议的分组接收和发送流程，进一步理解网络成协议的工作原理，并初步掌握网络成协议开发方法。
实验 6 通过对网络的观察视角转移到路由器中，将了解路由器是如何未分组选择路由，并逐跳地将分组发送到目的主机上，具体来说，本实验在前一个 IPv4 分组收发实验的基础上，增加了 IPv4 分组的转发功能，从而实现了路由器中的 IPv4 协议，本实验也会初步接触路由表这一重要的数据结构，认识路由器是如何根据路由表来确定分组转发的下一跳的节点。

二、 实验背景

（lab05 部分）
IPv4 协议是当前互联网的核心协议。它保证了网络节点（包括网络设备和主机）在网络层能够按照标准协议互相通信。在日常使用的计算机的主机协议栈中，IPv4 协议必不可少，它能够接收网络中传送给本机的分组，同时也能根据上层协议的要求将数据封装为 IPv4 分组发送出去。

主机协议栈主要完成数据的发送和接收两项功能，IPv4 协议需要辨别和标识源 IPv4 地址和目的 IPv4 地址。该协议一方面接收并处理发送给自己的分组，另一方面根据上层协议需求将上层数据封装成分组发送出去。通常，IPv4 地址可以在网络中唯一标识一台主机。因为，在相互通信时 ,填写在IPv4分组头部中的IPv4地址就起到了标识源主机和目的主机的作用。

处理流程设计:



接收操作流程：

接收到测试服务器发送来的 IPv4 分组，调用接收接口函数 stud_ip_recv( )

接收处理完成后，调用接口函数 ip_SendtoUp( ) 将需要上层协议进一步处理的信息提交给上层协议；

或者调用函数 ip_DiscardPkt( )丢弃有错误的分组并报告错误类型

发送操作流程：

调用发送接口函数 stud_ip_Upsend( ),此函数实现 IPv4 分组封装发送的功能. 根据所传参数完成 IPv4 分组的封装，之后调用接口函数 ip_SendtoLower( )把分组交给下层完成发送校验和计算

接收和发送 IPv4 分组时，需要计算 IPv4 分组头部校验和（Header checksum）。在发送报文时，为了计算校验和，首先把校验和字段置为 0。然后，对首部中每个 16bit 进行二进制求和（整个首部看成是一串 16bit 的字组成），如果和的高 16bit 不为 0，则将和的高 16bit 和低 16bit 反复相加，直到和的高 16bit 为 0，从而获得一个 16bit 的值，然后将该 16bit 的值取反，结果存在检验和字段中。

当接收 IP 数据报时，同样对首部中每个 16bit 进行二进制求和。由于接收方在计算过程中包含了发送方存在首部中的校验和

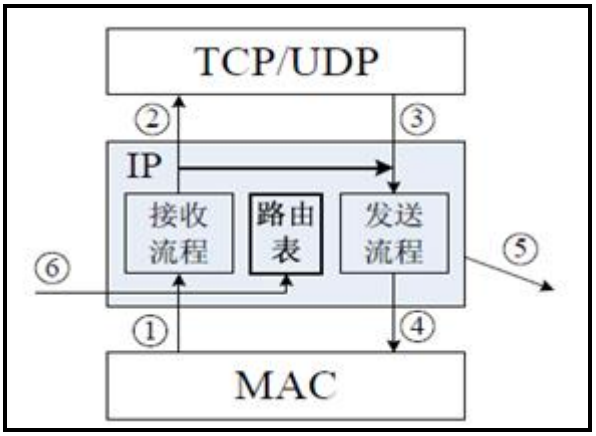
（lab06 部分）

分组转发是路由器的一个基本功能。分组转发的依据是路由表，以此将目的地址不同的分组发送到相应的接口上，逐跳转发并最终到达目的主机。本实验要求按照路由器协议栈的 IPv4 协议功能进行设计实现，接收处理所有收到的分组（而不只是目的地址为本机地址的分组），并根据分组的 IPV4 目的地址结合相关的路由信息，对分组进行转发、接收或丢弃操作。

转发流程设计：

在下层接收接口函数 Stud_fwd_deal( )中（下图中接口函数 1），实现分组接收处理。主要功能是根据分组中目的 IPv4 地址结合对应的路由信息对分组进行处理。如果分组需要上交，则调用接口函数 Fwd_LocalRcv( )（下图中接口函数 2）；需要丢弃，则调用函数 Fwd_DiscardPkt( )（下图中函数 5）；需要转发，则进行转发操作。转发操作的实现要点包括，TTL 值减 1，然后重新计算头校验和，最后调用发送接口函数 Fwd_SendtoLower( )（下图中接口函数 4）将分组发送出去。注意，接口函数 Fwd_SendtoLower( )比前面实验增加了一个参数 pNxtHopAddr，要求在调用时传入下一跳的 IPv4 地址，此地址是通过查找路由表得到的。

本实验增加了一个路由表配置的接口（下图中函数 6），要求能够根据系统所给信息来设定本机路由表。实验中只需要简单地设置静态路由信息，以作为分组接收和发送处理的判断依据，而路由信息的动态获取和交互，在有关路由协议的实验（RIP 协议）中会重点涉及。



路由表的设计：

路由信息包括地址段、距离、下一跳地址、操作类型等。在接收到 IPv4 分组后，要通过其目的地址匹配地址段来判断是否为本机地址，如果是则本机接收；如果不是，则通过其目的地址段查找路由表信息，从而得到进一步的操作类型，转发情况下还要获得下一跳的 IPv4 地址。发送 IPv4 分组时，也要拿目的地址来查找路由表，得到下一跳的 IPv4 地址，然后调用发送接口函数做进一步处理。在 IPv4 收发实验中，发送流程中没有查找路由表来确定下一跳地址的步骤，这项工作由系统来完成了，在本实验中则作为实验内容要求学生实现。需要进一步说明的是，在转发路径中，本路由器可能是路径上的最后一跳，可以直接转发给目的主机，此时下一跳的地址就是 IPv4 分组的目的地址；而非最后一跳的情况下，下一跳的地址是从对应的路由信息中获取的。因此，在路由表中转发类型要区分最后一跳和非最后一跳两种情况。

路由表数据结构的设计是非常重要的，会极大地影响路由表的查找速度，进而影响路由器的分组转发性能。链表结构是最简单的，但效率比较低；树型结构的查找效率会提高很多，但组织和维护有些复杂

二、实验设计要点

收发实验设计要点：

在分组转发的基础上，增加分组转发功能，对于每一个到达本机的分组，根据母的 IPv4 地址决定，对该分组执行的操作。

功能设计

-1-向上层协议上交目的地址为本机地址的分组

-2-根据路由表查找结果，丢弃查不到路由的分组

-3-根据路由查找结果，向相应接口转发不是本机接收的分组

实现设计：

-1-设计路由表的数据结构

-2-IPv4 分组的接收和发送

-3-IPv4 分组的转发

设计路由表的数据结构：
能够根据母的 IPv4 地址来确定分组处理行为，路由表的数据结构和查找算法是比较关键的部分
IPv4 分组的接收和发送
对前面的实验中的代码进行修改，在路由器协议栈的 IPv4 模块中能够正确完成分组的接收和发送处理
IPv4 分组的转发
对于需要转发的分组进行处理，获得下一跳的 IPv4 地址，然后调用发送接口函数做进一步处理

所有的接口函数说明：
（收发实验部分）
-1-丢弃分组函数
extern void ip_DiscardPkt(char* pBuffer,int type);
-2-发送分组函数
extern void ip_SendtoLower(char*pBuffer,int length);
-3-上层接收函数
extern void ip_SendtoUp(char *pBuffer,int length);
-4-获取本机地址函数
extern unsigned int getIpv4Address();
返回 IPv4 地址的主机字节序
（转发实验部分）
-5-本地处理函数
extern void fwd_LocalRcv(char *pBuffer, int length);
本函数是 IPv4 协议接收处理流程的上层接口函数，在对 IPv4 分组完成解析处理之后，如果分组的目的地址是本机的地址 ,则调用本函数将正确分组提交上层相应协议模块进一步处理。
-6-下层发送函数
extern void fwd_SendtoLower(char *pBuffer, int length, unsigned int nexthop);
本函数是发送流程的下层接口部分，在 IPv4 协议模块完成发送封装工作后，调用该接口函数进行后续发送处理，后续的发送处理过程包括分片处理，IPv4 地址到 MAC 地址的映射、封装成帧等工作，这部分内容不需要学生完成，有实验系统来提供。
-7-丢弃分组函数
extern void fwd_DiscardPkt(char *pBuffer, int type);

-8-获取本机地址函数

```
extern unsigned int getIpv4Address();
```

三、实验内容

IPV4 收发实验

Ipv4 头部结构体

```
typedef struct IPv4{
    byte Version;
    byte IHL;
    unsigned short len;
    byte ttl;
    byte protocol;
    unsigned short checksum;
    unsigned int Saddr;
    unsigned int Daddr;
};
```

校验码计算函数

```
unsigned short checksum(unsigned short * buf, unsigned short count){
    long sum=0;
    int size_s = sizeof(unsigned short int);
    while(count > 1){
        sum += *buf++;
        count -= size_s;
    }

    if(count > 0){
    sum += *(unsigned char*)buf;
    }

    while(sum>>16)
    sum = (sum & 0xFFFF) + (sum >> 16);

    return (unsigned short) (~sum);
}
```

IPv4 接收函数关键部分实现;

解析数据包的 IPv4 头部的内容：

```
struct IPv4 header;
//C语言的移位运算设计 char*可以当做char数组进行处理
header.Version = (pBuffer[0]>>4) & 0x0f;
header.IHL = pBuffer[0] & 0x0f;
header.len = ntohs(*((unsigned short *)(pBuffer+2)));
header.ttl = pBuffer[8];
```

```
header.protocol = pBuffer[9];
header.checksum = ntohs(*((unsigned short *)(pBuffer+10)));
header.Saddr = ntohl(*((unsigned int *)(pBuffer+12)));
header.Daddr = ntohl(*((unsigned int *)(pBuffer+16)));
```

判断头部信息是否有出错情况

```
if(header.Version != 4){
ip_DiscardPkt(pBuffer, STUD_IP_TEST_VERSION_ERROR);
  printf("version error\n\n");
return 1;
}

if (header.IHL < 5 || header.IHL > 15){
ip_DiscardPkt(pBuffer, STUD_IP_TEST_HEADLEN_ERROR);
  printf("IHL error\n\n");
return 1;
}

if(header.ttl <= 0){
    ip_DiscardPkt(pBuffer, STUD_IP_TEST_TTL_ERROR);
       printf("ttl error\n\n");
    return 1;
}

// if (header.len < 20 || header.len > 65535){
//  ip_DiscardPkt(pBuffer, STUD_IP_TEST_HEADLEN_ERROR);
//  return 1;
//}
if(checksum((unsigned short *)pBuffer, 4*header.IHL) != 0){
ip_DiscardPkt(pBuffer, STUD_IP_TEST_CHECKSUM_ERROR);
  printf("check error\n\n");
    return 1;
}
//目标地址正确分为两种情况，一是是本机ip地址，二是是广播地址oxFFFF
if (header.Daddr != getIpv4Address() && header.Daddr != 0xffffffff)
{
    ip_DiscardPkt(pBuffer, STUD_IP_TEST_DESTINATION_ERROR);
       printf("Daddr error\n\n");
    return 1;
 }
```

IPv4 发送函数关键部分实现;

如果没有差错，调用发送分组函数将去除 IPv4 头部的数据部分交付给 MAC 层。

```
//发送函数
int stud_ip_Upsend(char *pBuffer,unsigned short len,unsigned int srcAddr,
                   unsigned int dstAddr,byte protocol,byte ttl)
{
    char *send = new char[20+len];
    memset(send, 0, len+20);
    send[0] = 0x45;
    *((unsigned short*)(send+2)) = htons(len+20);
    *((unsigned int*)(send+12)) = htonl(srcAddr);
    *((unsigned int*)(send+16)) = htonl(dstAddr);
    send[8] = ttl;
    send[9] = protocol;
    *((unsigned short*)(send+10)) = checksum((unsigned short *)send, 20);
    memcpy(send+20, pBuffer, len);
    ip_SendtoLower(send, len+20);
    return 0;
}
```

IPv4 接收转发实验函数关键部分实现:

定义路由表表项的内容以及重载比较函数，路由表是用 set 集合来维护组织的

```
typedef struct Route_node{
    //重载比较函数
    unsigned int dest;
    unsigned int masklen;
    unsigned int nexthop;
}Route;

//指定排序的标准
//根据子网编号来建立set集合
class RouteSortCriterion {
public:
    bool operator() (const Route_node &a, const Route_node &b) const {
        unsigned int sub_a = a.dest &  (0xffffffff << (a.masklen));
        unsigned int sub_b = b.dest &  (0xffffffff << (b.masklen));
        if(sub_a < sub_b) return true;
        else return false;
    }
};
//定义一个set集合
set<Route, RouteSortCriterion> s;
```

路由表的维护操作函数（初始化路由表函数和添加路由信息函数）

```
void stud_Route_Init()
{
    s.clear();
    return;
}

void stud_route_add(stud_route_msg *proute)
{
    cout<<"******************************"<<endl;
    Route* t = new Route;
    t->dest = ntohl(proute->dest);
    t->masklen = ntohl(proute->masklen);
    t->nexthop = ntohl(proute->nexthop);
    unsigned char *p1 = (unsigned char *) &(t->dest);
    unsigned char *p2 = (unsigned char *) &(t->nexthop);
    cout<<"Dest: "<<(int)p1[3]<<":"<<(int)p1[2]<<":"<<(int)p1[1]<<":"<<(int)p1[0]<<endl;
    cout<<"Masklen: "<<t->masklen<<endl;
    cout<<"Nexthop: "<<(int)p2[3]<<":"<<(int)p2[2]<<":"<<(int)p2[1]<<":"<<(int)p2[0]<<endl;
    cout<<"************ ADD INFO ************"<<endl;

    s.insert(*t);
    return;
}
```

转发函数关键设计实现：
建立 32 个路由表项，分别存储处理掩码长度为 0~32 之间所有的情况，然后根据路由表项匹配时优先匹配子网号长度较大的表项，因此从掩码为 32 开始递减检测，一旦找到匹配的子网号，就会调用下层发送函数，此函数是发送流程的下层接口函数，在 IPv4 协议模块完成发送封装工作后，调用接口函数进行后续发送操作。后续的发送处理流程包括分片处理，IPv4 地址到 MAC 地址的映射，封装成为帧等。

```
Route temp[33];
//0的时候，没有子网划分，32的时候没有子网内部的主机编号的划分
for(int i=1; i<33; i++){
    temp[i].dest = header.Daddr;
    temp[i].masklen = i;
}

//去 set 集合中查找路由信息
set<Route, RouteSortCriterion>::iterator iter;
//如何在set集合中查找目的IP的节点
int j;
for(j=32; j>0; j--){
    iter = s.find(temp[j]);
    if(iter != s.end()) break;
}
```

```
//没有查找到路由信息
if(iter == s.end()) {
    fwd_DiscardPkt(pBuffer, STUD_FORWARD_TEST_NOROUTE);
    return -1;
}
//找到路由信息，但是TTL信息出错
if (header.ttl <= 1)
{
    fwd_DiscardPkt(pBuffer, STUD_FORWARD_TEST_TTLERROR);
    return -1;
}
//修改TTL并重新计算校验和
pBuffer[8]--;
(*((unsigned short*) (pBuffer+10))) = 0;
(*((unsigned short*) (pBuffer+10))) = checksum((unsigned short*)pBuffer, header.IHL*4);
```

在转发的时候需要区分最后一跳和非最后一跳：

若在路由表中查找到了相应的路由信息，判断目的地址是否和当前的路由器所处的子网的网络编号相同，若相同，则直接发送分组到目的地址，否则需要按照路由表项发送到下一跳指定的路由器地址。

```
//区分处理最后一跳和非最后一跳
if ((header.Daddr ^ getIpv4Address()) & (0xffffffff << (32-j)) == 0x00000000 )
        fwd_SendtoLower(pBuffer, length, header.Daddr);
else
        fwd_SendtoLower(pBuffer, length, (*iter).nexthop);
return 0;
```

四、实验结果

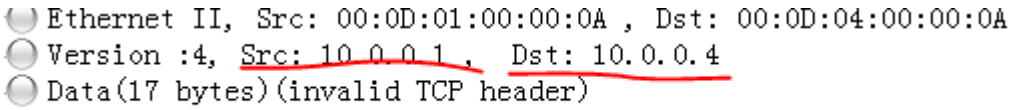
PART 1- IPv4 收发实验

收发数据包截图部分分析比较简单，故此处略去

PART 2_ IPv4 转发实验

本地接收实验

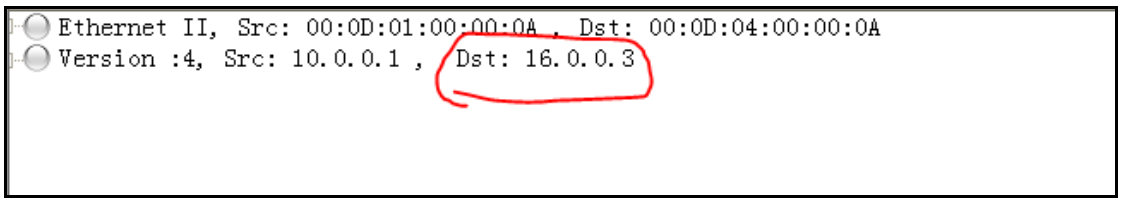
```
begin test!, testItem = 2  testcase = 0
accept len = 32 packet
accept len = 244 packet
accept len = 41 packet
********** START **********
Dest: 11:0:0:2
Masklen: 24
Nexthop: 11:0:0:11
********** END **********
********** START **********
Dest: 12:0:0:3
Masklen: 24
Nexthop: 12:0:0:12
********** END **********
accept len = 55 packet
send a message to main ui, len = 53  type = 2  subtype = 0
accept len = 6 packet
result = 0
send a message to main ui, len = 6  type = 1  subtype = 7
```



分析：目的地址为本机地址的数据包，接收以后传给上层处理。

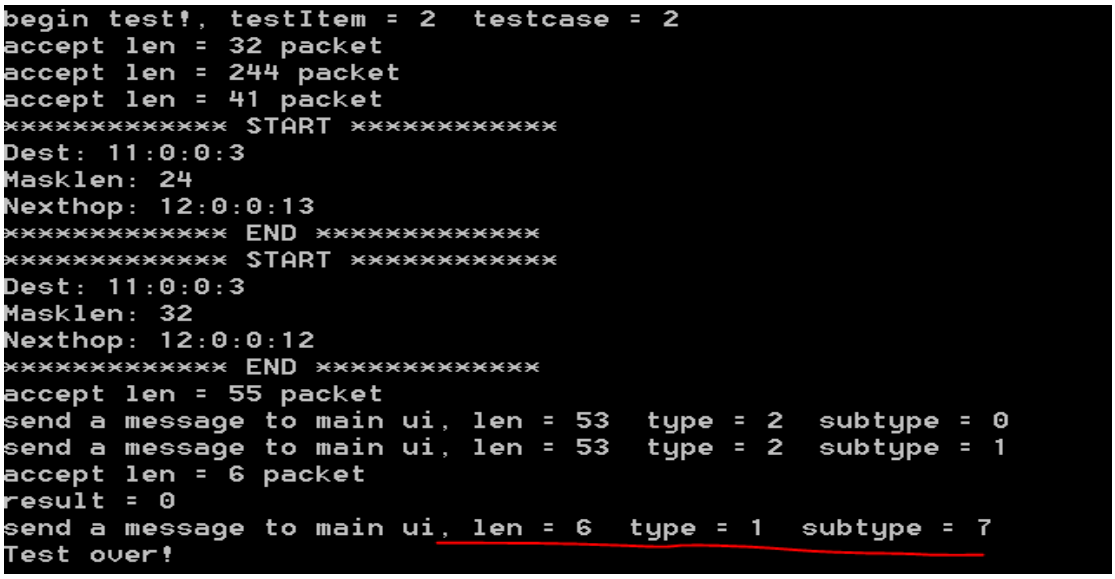
无法获得路由信息

```
begin test!, testItem = 2  testcase = 1
accept len = 32 packet
accept len = 244 packet
accept len = 41 packet
********** START **********
Dest: 11:0:0:2
Masklen: 24
Nexthop: 11:0:0:11
********** END **********
********** START **********
Dest: 12:0:0:3
Masklen: 24
Nexthop: 12:0:0:12
********** END **********
accept len = 38 packet
send a message to main ui, len = 36  type = 2  subtype = 0
accept len = 6 packet
result = 0
send a message to main ui, len = 6  type = 1  subtype = 7
```



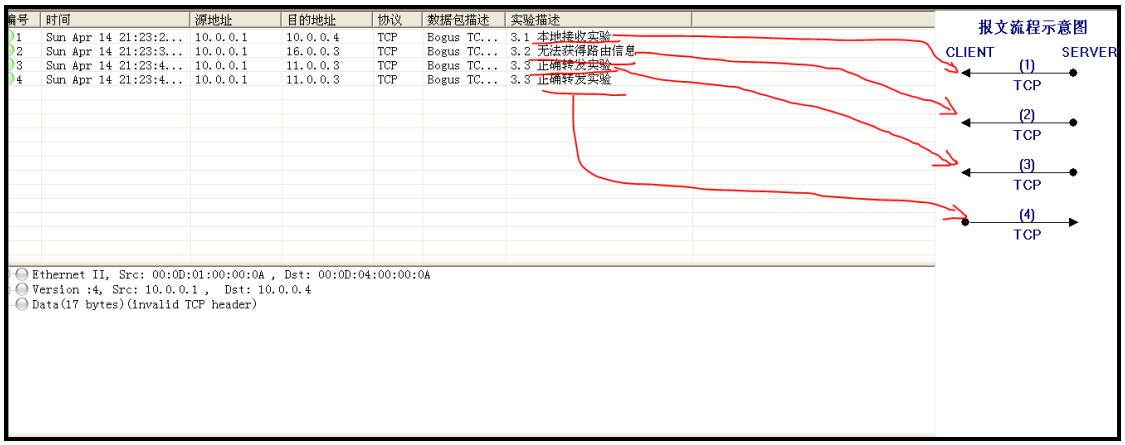
目标地址不是本机地址，需要转发数据报，但是根据控制台输出信息可以知道，当前路由表的信息中没有和 16.0.0.3 对应的路由信息， 所以应该丢弃分组。

正确转发实验



分析：在路由表中查找到了目标地址，然后判断是否是最后一跳，如果是最后一跳，则将此数据报直接发送，否则转发到下一跳。

对应的数据报以及流程图如下图：



五、思考问题

【1】 IP 报文中为什么需要有一个总长度字段：

因为在发送 IP 报文需要封装，添加 MAC 头部中需要保存长度信息，而 IP 报文首部中的总长度值传递给 MAC 帧头中的长度字段。

【2】 IP 报文首部中的校验和的计算采用的是反码和而不是补码和？

-1- 反码和的溢出有后效性（蔓延性）；反码和将高位溢出加到低位，导致这个溢出会对后面的操作有永久影响，有后效性，而补码和直接将高位和溢出，导致这个溢出对后面的操无影响，因此没有后效性。

-2- 反码校验无需考虑字节序，因为反码和的溢出有后效性，导致大端字节序和小端字节序对同一数据序列产生的校验和也只是字节序相反，而补码和因为将溢出丢掉了，不同字节序之间的校验和大部相同且没有什么联系。

【3】 校验和计算式发生在被转换为网络字节序之前还是之后？

校验和运算既可以选择在转换为网络字节序前，也可以选择在之后，只要保证被检验的字段和填写的校验和字段的字节序保持一致就可以。实验中发送时是先转换成网络字节序，然后计算校验和的。接收数据报是将其转换为主机字节序。

【4】 IPv6 首部中为什么将校验和字段取消？

IPv6 的首部没有了 checksum 字段，所以他的上层传输层必须要有 checksum 字段。

IPv6 包的校验依赖高层的协议来完成，这样的好处是免去了 checksum 校验所需要的时间，减小了网络延迟。

【5】 路由表的数据结构设计：

采用了 C++ STL 提供的 set 集合容器，因为他实现了红黑树的平衡二叉搜索树的数据结构，插入元素时，它会自动的调整二叉树的排列，把元素放到适当的位置，以保证每个子树根节点键值大于左子树所有节点的键值，小于右子树所有节点的键值，还要保证根节点左子树和右子树的高度相等。

平衡二叉树使用了中序遍历算法，检索的效率高于 vector, deque 和 list 等容器，另外使用中序遍历可以将键值按照从小到大遍历出来。

因此我采用了 set 来维护路由表结构。

而 set 内部可以避免插入重复路由表项，并且可以自动建立平衡二叉树，所以操作很简单：直接调用 insert( )即可。

```
void stud_Route_Init()
{
    s.clear();
    return;
}

void stud_route_add(stud_route_msg *proute)
{
    cout<<"*****************************"<<endl;
    Route* t = new Route;
    t->dest = ntohl(proute->dest);
    t->masklen = ntohl(proute->masklen);
    t->nexthop = ntohl(proute->nexthop);
    unsigned char *p1 = (unsigned char *) &(t->dest);
    unsigned char *p2 = (unsigned char *) &(t->nexthop);
    cout<<"Dest: "<<(int)p1[3]<<":"<<(int)p1[2]<<":"<<(int)p1[1]<<":"<<(int)p1[0]<<endl;
    cout<<"Masklen: "<<t->masklen<<endl;
    cout<<"Nexthop: "<<(int)p2[3]<<":"<<(int)p2[2]<<":"<<(int)p2[1]<<":"<<(int)p2[0]<<endl;
    cout<<"************ ADD INFO ************"<<endl;

    s.insert(*t);
    return;
}
```

【6】 判断路由表项的目的地址与要路由的目的地址是否属于一个子网：

由于发送的数据报中不存在掩码，同时实际中采用的是与子网号相同的最长地址匹配，所以，在分析的时候直接判断就可以了：分别计算他们的子网号，判断是否相等，按照路由表项的掩码长度来计算目的地址的子网号。

```
Route temp[33];
//0的时候，没有子网划分，32的时候没有子网内部的主机编号的划分
for(int i=1; i<33; i++){
    temp[i].dest = header.Daddr;
    temp[i].masklen = i;
}

//去 set 集合中查找路由信息
set<Route, RouteSortCriterion>::iterator iter;
//如何在set集合中查找目的IP的节点
int j;
for(j=32; j>0; j--){
    iter = s.find(temp[j]);
    if(iter != s.end()) break;
}
```

【7】 转发类型需要分别处理最后一跳和非最后一跳的情况：

在进行最后一跳的时候，路由器在接收到数据包时，会查找路由表来确定下一跳的地址，而查找路由表的时候，路由器只是通过掩码所对应的网络号进行查找，而在进行最后一跳的时候，路由器识别到的是一个同网段的地址，此时，如果路由器仍旧去查路由表，则查到的下一跳很可能是该路由器本身的地址，此时，如果路由器仍旧去查找路由表，则查找到的下一跳应该就是路由器本身的地址，因此在最后一跳的时候，即当路由器发现数据包的目的地址和路由器字段在同一个网段的时候，路由器直接将数据报发送给目标地址。

【8】 真实的网络中实现代码在接收数据包时，如何判断接收到的数据报是否是第一个分

片数据包，中间分片数据包或者最后一个分片数据包；如何处理分片数据包超市问题

分析：IP 分片和完整 IP 报文差不多拥有相同的 IP 头，ID 域对于每个分片都是一致的，这样才能在重新组装的时候识别出来自同一个 IP 报文的分片。在 IP 头里面，16 位识别号唯一记录了一个 IP 包的 ID，具有同一个 ID 的 IP 分片将会重新组装；而 13 位片偏移则记录了某 IP 片相对整个包的位置；而这两个表中间的 3 位标志则标志着该分片后面是否还有新的分片。这三个域就组成了 IP 分片的所有信息，接受方就可以利用这些信息对 IP 数据进行重新组织。

标志字段在分片数据报中起了很大作用，在数据报分片时把它的值复制到每片中。标志字段的其中一个比特称作“不分片”位，用其中一个比特来表示“更多的片”。除了最后一片外，其他每个组成数据报的片都要把该比特置 1。片偏移字段指的是该片偏移原始数据报开始处的位置。另外，当数据报被分片后，每个片的总长度值要改为该片的长度值。如果将标志字段的"不分片"比特置 1，则 IP 将不对数据报进行分片。

如果分片数据包某一片超时，其他的分片数据包都需要丢弃，然后重传所有的分片数据包。

六、 实验中遇到的问题

主要是在设计路由表项的数据结构的时候，采用链表的性能太差，时间复杂度是 O(n)的，采用了 set 容器后，根据是最长掩码匹配的原则可以依次搜索掩码长度为 32 递减到 1 时间复杂度是 O(32*lgn);

七、 实验的启示/意见和建议

最近这周忙到想撞墙了！由于最近在做编译原理实验，基本上都是自己设计的，熬了好几夜调试分析，虽然把实验做完了，但是一方面实验平台有问题，另外一方面感觉实验中自己实现的都很表层，不过通过对 IPv4 收发和转发的实现主要加深了对它头部的了解以及校验和计算！

但是问题的难点是不明白校验和计算的原理，以及路由表表项的设计原理（本质），我们设计的这种静态路由应该是最无脑的吧，现实中应该没有任何意义吧。

唯一感觉有收获的地方就是查找了很多关于 set 内部实现机制的博客，并重载了 set 容器的比较设置，然后采用 set 的话，在插入的时候直接调用系统函数就可以了可以避免繁琐的判断和查找尾节点等操作。

最后总结一下这个辛苦的周末的感受就是当程序员这是过的人的日子吗？？？

额，不过还是要脚踏实地，继续努力，为了自己的梦想！