哈爾濱工業大學
Harbin Institute of Technology

计算机网络
课程实验报告

实验名称	IPv4 分组的收发和转发				
姓名	刘璟烁		院系	计算机科学与技术学院	
班级	1803104		学号	1180500301	
任课教师	聂兰顺		指导教师	聂兰顺	
实验地点	格物 207		实验时间	2020-11-14	
实验课表现	出勤、表现得分(10)		实验报告得分(40)	实验总分	
	操作结果得分(50)				
教师评语					

计算机科学与技术学院 SINCE 1956...
School of Computer Science and Technology

实验目的:

1.**IPv4 分组收发实验**:

IPv4 协议是互联网的核心协议，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。IPv4 地址唯一标识了网络节点和网络的连接关系。在我们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报文封装为 IPv4 分组发送出去。

本实验通过设计实现主机协议栈中的 IPv4 协议，让学生深入了解网络层协议的基本原理，学习 IPv4 协议基本的分组接收和发送流程。另外，通过本实验，学生可以初步接触互联网协议栈的结构和计算机网络实验系统，为后面进行更为深入复杂的实验奠定良好的基础。

2.IPv4 分组转发实验:

通过前面的实验，我们已经深入了解了 IPv4 协议的分组接收和发送处理流程。本实验需要将实验模块的角色定位从通信两端的主机转移到作为中间节点的路由器上，在 IPv4 分组收发处理的基础上，实现分组的路由转发功能。

网络层协议最为关注的是如何将 IPv4 分组从源主机通过网络送达目的主机，这个任务就是由路由器中的 IPv4 协议模块所承担。路由器根据自身所获得的路由信息，将收到的 IPv4 分组转发给正确的下一跳路由器。如此逐跳地对分组进行转发，直至该分组抵达目的主机。IPv4 分组转发是路由器最为重要的功能。

本实验设计模拟实现路由器中的 IPv4 协议，可以在原有 IPv4 分组收发实验的基础上，增加 IPv4 分组的转发功能。对网络的观察视角由主机转移到路由器中，了解路由器是如何为分组选择路由，并逐跳地将分组发送到目的主机。本实验中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对分组进行转发的。

实验内容:

1)实现 IPv4 分组的基本接收处理功能对于接收到的IPv4分组，检查目的地址是否为本地地址，并检查IPv4分组头部中其它字段的合法性。提交正确的分组给上层协议继续处理，丢弃错误的分组并说明错误类型。

2)实现 IPv4 分组的封装发送根据上层协议所提供的参数，封装 IPv4 分组，调用系统提供的发送接口函数将分组发送出去。

3)设计路由表数据结构。设计路由表所采用的数据结构。要求能够根据目的 IPv4 地址来确定分组处理行为（转发情况下需获得下一跳的 IPv4 地址）。路由表的数据结构和查找算法会极大的影响路由器的转发性能，有兴趣的同学可以深入思考和探索。

4) IPv4 分组的接收和发送。对前面实验（IP 实验）中所完成的代码进行修改，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。具体要求不做改变，参见“IP 实验”。

5) 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 IP 地址，然后调用发送接口函数做进一步处理。

实验过程:

1.剖析实验原理

(1).IPv4收发数据报原理

在主机协议栈中，IPv4 协议主要承担辨别和标识源 IPv4 地址和目的IPv4 地址的功能，一方面接收处理发送给自己的分组，另一方面根据应用需求填写目的地址并将上层报文封装发送。IPv4 地址可以在网络中唯一标识一台主机，因而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。

一个 IPv4 数据报文由首部和数据两部分组成。首部的前一部分是固定长度，共 20 个字节，是所有IPv4 数据报必须具有的。在首部的固定部分的后面是一些可选字段，其长度是可变的。具体的报文格式如图1：

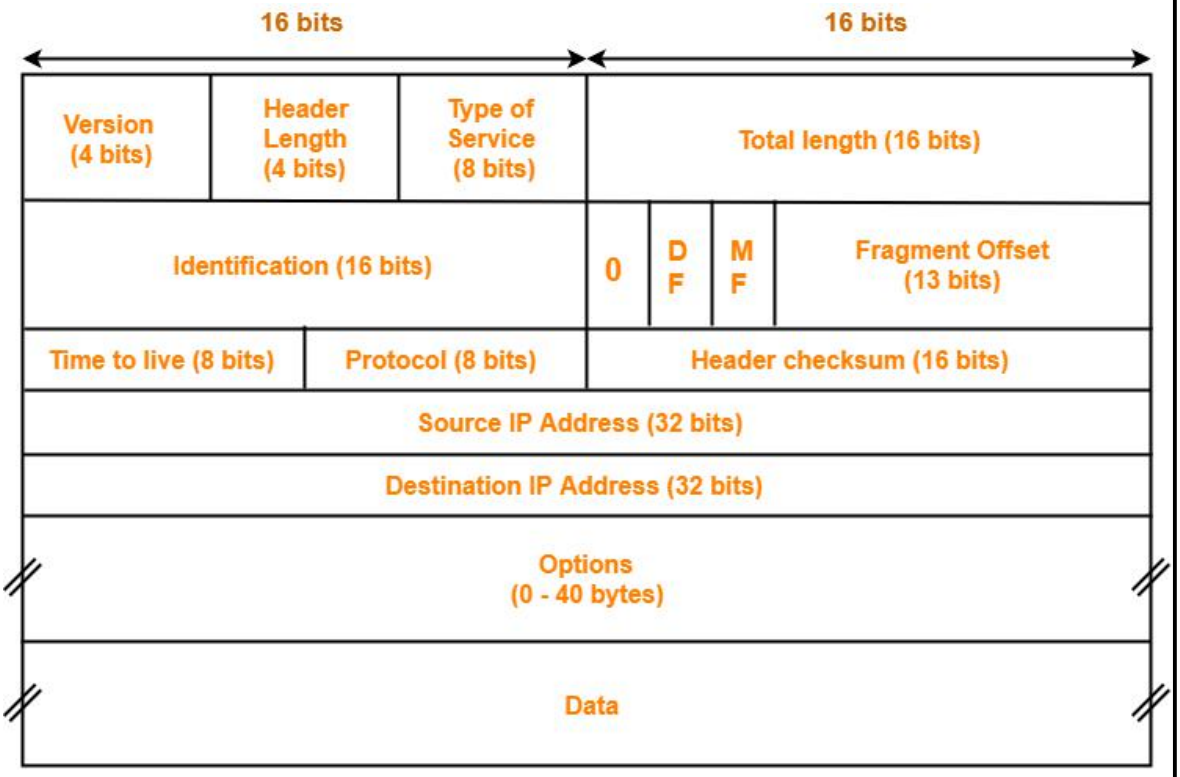


图 1.IPv4 数据报首部格式

其中，Version 字段占 4 bits，显示当前正在运行的 IP 版本信息。
HL 字段占 4 bits，标明了以 32 比特为单位的消息中数据报报头的长度，这是所有报头的总长度。应该注意，它以四字节为单位计数，最小值为 5。
TypeOfService 占 8bits，一般不应用。
Total Length 占 16 bits，标明整个分组的长度，以字节为单位。总长度减去 IHL 就是数据有效载荷的长度。理论上最大长度可以为 65535，但报文经过数据链路层时很可能被分片。
Identification 占 16 bits，包含一个整数，用来标识当前的数据报。这是一个序列号。
Flag:占 3 bits，实现不同的关于分片的控制功能。
Fragment Offset 占 13 bits。指示分段在数据包中的位置，用于重组数据分段。这个字段允许标记字段终止在 16 Bit 的边界
TTL 占 8 bits，指示分组在网络传输中的允许保持的时间，以秒为单位。当该计时器为 0 时，数据报将被丢弃。
Protocol :占 8 bits，指明了在 IP 处理过程结束后，哪一个上层协议将接收这些数据。
Header Checksum 占 16 bits，用于确保 IP 头的完整性。
Source Address 占 32 bits，指明了发送节点的 IP 地址。
Destination Address 占 32 bits，指明了接收节点的 IP 地址。
在IPv4数据报的接收任务中，需要对接收到的数据报对应的字段（比如TTL、checksum、version）按照上述机制进行检查，如果有违法的值则说明该IPv4数据报不应予以接收，反之可以接收并调用相关接口函数转发给上层。

(2).IPv4收发数据报原理

分组转发是路由器最重要的功能。分组转发的依据是路由信息，以次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，路由器协议栈的IPv4协议功能能够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。

为了实现功能，需要了解以下不同种类型数据报的区别：

主机单播IP报：上层协议要求发送数据包时，目的是否直连，是解析目的主机硬件地址（ARP工作原理），否解析网管硬件地址，封装成帧并从相应接口发出。

路由单播IP报：数据报入站，是否本机（不是查询路由表），是否直连。直连解析目的主机的硬件地址，非直连解析下一跳路由器的硬件地址，封装帧发出。

主机接收IP报：1.目的地址等于本机IP地址，2.目的地址是一个广播地址，3.目的地址为组播地址，本机属于该组播组，否丢弃数据包。

因此，从协议栈层面，我们需要实现路由器在控制平面的功能。即给定一个的目的IP地址，通过查询由静态配置或动态配置（本次实验不涉及）构建的路由表确定下一跳的方向，具体的，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，下图为主机中的路由表：

Routing Tables: Public
Destinations : 10 Routes : 19

Destination/Mask	Proto	Pre	Cost	Flags	NextHop	Interface
127.0.0.0/8	Direct	0	0	D	127.0.0.1	InLoopBack0
127.0.0.1/32	Direct	0	0	D	127.0.0.1	InLoopBack0
127.255.255.255/32	Direct	0	0	D	127.0.0.1	InLoopBack0
192.168.10.0/24	OSPF	10	2	D	192.168.40.254	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.4	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.2	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.5	GigabitEthernet0/0/0
192.168.20.0/24	OSPF	10	2	D	192.168.40.254	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.4	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.2	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.5	GigabitEthernet0/0/0
192.168.30.0/24	OSPF	10	2	D	192.168.40.254	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.4	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.2	GigabitEthernet0/0/0
	OSPF	10	2	D	192.168.40.5	GigabitEthernet0/0/0
192.168.40.0/24	Direct	0	0	D	192.168.40.1	GigabitEthernet0/0/0
192.168.40.1/32	Direct	0	0	D	127.0.0.1	GigabitEthernet0/0/0
192.168.40.255/32	Direct	0	0	D	127.0.0.1	GigabitEthernet0/0/0
255.255.255.255/32	Direct	0	0	D	127.0.0.1	InLoopBack0

图 2.路由表

本实验中主要关注分组的目的IP地址以及下一跳IP地址，分别对应图中的Destination和Nexthop字段。

2.程序流程图

(1).接收函数stud_ip_recv()程序流程图：

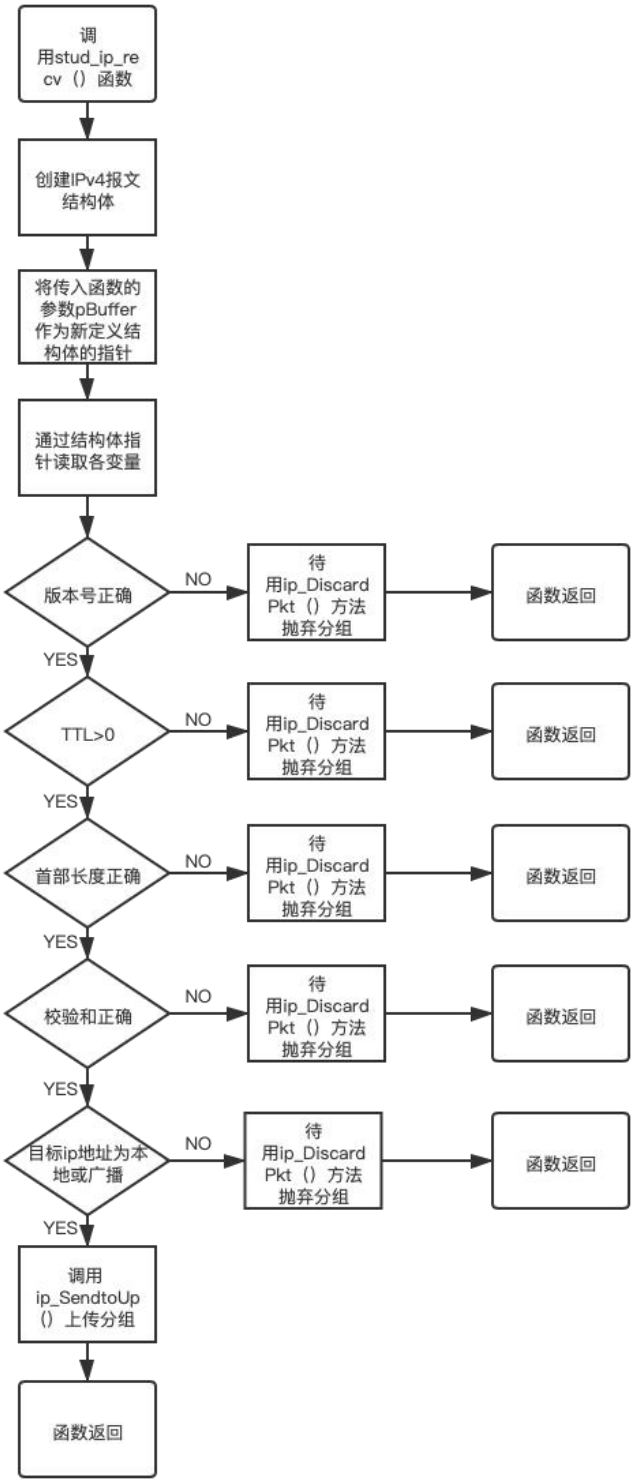


图3.接收函数stud_ip_recv()程序流程图

(2).分组发送函数stud_ip_Upsend()程序流程图：

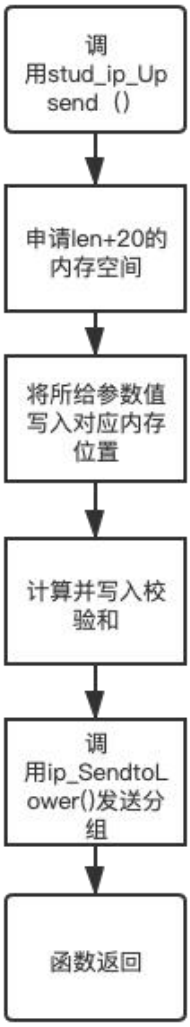


图4.发送函数stud_ip_Upsend()流程图

(3).路由表初始化stud_Route_Init()和调价函数stud_Route_add()流程图：

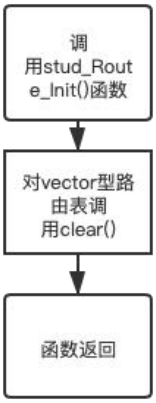


图5.stud_Route_Init()流程图

图6.stud_Route_add()流程图

(4).转发函数stud_fwd_deal()流程图：

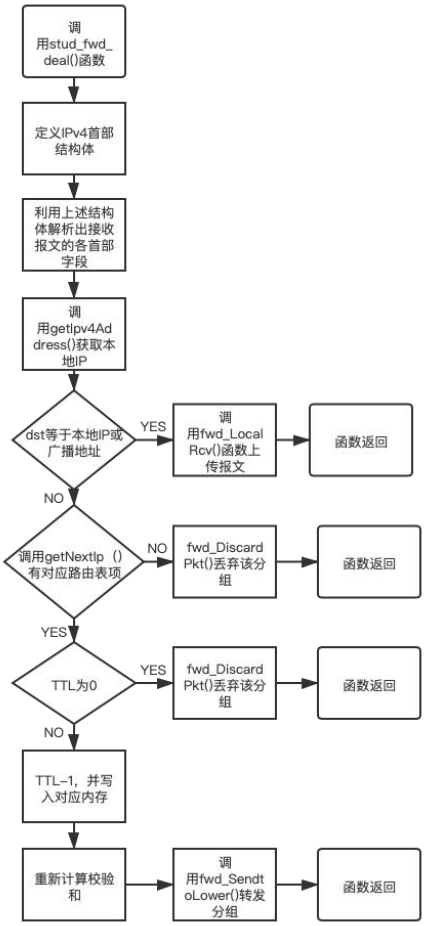


图7.stud_fwd_deal()流程图

3.实验各要求实现过程

(1).**IPv4数据报收发基本功能实现**

a.接收函数stud_ip_recv((char *pBuffer,unsigned short length)实现：

首先，为解析传入的分组首部各字段内容，定义一个模拟IPv4报文首部的结构体IPV4如下：

```
typedef struct{
        char ver_ihl;   // 第一字节，包含版本号和首部长度
        char service_type;
        unsigned short total_lengtg;   // 报文总长度
        unsigned int iden_off;
        char TTL;   //跳步数
        char protocal;
        unsigned short header_checksum;   // 校验和
        unsigned int src_ip;   // 源ip
        unsigned int trg_ip;   // 目的ip
    }IPV4;
```

随后将pBuffer赋给一个IPV4结构体作为指针，并根据其字段内容赋值给变量version等（注意超过一个字节的字段内容需要调用ntohl()函数调整字节序），根据合法报文的要求，我们做如下四个判断，并在不满足的情况下丢弃：

```
    if(version != 4){   //非ipv4报文则丢弃
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_VERSION_ERROR);
        return 1;
    }

    if(head_length < 5){   //header length  比5小则丢弃
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_HEADLEN_ERROR);
        return 1;
    }

    if(TTL <= 0){   //TTL违法则丢弃
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_TTL_ERROR);
        return 1;
    }

    if( !(trg_IP == getIpv4Address() || trg_IP == 0xffffffff)){   //判断目的IP是否为本机IP或广播地址
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_DESTINATION_ERROR);
        return 1;
    }
```

如果上述判断均没有问题，我们再计算报文的校验和是否正确，具体的，将bBuffer指向的区域按照两字节为单位进行读取，累加、溢出回卷并最终取反，最终结果不是0xffff则校验和错误，

抛弃。
否则可以接收，调用ip_SendtoUp()函数即可。

b. 发送函数 stud_ip_Upsend(char *pBuffer,unsigned short len,unsigned int srcAddr,unsigned int dstAddr,byte protocol,byte ttl)的实现：
为了向目的ip地址实现发送报文的功能，我们首先需要将上层传来的数据分组封装为ip数据报的格式，因此需要申请一个len+20的空间用于存储ip数据报首部和数据部分的内容：

```
char *pIpv4 = (char*)malloc((len+20)*sizeof(char));
```

随后向该地址空间中对应部分写入IPv4数据报各字段内容即可：

```
pIpv4[0] = 0x45;
...
memcpy(pIpv4+16,&trg_ip,4);  //将目标IP地址存入该区域
```

注意，这里还未写入校验和，需要对上述字段进行校验和计算再写入对应字段，具体计算方法同上，最后调用ip_SendtoLower()函数发送即可。

c.IPv4转发实验——路由数据结构实现：
现在说明lab5中路由表的数据结构的实现方法，首先需要定义路由表中存储的路由表项，首先我们列出系统调用表项添加函数时作为参数的结构体：

```
typedef struct stud_route_msg
{
    unsigned int dest;
    unsigned int masklen;
    unsigned int nexthop;
} stud_route_msg;
```

可知其中给定了目的ip、下一跳ip、子网掩码，则我们可定义路由表项数据结构如下：

```
struct route_struct
{
    uint low;   //子网范围下限
    uint high;   //子网范围上限
    uint masklen;   //子网掩码
    uint next_ip;    //下一跳ip
    route_struct(uint low,uint high,uint masklen,uint next_ip){
       this->low = low;
       this->high = high;
       this->masklen = masklen;
       this->next_ip = next_ip;
}
```

即给定一个目的ip，我们可以查询是否有一个路由表项使得该目的ip在这个路由表项的子网范围之中（即大于下限、小于上限），路由表项可存于vector中：

```
vector <route_struct> router_table;
```

d.路由表初始化、添加表项实现：
该部分实现简单，只需调用router_table.clear()即可初始化。
通过遍历vector中的表项，寻找子网范围包含目的地址且掩码最长的表项即可。

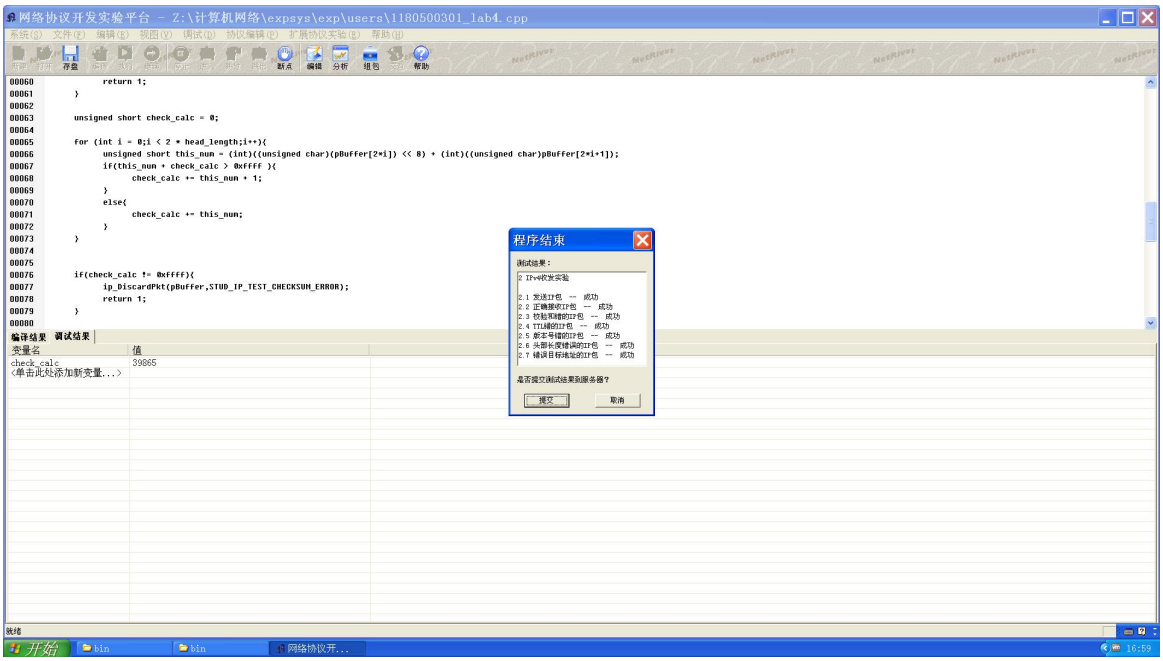
e.转发函数stud_fwd_deal(char *pBuffer, int length)实现：
类似前面给出的报文接收函数，首先应用自定义的IPv4结构体解析出各字段的内容。如果解析出的目的ip为广播地址或本机，则调用fwd_LocalRcv()上传报文。
否则，查询路由表，若不能找到对应表项，抛弃该分组；否则校验TTL合法与否，若合法则减一并写入相应字段：

```
TTL =TTL -  1;
pBuffer[8] = (unsigned char)(TTL & 0xff);
```

随后重现计算校验和，计算方法如上，完成后调整字节序并写入相应空间。
调用fwd_SendtoLower()即可完成转发功能。

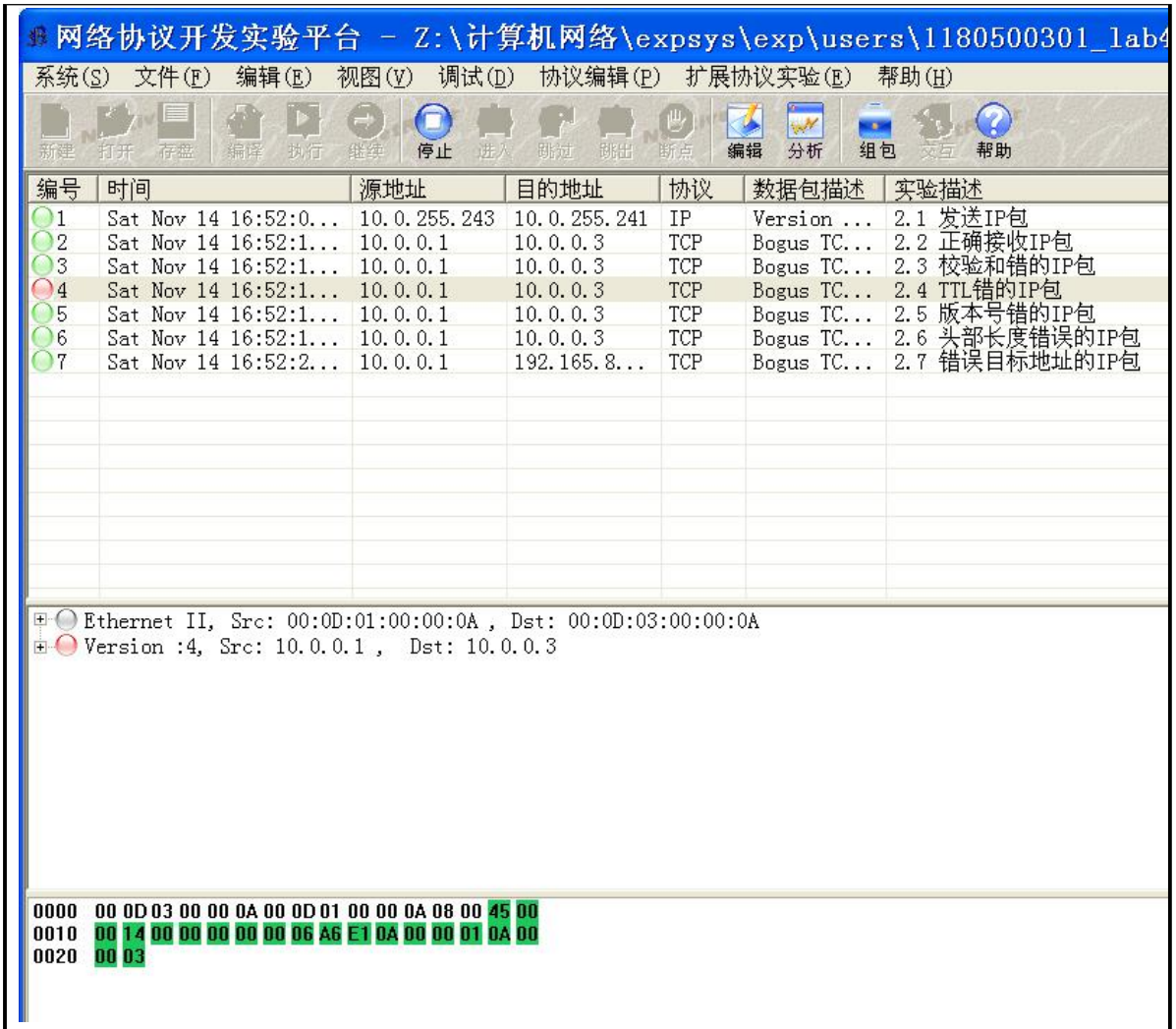
实验结果：

采用演示截图、文字说明等方式，给出本次实验的实验结果。
1.IPv4收发实验：
登陆NetRiver系统，选择对应实现内容并执行对应程序后，得到如下结果：



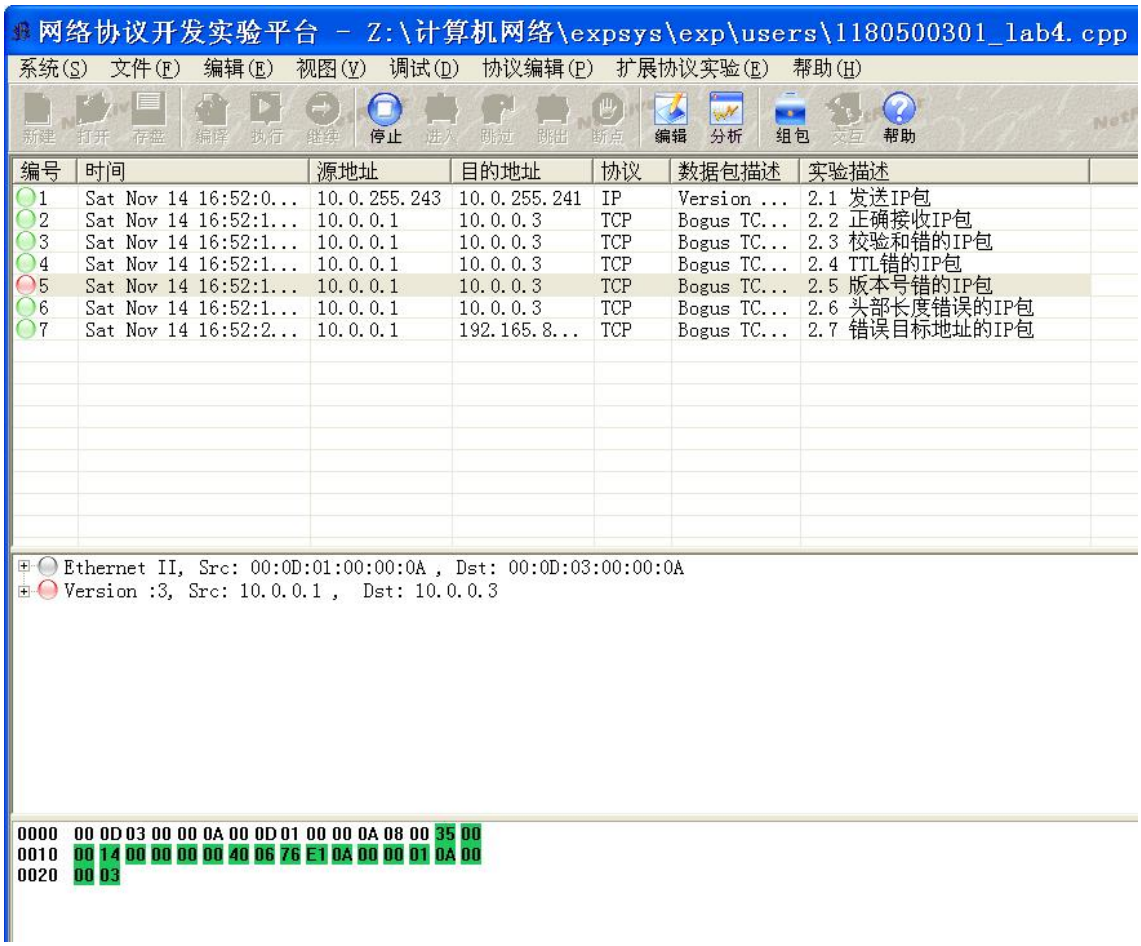
可以看到，所有的任务均已执行成功，下面具体说明TTL等任务的错误报文。

a.ttl不合法：



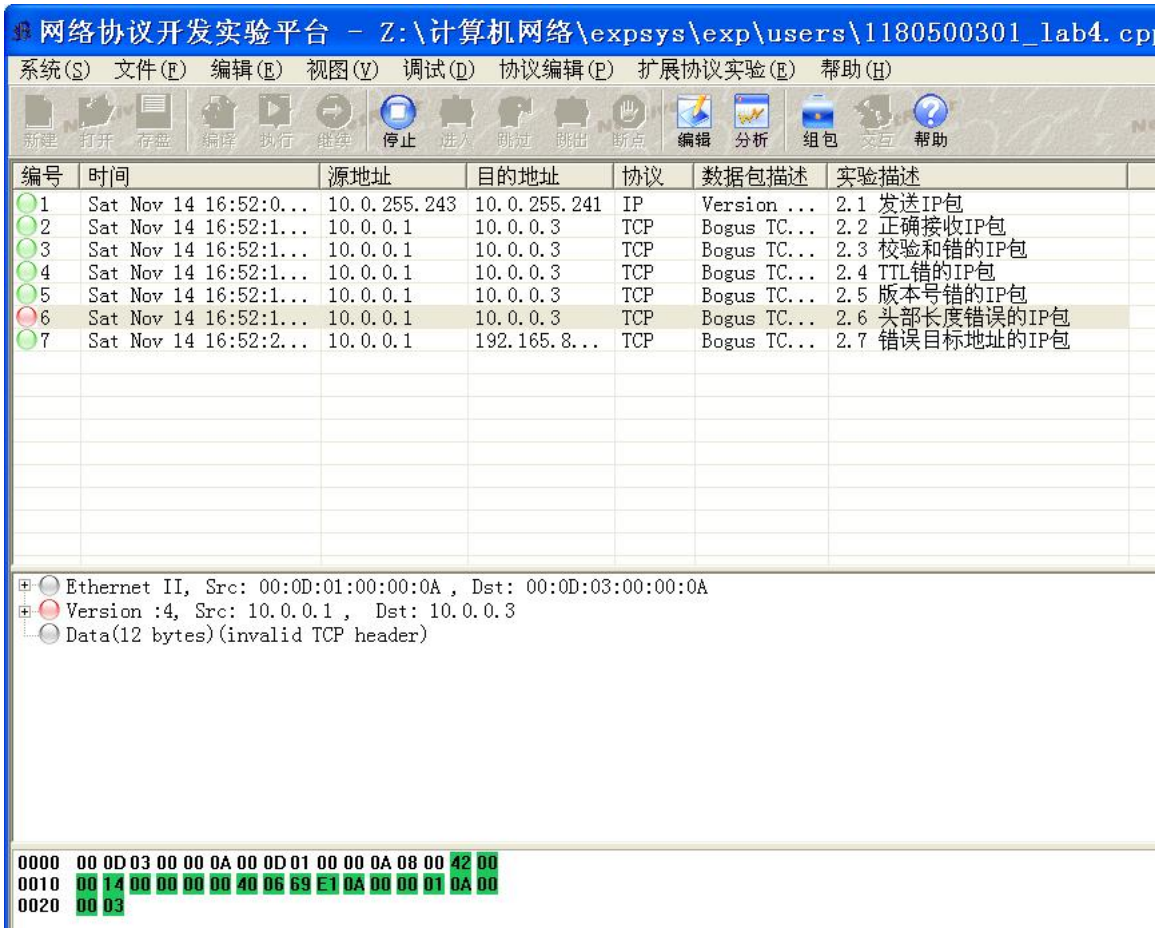
可以看到，在绿色（ip数据报首部）第9个字节处TTL为0，不是合法的数据报，应该丢弃。

b.版本号错误：



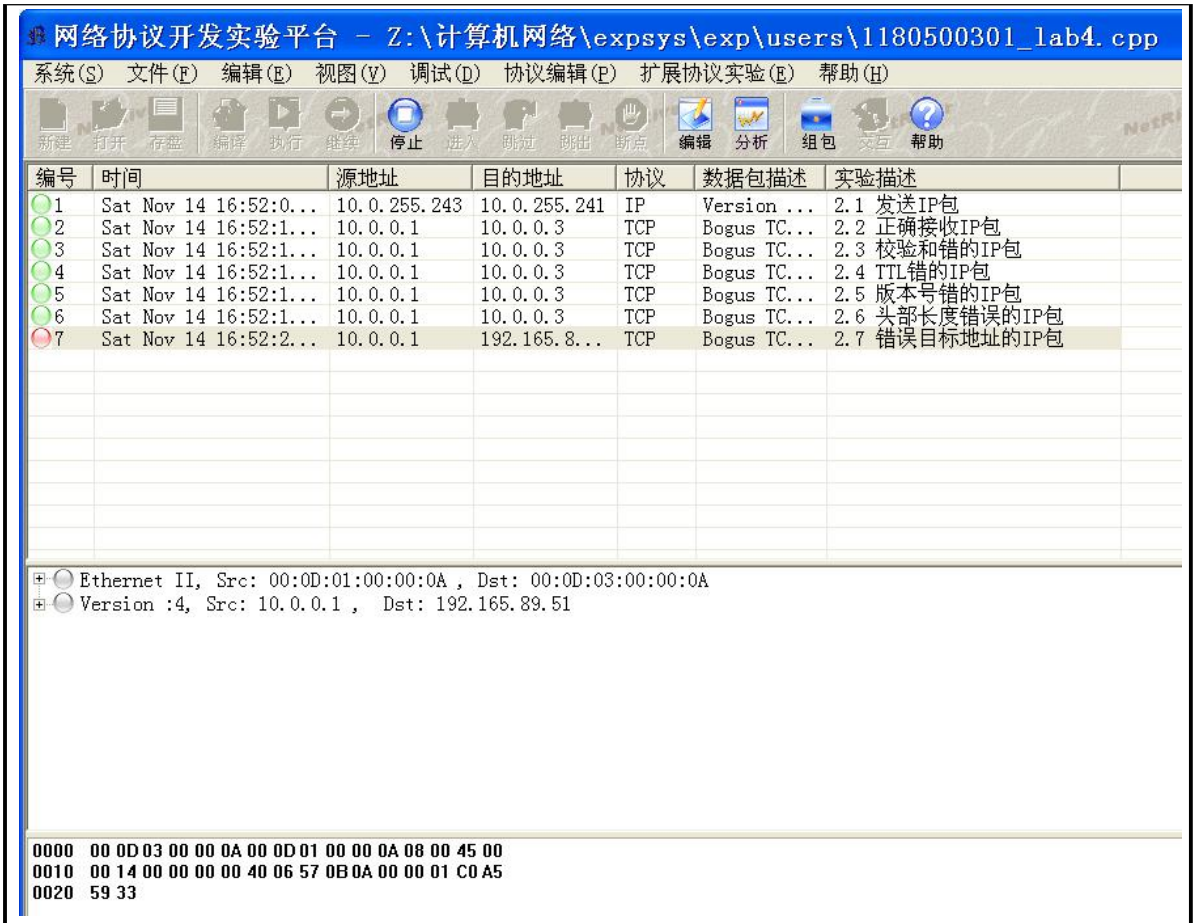
可以看到绿色部分第一个字节为35而非45，并非ipv4的版本号，故错误。

c.头部长度错误：



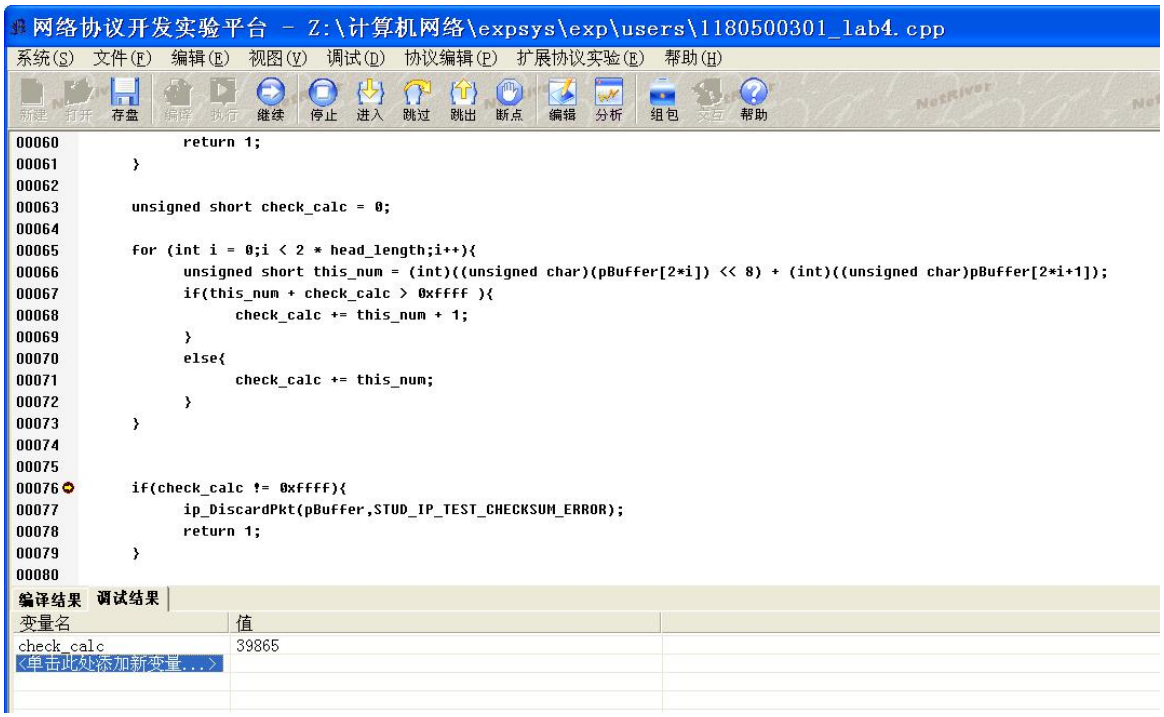
可以看到第一个字节为42，即首部长度为2而不是5，这是错误的。

d.错误目标地址：



可以看到解析出的目标地址为192.165.89.51，而非之前我们已经看到的本机ip地址10.0.0.3，ip地址错误。

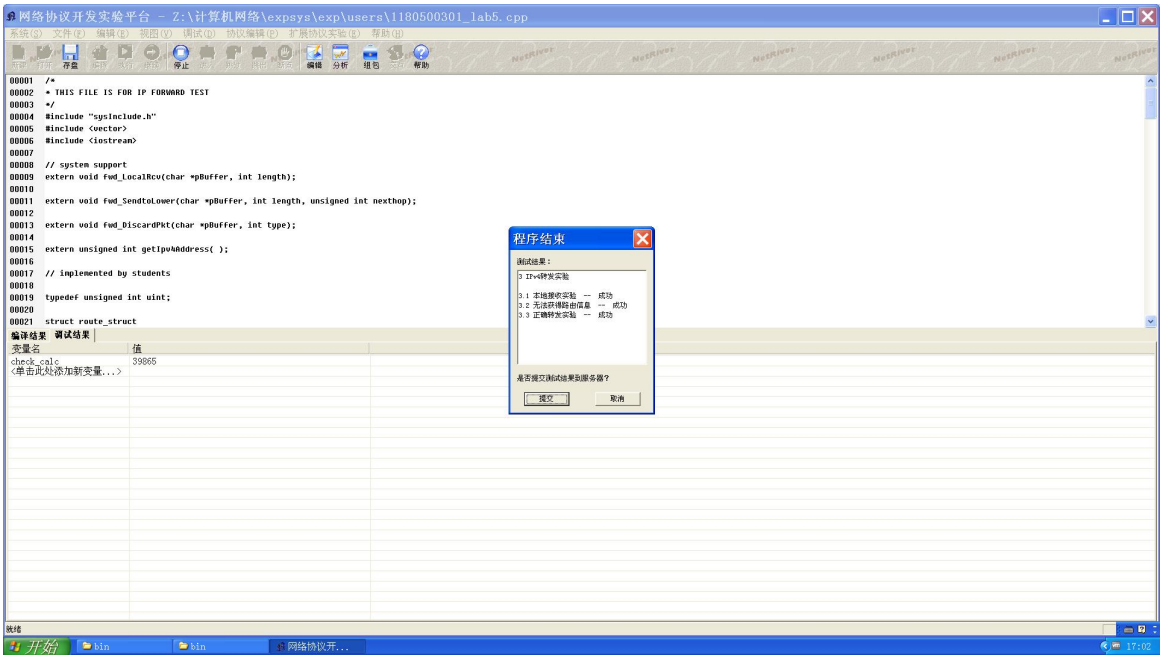
e.校验和错误：



我们可以看到，在IPv4首部的全部20字节校验和累加后，并没有得到0xffff，而是39865，校验和错误。

2.IPv4转发实验：

我们给出程序执行成功的图示：



心得体会：

(1) 对实验指导书中的问题：“在存在大量分组的情况下如何提高转发效率”的分析：

我们在实验中使用了vector作为路由表的主体存储结构，并通过循环遍历来寻找对应的路由表项。当在存在大量分组时，该结构的转发效率显然是较低的，因此我们可以将总体结构改进为trie树的形式，根据子网范围的大小顺序将节点插入对应位置，从而可实现查找时间从线性时间降至对数实现的提升。

(2) 掌握了IPv4报文格式的实现方法，学习了数据报的接收、发送及转发过程具体实现方法。对网络层有了更深刻的了解。

(3).通过自定义IPv4、应用vector等数据结构实现路由表及解析报文功能，增强了相应工程能力。

源代码：

Lab4: lab4.cpp

```
#include "sysInclude.h"
#include <stdio.h>
#include <stdlib.h>


extern void ip_DiscardPkt(char* pBuffer,int type);

extern void ip_SendtoLower(char*pBuffer,int length);

extern void ip_SendtoUp(char *pBuffer,int length);

extern unsigned int getIpv4Address();

// implemented by students

int stud_ip_recv(char *pBuffer,unsigned short length)    // pBuffer 为指向接收缓冲区的指针 length: the length of ipv4 group
{
    typedef struct{
    char ver_ihl;
    char service_type;
    unsigned short total_lengtg;
    unsigned int iden_off;
    char TTL;
    char protocal;
    unsigned short header_checksum;
    unsigned int src_ip;
    unsigned int trg_ip;
    }IPV4;
```

```
    IPV4* recv_point = (IPV4*)pBuffer;
    IPV4 recv_messa = *recv_point;

    int    version    = recv_messa.ver_ihl >> 4 ; //版本号
    int head_length = recv_messa.ver_ihl & 0xf;
    int TTL = recv_messa.TTL;
    int checksum = recv_messa.header_checksum;
    int trg_IP = ntohl(recv_messa.trg_ip);

    if(version != 4){
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_VERSION_ERROR);
        return 1;
    }

    if(head_length < 5){    //header length
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_HEADLEN_ERROR);
        return 1;
    }

    if(TTL <= 0){    //TTL
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_TTL_ERROR);
        return 1;
    }

    if( !(trg_IP == getIpv4Address() || trg_IP == 0xffffffff)){    //判断目的IP是否为本机IP或广播地
址
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_DESTINATION_ERROR);
        return 1;
    }

    unsigned short check_calc = 0;

    for (int i = 0;i < 2 * head_length;i++){
        unsigned short this_num = (int)((unsigned char)(pBuffer[2*i]) << 8) + (int)((unsigned
char)pBuffer[2*i+1]);
        if(this_num + check_calc > 0xffff ){
            check_calc += this_num + 1;
        }
        else{
            check_calc += this_num;
        }
    }
```

```
    if(check_calc != 0xffff){
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_CHECKSUM_ERROR);
        return 1;
    }

    ip_SendtoUp(pBuffer,length);   //没问题就上传数据
    return 0;
}

int stud_ip_Upsend(char *pBuffer,unsigned short len,unsigned int srcAddr,
                   unsigned int dstAddr,byte protocol,byte ttl)
{
    char *pIpv4 = (char*)malloc((len+20)*sizeof(char));
    memset(pIpv4,0,len+20);
    pIpv4[0] = 0x45;
    unsigned short total_length = htons(len + 20);
    memcpy(pIpv4+2,&total_length , 2);
    unsigned int random_iden = rand() % 65535;
    memcpy(pIpv4+4,&random_iden,4);
    pIpv4[8] = ttl;
    pIpv4[9] = protocol;
    unsigned int src_ip = htonl(srcAddr);
    unsigned int trg_ip = htonl(dstAddr);
    memcpy(pIpv4+12,&src_ip,4);   //将源IP地址存入该区域
    memcpy(pIpv4+16,&trg_ip,4);   //将目标IP地址存入该区域

    unsigned short check_calc = 0;
    for (int i = 0;i < 2 * 5;i++){
        unsigned short this_num = ((unsigned char)(pIpv4[2*i]) << 8) + ((unsigned
char)pIpv4[2*i+1]);
        if(this_num + check_calc > 0xffff ){
            check_calc += this_num + 1;
        }
        else{
            check_calc += this_num;
        }
    }
    unsigned short check_s = htons(0xffff-check_calc);
    memcpy(pIpv4+10,&check_s,2);
    memcpy(pIpv4+20,pBuffer,len);
    ip_SendtoLower(pIpv4,len+20);

    return 0;
```

```
}



Lab5:                    lab5.cpp
#include "sysInclude.h"
#include <vector>
#include <iostream>

// system support
extern void fwd_LocalRcv(char *pBuffer, int length);

extern void fwd_SendtoLower(char *pBuffer, int length, unsigned int nexthop);

extern void fwd_DiscardPkt(char *pBuffer, int type);

extern unsigned int getIpv4Address( );

// implemented by students

typedef unsigned int uint;

struct route_struct
{
uint low;
uint high;
uint masklen;
uint next_ip;

route_struct(uint low,uint high,uint masklen,uint next_ip){
this->low = low;
this->high = high;
this->masklen = masklen;
this->next_ip = next_ip;
}

};



vector <route_struct> router_table;

uint get_low(uint dst_ip,uint masklen)    // ×ÓÍøÏÂÎÏÞ
```

```
{
	masklen = 32 - masklen;
	uint low = dst_ip >> masklen;
	low = low<<8;
	return low;

}

uint get_high(uint dst_ip,uint masklen)    // ×ÓÍøÉÏÏÞ
{
	masklen = 32 - masklen;
	uint high = dst_ip | ((1 << masklen) - 1);
	return high;
}

void stud_Route_Init()
{
	router_table.clear();
	return;
}

void stud_route_add(stud_route_msg *proute)
{
	uint dest = ntohl(proute->dest);
	uint masklen = ntohl(proute->masklen);
	uint nexthop = ntohl(proute->nexthop);
	uint low = get_low(dest,masklen);
	uint high = get_high(dest,masklen);
	route_struct new_rou = route_struct(low,high,masklen,nexthop);
	router_table.push_back(new_rou);
	return;
}

bool get_next(uint dst,uint &nextIP)
{
	uint len = 0;
	bool result = false;
	for(int i = 0; i < router_table.size(); i++ )
	{
		route_struct thisone = router_table[i];
		if( thisone.low <= dst && dst <= thisone.high){
			if(thisone.masklen > len){
				len = thisone.masklen;
				nextIP = thisone.next_ip;
```

```
                        result = true;
                    }
                }
            }
        return result;
}

int stud_fwd_deal(char *pBuffer, int length)
{   typedef struct{
        char ver_ihl;
        char service_type;
        unsigned short total_lengtg;
        uint iden_off;
        char TTL;
        char protocal;
        unsigned short header_checksum;
        uint src_ip;
        uint trg_ip;
        }IPV4;

        IPV4* recv_point = (IPV4*)pBuffer;
        IPV4 recv_messa = *recv_point;
        uint TTL = recv_messa.TTL;
        uint checksum = recv_messa.header_checksum;
        uint trg_IP = ntohl(recv_messa.trg_ip);

        uint local_ip = getIpv4Address();
        if (trg_IP == local_ip || trg_IP == 0xffffffff){
            fwd_LocalRcv(pBuffer,length);
            return 0;
        }

        uint next_ip;
        if (get_next(trg_IP,next_ip)){
            if(TTL <= 0 ){ // ¼ì²éTTL
                fwd_DiscardPkt(pBuffer, STUD_FORWARD_TEST_TTLERROR);
                return 1;
            }
            TTL =TTL -    1;
            pBuffer[8] = (unsigned char)(TTL & 0xff);

            unsigned short check_calc = 0;
            for (int i = 0;i < 2 * 5;i++){
                if(i != 5){
```

```
                    unsigned short this_num = ((unsigned short )(pBuffer[2*i] << 8) + ((unsigned
short )pBuffer[2*i + 1]));
                        if(this_num + check_calc > 0xffff ){
                        check_calc += this_num + 1;
                    }
                    else{
                        check_calc += this_num;
                    }
                }
            }
            unsigned short check_s = htons(0xffff-check_calc);
            memcpy(pBuffer+10,&check_s,2);

            fwd_SendtoLower(pBuffer, length, next_ip);
                return 0;
            }
        else{
            fwd_DiscardPkt(pBuffer, STUD_FORWARD_TEST_NOROUTE);
                return 1;
        }
}
```