<<计算机网络>>

实验报告

(2017 年度春季学期)

姓名:	赵浩宁
学号:	1140310226
学院:	计算机科学与技术学院
教师:	聂兰顺

一、实验目的

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。

二、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。

三、实验过程及结果

根据计算机网络实验系统所提供的上下层接口函数和协议中分组收发的主要流程，独立设计实现一个简单的 IPv4 分组收发模块。要求实现的主要功能包括：

1) IPv4 分组的基本接收处理，能够检测出接收到的 IP 分组是否存 在如下错误：校验和错、TTL 错、版本号错、头部长度错、错误 目标地址；
2) 2) IPv4 分组的封装发送；

注：不要求实现 IPv4 协议中的选项和分片处理功能

1. Ipv4 分组头部格式

表1 IPv4报文各字段的含义

字段	长度	含义
版本	4比特	IP协议的版本号，分为IPv4和IPv6协议。
首部长度	4比特	IPv4的首部长度。
区分服务	8比特	用来获得更好的服务。只有在使用区分服务时，这个字段才起作用。
总长度	16比特	指首部和数据之和的长度。
标识	16比特	IPv4软件在存储器中维持一个计数器，每产生一个数据报，计数器就加1，并将此值赋给标识字段。
标志	3比特	目前只有两位有意义。最低位为1表示后面“还有分片”的数据报，为0表示这已经是最后一个数据片；中间一位为1表示“不能分片”，为0才允许分片。
片位移	13比特	指出较长的分组在分片后，该片在原分组中的相对位置。
生存时间TTL（Time To Live）	8比特	表示数据报在网络中的寿命，功能是“跳数限制”。
协议	8比特	指出此数据报携带的数据是使用何种协议。
首部检验和	16比特	数据报每经过一个设备，设备都要重新计算一下首部检验和，若首部未发生变化，则此结果必为0，于是就保留这个数据报。这个字段只检验数据报的首部，但不包括数据部分。
源地址	32比特	报文发送方的IPv4地址。
目的地址	32比特	报文接收方的IPv4地址。
选项字段	0~40字节（长度可变）	用来支持排错、测量以及安全等措施。在必要的时候插入值为0的填充字节。
数据部分	可变	用来填充报文。

2. 函数接口

a) **接收接口**

int stud_ip_recv(char * pBuffer, unsigned short length)

参数：

pBuffer：指向接收缓冲区的指针，指向 IPv4 分组头部

length：IPv4 分组长度

返回值：

0：成功接收 IP 分组并交给上层处理

1：IP 分组接收失败

b) **发送接口**

int stud_ip_Upsend(char* pBuffer, unsigned short len, unsigned int srcAddr, unsigned int dstAddr ,byte protocol, byte ttl)

参数：

pBuffer：指向发送缓冲区的指针，指向 IPv4 上层协议数据头部

len：IPv4 上层协议数据长度

srcAddr：源 IPv4 地址

dstAddr：目的 IPv4 地址

protocol：IPv4 上层协议号

ttl：生存时间（Time To Live）

返回值：

0：成功发送 IP 分组

1：发送 IP 分组失败

3. 系统函数

a) **丢弃分组**

void ip_DiscardPkt(char * pBuffer ,int type)

参数：

pBuffer：指向被丢弃分组的指针

type：分组被丢弃的原因，可取以下值：

STUD_IP_TEST_CHECKSUM_ERROR //IP 校验和出错

STUD_IP_TEST_TTL_ERROR //TTL 值出错

STUD_IP_TEST_VERSION_ERROR //IP 版本号错

STUD_IP_TEST_HEADLEN_ERROR //头部长度错

STUD_IP_TEST_DESTINATION_ERROR //目的地址错

b) **发送分组**

void ip_SendtoLower(char *pBuffer ,int length)

参数：

pBuffer：指向待发送的 IPv4 分组头部的指针

length：待发送的 IPv4 分组长度

c) **上层接收**

void ip_SendtoUp(char *pBuffer, int length)

参数：

pBuffer：指向要上交的上层协议报文头部的指针

length：上交报文长度

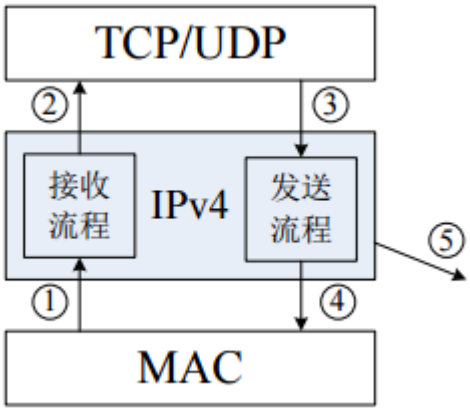
d) **获取本地 Ipv4 地址**

unsigned int getIpv4Address( )

参数：

无

3. 发送接收函数实现程序流程图



接收流程

① 检查接收到的 IPv4 分组头部的字段，包括版本号（Version）、头 部长度（IP Head length）、生存时间（Time to live）以及头校验 和（Header checksum）字段。对于出错的分组调用 ip_DiscardPkt( ) 丢弃，说明错误类型。

② 检查 IPv4 分组是否应该由本机接收。如果分组的目的地址是本 机地址或广播地址，则说明此分组是发送给本机的；否则调用 ip_DiscardPkt( )丢弃，并说明错误类型。

③ 如果 IPV4 分组应该由本机接收，则提取得到上层协议类型，调用 ip_SendtoUp( )接口函数，交给系统进行后续接收处理。

发送流程

① 根据所传参数（如数据大小），来确定分配的存储空间的大小并 申请分组的存储空间。

② 按照 IPv4 协议标准填写 IPv4 分组头部各字段，标识符 （Identification）字段可以使用一个随机数来填写。（注意：部分 字段内容需要转换成网络字节序）

③ 完成 IPv4 分组的封装后，调用 ip_SendtoLower( )接口函数完成 后续的发送处理工作，最终将分组发送到网络中

4. 数据结构

按照 IPv4 首页的顺序构造结构体，其中 char 是一个字节，即 8Bits，short 是两个字节，即 16 bit, unsigned int 是 4 个字节，即 32 Bits。

结构体的构造符合 IPv4 首部的情况

```
struct Ipv4
{
    char version_ihl;                   // 版本号
    char type_of_service;               .// 协议类型
    short total_length;                 // 总长度
    short identification;               // 标志符
    short fragment_offset;              // 偏移量
    char time_to_live;                  // TTL
    char protocol;                      // 协议
```

```
    short header_checksum;                  // 首部校验和
    unsigned int source_address;            // 源地址
    unsigned int destination_address;       // 目标地址
}
```

5. 错误检测原理

版本号校验

版本号在第一个字节的前 4 位里面，version_ihl 是结构体的第一个字节，则它的前 4 位代表版本号，右移 4 位，和 0xF 取和运算，如果结果依旧是 4，则代表版本号正确

```
int version = 0xf & ((ipv4->version_ihl)>> 4);
  if(version != 4)  {
     ip_DiscardPkt(pBuffer,STUD_IP_TEST_VERSION_ERROR);
     return 1;
  }
```

首部长度出错

首部长度在第一个字节的后 4 位里面，则 version_ihl 和 0xF 取和，如果结果为 5，则代表头部长度没有问题

```
  int ihl = 0xf & ipv4->version_ihl;
  if(ihl < 5) {
     ip_DiscardPkt(pBuffer,STUD_IP_TEST_HEADLEN_ERROR);
     return 1;
  }
```

TTL 出错

TTL 存在于第 9 个字节里面，按照数据结构的定义，存在于 time_to_live 里面，按照规定，如果 ttl 的值是 0，则代表生命周期结束，要抛弃这个包

```
   int ttl = (int)ipv4->time_to_live;
   if(ttl == 0) {
       ip_DiscardPkt(pBuffer,STUD_IP_TEST_TTL_ERROR);
       return 1;
   }
```

目标地址出错

目标地址存在于首部的第 17 -20 个字节中，取出他和本地 Ip 地址做比较，如果不等于本地地址，并且也不等于 0xffffffff，即广播地址，则表示目标地址出错。需要注意字节码序的转变

```
   int destination_address = ntohl(ipv4->destination_address);
   if(destination_address != getIpv4Address() && destination_address != 0xffffffff) {
      ip_DiscardPkt(pBuffer,STUD_IP_TEST_DESTINATION_ERROR);
      return 1;
   }
```

校验和出错

校验和存在于 11-12 个字节，校验和检测的规则如下： 16 位二进制反码求和，也就是说是将所有的字节加起来（校验和部分忽略，即为 0），然后把进位加到最后，然后按位取反，将得到的结构和 checksum 的值进行比较，如果不相等说明出错

```
int header_checksum = ntohs(ipv4->header_checksum);
```

```
int sum = 0;
for(int i = 0; i < ihl*2; i++) {
   if(i!=5)
   {
      sum += (int)((unsigned char)pBuffer[i*2] << 8);//校验和，扩展为 16 位，多余的进位放在 32 位的字节里
      sum += (int)((unsigned char)pBuffer[i*2+1]);
   }
}

while((sum & 0xffff0000) != 0) {//存在进位的情况
   sum = (sum & 0xffff) + ((sum >> 16) & 0xffff);//加上进位
}
unsigned short int ssum = (~sum) & 0xffff;//校验和错
if(ssum != header_checksum) {
   ip_DiscardPkt(pBuffer,STUD_IP_TEST_CHECKSUM_ERROR);
   return 1;
}  }
```

6. 源代码

```
/*
* THIS FILE IS FOR IP TEST
*/
// system support
#include "sysInclude.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

struct Ipv4
{
   char version_ihl;
   char type_of_service;
   short total_length;
   short identification;
   short fragment_offset;
   char time_to_live;
```

```
    char protocol;
    short header_checksum;
    unsigned int source_address;
    unsigned int destination_address;
    Ipv4() {
        memset(this,0,sizeof(Ipv4));//无参构造函数
    }
    Ipv4(unsigned int len,unsigned int srcAddr,unsigned int dstAddr,
        byte _protocol,byte ttl) {//有参构造函数，封装 IPV4 报文
        memset(this,0,sizeof(Ipv4));
        version_ihl = 0x45;
        total_length = htons(len+20);//20 字节头部
        time_to_live = ttl;
        protocol = _protocol;
        source_address = htonl(srcAddr);//转为网络字节序
        destination_address = htonl(dstAddr);

        char *pBuffer;
        memcpy(pBuffer,this,sizeof(Ipv4));
        int sum = 0;
        for(int i = 0; i < 10; i++) {//cheksum，求校验和
            if(i != 5) {
                sum += (int)((unsigned char)pBuffer[i*2] << 8);
                sum += (int)((unsigned char)pBuffer[i*2+1]);
            }
        }
        while((sum & 0xffff0000) != 0) {
            sum = (sum & 0xffff) + ((sum >> 16) & 0xffff);
        }
        unsigned short int ssum = sum;
        header_checksum = htons(~ssum);
    }
};

int stud_ip_recv(char *pBuffer,unsigned short length)
{
    Ipv4 *ipv4 = new Ipv4();
    *ipv4 = *(Ipv4*)pBuffer;
    int version = 0xf & ((ipv4->version_ihl)>> 4);//版本号校验
    if(version != 4)  {
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_VERSION_ERROR);
        return 1;
    }
    int ihl = 0xf & ipv4->version_ihl;//首部长度校验
```

```
    if(ihl < 5) {
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_HEADLEN_ERROR);
        return 1;
    }
    int ttl = (int)ipv4->time_to_live;//生存时间校验
    if(ttl == 0) {
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_TTL_ERROR);
        return 1;
    }
    int destination_address = ntohl(ipv4->destination_address);
    if(destination_address != getIpv4Address() && destination_address != 0xffffffff) {
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_DESTINATION_ERROR);//目的地址不是本机或者
广播地址
        return 1;
    }
    int header_checksum = ntohs(ipv4->header_checksum);
    int sum = 0;
    for(int i = 0; i < ihl*2; i++) {
        if(i!=5)
        {
            sum += (int)((unsigned char)pBuffer[i*2] << 8);//校验和，扩展为 16 位，多余的进
位放在 32 位的字节里
            sum += (int)((unsigned char)pBuffer[i*2+1]);
        }
    }

    while((sum & 0xffff0000) != 0) {//存在进位的情况
        sum = (sum & 0xffff) + ((sum >> 16) & 0xffff);//加上进位
    }
    unsigned short int ssum = (~sum) & 0xffff;//校验和错
    if(ssum != header_checksum) {
        ip_DiscardPkt(pBuffer,STUD_IP_TEST_CHECKSUM_ERROR);
        return 1;
    }
    ip_SendtoUp(pBuffer,length);
    return 0;
}

int stud_ip_Upsend(char *pBuffer,unsigned short len,unsigned int srcAddr,
                   unsigned int dstAddr,byte protocol,byte ttl)
{
    char *pack_to_sent = new char[len+20];
    memset(pack_to_sent,0,len+20);
    *((Ipv4*)pack_to_sent) = Ipv4(len,srcAddr,dstAddr,protocol,ttl);
```

```
    memcpy(pack_to_sent+20,pBuffer,len);
    ip_SendtoLower(pack_to_sent,len+20);
    delete[] pack_to_sent;

    return 0;
}
```

1. 实验目的

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。

2. 实验内容

在前面 IPv4 分组收发实验的基础上，增加分组转发功能。具体来说，对于每一个到达本机的 IPv4 分组，根据其目的 IPv4 地址决定分组的处理行为，对该分组进行如下的几类操作：

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。

3. 实验要点

1) 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。

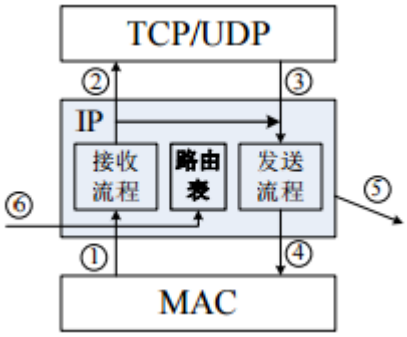
2) IPv4 分组的接收和发送。

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”。

3) 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

4. 实验结果



路由表结构体：

```
<map><unsigned int DEST,int unsigned int next>
```

路由表的实现使用的是 STL 库函数 MAP 映射，计算出能匹配最长前缀的子网地址=IP 地址&子网掩码，每一个子网地址对应的是下一跳地址。

路由表初始化：

```
void stud_Route_Init()    //初始化路由表
        {
                mapTable.clear();      //清空 map
                 return;
        }
```

路由表增加函数：

```
void stud_route_add(stud_route_msg *proute)  //路由表中添加项
{
          int DestinationAddress,nextHop;
          DestinationAddress=(ntohl(proute->dest))&(0xffffffff<<(32-
(proute->masklen)));
         =ntohl(proute->nexthop);
          mapTable[DestinationAddress]=nextHop;    //插入
 }
```

通过子网掩码&IP 地址=子网地址，可以得到路由聚合的最长匹配子网地址，把这个信息映射到下一跳，存在映射表里

处理收到的分组：

```
int stud_fwd_deal(char *pBuffer, int length)  //处理收到的分组
{
 int IHL=pBuffer[0]&0xf;
 int TTL=(int)pBuffer[8];
 int Head_Checksum=ntohs(*(unsigned short*)(pBuffer+10));
 int Dst_IP=ntohl(*(unsigned*)(pBuffer+16));
 if(TTL<=0) {
  fwd_DiscardPkt(pBuffer,STUD_FORWARD_TEST_TTLERROR);
  return 1;
 }
 if(Dst_IP==getIpv4Address()) {
  fwd_LocalRcv(pBuffer,length);  //上交分组
  return 0;
 }
 map<int,int>::iterator iter;
 iter = mapTable.find(Dst_IP);
 if(iter!=mapTable.end()){
  char *buffer=new char[length];
   memcpy(buffer,pBuffer,length);
   buffer[8]--;  //ttl 减一
   int sum=0,i;
   unsigned short LocalChecksum=0;
   for(i=0;i<2*IHL;i++) {
```

```
    if(i!=5){
     sum+=(buffer[2*i]<<8)|(buffer[2*i+1]);
     sum%=65535;
    }
   if(i+1 == 2*IHL){
   LocalChecksum=htons(~(unsigned short)sum); //重新计算校验和
   memcpy(buffer+10,&LocalChecksum,2);
   }
   }
   fwd_SendtoLower(buffer,length,iter->second); //调用下层进行发送处理
   return 0;
 }
 fwd_DiscardPkt(pBuffer,STUD_FORWARD_TEST_NOROUTE);
 return 1;
}
```

对于收到的 IPV4 数据报，要判断生存时间是否大于零，不然的话要丢弃分组，无法再进行转发。然后判断目地地址是否是本机 IP，如果是就接受分组，并转发给上层协议进行处理。对于其他情况，首先计算出校验和，并保存在分组的 checksum 字段，并且查找路由表，得到下一跳的信息，并把分组转发给下一跳的 IP。其他情况下，说明分组有错或者查不到路由表，则丢弃分组

5. 源代码

```
#include "sysInclude.h"
#include<map>
using std::map;
// 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
map<int,int>mapTable;         //路由表存储  第一个参数 目的地址  第二个参数 下
一跳
void stud_Route_Init()   //初始化路由表
{
 mapTable.clear();       //清空 map
 return;
}
void stud_route_add(stud_route_msg *proute)  //路由表中添加项
{
 int DestinationAddress,nextHop;
 DestinationAddress=(ntohl(proute->dest))&(0xffffffff<<(32-
htonl(proute->masklen)));
```

```
 nextHop=ntohl(proute->nexthop);
 mapTable[DestinationAddress]=nextHop;    //插入
}
int stud_fwd_deal(char *pBuffer, int length)  //处理收到的分组
{
 int IHL=pBuffer[0]&0xf;
 int TTL=(int)pBuffer[8];
 int Head_Checksum=ntohs(*(unsigned short*)(pBuffer+10));
 int Dst_IP=ntohl(*(unsigned*)(pBuffer+16));
 if(TTL<=0) {
  fwd_DiscardPkt(pBuffer,STUD_FORWARD_TEST_TTLERROR);
  return 1;
 }
 if(Dst_IP==getIpv4Address()) {
  fwd_LocalRcv(pBuffer,length);  //上交分组
  return 0;
 }
 map<int,int>::iterator iter;
 iter = mapTable.find(Dst_IP);
 if(iter!=mapTable.end()){
  char *buffer=new char[length];
   memcpy(buffer,pBuffer,length);
   buffer[8]--;  //ttl 减一
   int sum=0,i;
   unsigned short LocalChecksum=0;
   for(i=0;i<2*IHL;i++) {
    if(i!=5){
     sum+=(buffer[2*i]<<8)|(buffer[2*i+1]);
     sum%=65535;
    }
   if(i+1 == 2*IHL){
   LocalChecksum=htons(~(unsigned short)sum); //重新计算校验和
   memcpy(buffer+10,&LocalChecksum,2);
   }
   }
   fwd_SendtoLower(buffer,length,iter->second); //调用下层进行发送处理
   return 0;
 }
 fwd_DiscardPkt(pBuffer,STUD_FORWARD_TEST_NOROUTE);
 return 1;
}
```