

**计算机网络**

**课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | IPv4分组收发实验，  Ipv4分组转发实验 | | | | | |
| 姓名 | 余涛 | | 院系 | 计算机科学与技术学院 | | |
| 班级 | 1803202 | | 学号 | 1180300829 | | |
| 任课教师 | 刘亚维 | | 指导教师 | 刘亚维 | | |
| 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 | | | | | | |
|  | | | | | | |

**计算学部**

|  |
| --- |
| 实验目的： |
| 本次实验的主要目的。  **实验四**：  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。  **实验五**：  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： |
| 概述本次实验的主要内容，包含的实验项等。  **实验四：**  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。  **实验五：**  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。 |
| 实验过程： |
| 以文字描述、实验结果截图等形式阐述实验过程，必要时可附相应的代码截图或以附件形式提交。  **一：IPv4分组收发实验**  **(1) 发送和接收函数的实现程序流程图**  （1.1）下图为发送函数stud\_ip\_Upsend()的流程图：    （1.2）下图为接收函数stud\_ip\_recv()的流程图：    **(2) 自己所新建的数据结构的说明**  没有新建数据结构，所以此项不进行说明。  **(3) 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）字段的错误**  **检测原理并根据实验具体情况给出错误的具体数据**  下图为Ipv4分组的数据包头部信息：    （3.1）版本号（Version）字段的错误检测原理：  直接提取Ipv4数据包第0个字节内的最开始4个bits，然后将这4个bits储存在version变量内，  然后用version与4判断是否相等即可，若不相等，则调用ip\_DiscardPkt()函数并输入错误类型STUD\_IP\_TEST\_VERSION\_ERROR。  if (version != 4)  {  //如果出现版本号不为4的错误，调用ip\_DiscardPkt并报道错误类型  ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_VERSION\_ERROR);  return 1;  }  （3.2）头部长度（IP Head Length）字段的错误检测原理：  直接提取Ipv4数据包第0个字节内的后面的4个bits，然后将这4个bits储存在head\_length变量内，然后用head\_length与5判断即可(由于头部字段以4字节为单位且头部字段最小为20字节)，若小于5，则调用ip\_DiscardPkt()函数并输入错误类型STUD\_IP\_TEST\_HEADLEN\_ERROR。  if (head\_length < 5)  {  //如果出现头部长度小于20个字节的错误(由于头部长度字段以4字节为单位，故只需与5比较即可)，调用ip\_DiscardPkt并报道错误类型  ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_HEADLEN\_ERROR);  return 1;  }  （3.3）生存时间（Time to live）字段的错误检测原理：  直接提取Ipv4数据包第8个字节内的8个bits，然后将这8个bits储存在ttl变量内，然后用ttl与0判断即可，若小于等于0，则调用ip\_DiscardPkt()函数并输入错误类型STUD\_IP\_TEST\_TTL\_ERROR。  if (ttl <= 0)  {  //如果出现TTL<=0的错误，调用ip\_DiscardPkt并报道错误类型  ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_TTL\_ERROR);  return 1;  }  （3.4）头部校验和（Header Checksum）字段的错误检测原理：  直接提取Ipv4数据包第10个字节内的16个bits，然后将这16个bits储存在checksum变量内。对于校验和的错误，需要首先计算校验和，首先对IPv4的数据包包头以两字节为单位，两两相加，计算相加后的和sum，然后取出sum的低16位与高16位相加与0xffff比较。若不为0xffff，  则调用ip\_DiscardPkt()函数并输入错误类型STUD\_IP\_TEST\_CHECKSUM\_ERROR。  //对于校验和的错误，需要首先计算校验和  unsigned long sum = 0;  unsigned long temp = 0;  int i;  //首先对IPv4的数据包包头以两字节为单位，两两相加，计算相加后的和sum  for (i = 0; i < head\_length \* 2; i++)  {  temp += (unsigned char)pBuffer[i \* 2] << 8;  temp += (unsigned char)pBuffer[i \* 2 + 1];  sum += temp;  temp = 0;  }  unsigned short low\_of\_sum = sum & 0xffff; //取出低16位  unsigned short high\_of\_sum = sum >> 16; //取出高16位  if (low\_of\_sum + high\_of\_sum != 0xffff) //低16位与高16位相加  {  //如果出现首部校验和的错误(计算结果不为0xffff)，调用ip\_DiscardPkt并报道错误类型  ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_CHECKSUM\_ERROR);  return 1;  }  （3.5）目的地址（destination ip）字段的错误检测原理：  直接提取Ipv4数据包第16个字节内的32个bits，然后将这32个bits储存在destination变量内。然后用destination与地址判断即可。若不为目的地址或者为广播地址，则调用ip\_DiscardPkt()函数并输入错误类型STUD\_IP\_TEST\_DESTINATION\_ERROR。  if (destination != getIpv4Address() && destination != 0xffff)  {  //如果出现错误目的地址的错误，调用ip\_DiscardPkt并报道错误类型  ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_DESTINATION\_ERROR);  return 1;  }  **(4)实验代码（含详细注释）**  见附录  **二：IPv4分组转发实验**  **(1) 路由表初始化、路由增加、路由转发的实现程序流程图**  （1.1）下图为路由表初始化函数stud\_Route\_Init ()的流程图：  由于使用了vector全局变量作为存储路由表的结构，其在创建的时候便已经自动完成了初始化，因此在stud\_Route\_Init ()不需要执行任何信息，因此该函数的流程图便没有画的必要。  路由表结构如下：  vector<stud\_route\_msg> route; //设置遍历器结构作为路由表  （1.2）下图为路由增加函数stud\_route\_add ()的流程图：    （1.3）下图为路由转发函数stud\_fwd\_deal()的流程图：    **(2) 自己所新建的数据结构的说明**  为了实现路由表的功能，定义了一个vector类型的全局变量route来作为路由表，里面储存了所有的路由表项stud\_route\_msg。对于需要实现功能中的初始化路由表，在创建route变量的时候就已经完成。对于需要实现功能中增加路由表项，只需要在route的尾部增加新的路由表项即可。在route中进行遍历，只能从头部进行线性的遍历搜索。  vector<stud\_route\_msg> route; //设置遍历器结构作为路由表  **(3) 提高转发效率的原理**  通过提高遍历路由表的速度(比如对路由表进行有序存储)可以提高转发效率，这样能够更快找到符合要求的路由表项进行转发。  **(4)实验代码（含详细注释）**  见附录 |
| 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。  一.Ipv4分组收发实验：    二.Ipv4分组转发实验：    登录到实验平台查看分数如下： |
| 问题讨论： |
| 对实验过程中的思考问题进行讨论或回答。  1.为了提高转发效率，可以从路由表的存储顺序出发考虑，通过按照IP目的地址从小到大来为路由表中的路由表项进行排序，查找vector时可以通过二分查找来提高效率。  2. 为了提高转发效率，也可以通过在建立路由表时建立查找效率高的数据结构来实现，这样查询的时间将大大缩短，提高转发效率。 |
| 心得体会： |
| 结合实验过程和结果给出实验的体会和收获。  通过此次实验我了解并掌握了Ipv4数据包的结构和每一个字节所储存的内容，掌握了路由表的作用，明白了Ipv4数据包的收发和分组转发的具体过程，掌握了检验和的求法，对路由有了更深的理解。 |

**附录：详细注释源程序：**

Ipv4分组收发实验：

/\*

IPV4 分组收发实验

\*/

#include "sysInclude.h"

#include <stdio.h>

#include <malloc.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();

/\*

接收接口函数

输入：pBuffer为指向接收缓冲区的指针，指向IPv4 分组头部

length为IPv4为分组长度

返回：0：成功接收IP 分组并交给上层处理

1：IP 分组接收失败

\*/

int stud\_ip\_recv(char\* pBuffer, unsigned short length)

{

int version = pBuffer[0] >> 4; // pBuffer第0个字节内的最开始4个bits为版本号

int head\_length = pBuffer[0] & 0xf; // pBuffer第0个字节内的紧接着4个bits为头部长度

short ttl = (unsigned short)pBuffer[8]; // pBuffer第8个字节内的8个bits为生存时间ttl

short checksum = ntohs(\*(unsigned short\*)(pBuffer + 10)); // pBuffer第10个字节后的16bits为头检验和（short int）

int destination = ntohl(\*(unsigned int\*)(pBuffer + 16)); // pBuffer第16个字节后的32bits为目的ip地址（long int）

if (ttl <= 0)

{

//如果出现TTL<=0的错误，调用ip\_DiscardPkt并报道错误类型

ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_TTL\_ERROR);

return 1;

}

if (version != 4)

{

//如果出现版本号不为4的错误，调用ip\_DiscardPkt并报道错误类型

ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_VERSION\_ERROR);

return 1;

}

if (head\_length < 5)

{

//如果出现头部长度小于20个字节的错误(由于头部长度字段以4字节为单位，故只需与5比较即可)，调用ip\_DiscardPkt并报道错误类型

ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_HEADLEN\_ERROR);

return 1;

}

if (destination != getIpv4Address() && destination != 0xffff)

{

//如果出现错误目的地址的错误，调用ip\_DiscardPkt并报道错误类型

ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_DESTINATION\_ERROR);

return 1;

}

//对于校验和的错误，需要首先计算校验和

unsigned long sum = 0;

unsigned long temp = 0;

int i;

//首先对IPv4的数据包包头以两字节为单位，两两相加，计算相加后的和sum

for (i = 0; i < head\_length \* 2; i++)

{

temp += (unsigned char)pBuffer[i \* 2] << 8;

temp += (unsigned char)pBuffer[i \* 2 + 1];

sum += temp;

temp = 0;

}

unsigned short low\_of\_sum = sum & 0xffff; //取出低16位

unsigned short high\_of\_sum = sum >> 16; //取出高16位

if (low\_of\_sum + high\_of\_sum != 0xffff) //低16位与高16位相加

{

//如果出现首部校验和的错误(计算结果不为0xffff)，调用ip\_DiscardPkt并报道错误类型

ip\_DiscardPkt(pBuffer, STUD\_IP\_TEST\_CHECKSUM\_ERROR);

return 1;

}

ip\_SendtoUp(pBuffer, length); //提交给上层协议

return 0;

}

/\*

发送接口函数

输入：pBuffer为指向接收缓冲区的指针，指向IPv4 分组头部

len为IPv4上层协议数据长度

srcAddr为源IPv4地址

dstAddr为目的IPv4地址

protocol为IPv4上层协议号

ttl为生存时间

返回：0：成功发送IP分组

1：发送IP分组失败

\*/

int stud\_ip\_Upsend(char\* pBuffer, unsigned short len, unsigned int srcAddr,

unsigned int dstAddr, byte protocol, byte ttl)

{

//一般默认头部长度为字节

short ip\_length = len + 20; //得到这层的数据长度

char\* buffer = (char\*)malloc(ip\_length \* sizeof(char));

memset(buffer, 0, ip\_length);

buffer[0] = 0x45; //规定版本号和首部长度为4和5×4=20

buffer[8] = ttl; //规定生存时间ttl

buffer[9] = protocol; //规定协议号

// 将数据长度转换为网络字节序

unsigned short network\_length = htons(ip\_length);

// buffer[2] = network\_length >> 8;

// buffer[3] = network\_length & 0xff;

memcpy(buffer + 2, &network\_length, 2);

unsigned int src = htonl(srcAddr); //解析源IPv4地址

unsigned int dst = htonl(dstAddr); //解析目的IPv4地址

memcpy(buffer + 12, &src, 4);

memcpy(buffer + 16, &dst, 4);

//计算校验和

unsigned long sum = 0;

unsigned long temp = 0;

int i;

//首先对IPv4的数据包包头以两字节为单位，两两相加，计算相加后的和sum

for (i = 0; i < 20; i += 2)

{

temp += (unsigned char)buffer[i] << 8;

temp += (unsigned char)buffer[i + 1];

sum += temp;

temp = 0;

}

unsigned short low\_of\_sum = sum & 0xffff; //取出低16位

unsigned short high\_of\_sum = sum >> 16; //取出高16位

unsigned short checksum = low\_of\_sum + high\_of\_sum; //低16位与高16位相加得到校验和(未取反)

checksum = ~checksum; //取反得到校验和

unsigned short header\_checksum = htons(checksum); //将校验和更新

// buffer[10] = header\_checksum >> 8;

// buffer[11] = header\_checksum & 0xff;

memcpy(buffer + 10, &header\_checksum, 2);

memcpy(buffer + 20, pBuffer, len);

// ip\_SendtoLower(buffer, ip\_length);

ip\_SendtoLower(buffer, len + 20); //发送分组

return 0;

}

Ipv4分组转发实验:

/\*

IPV4 分组转发实验部分

\*/

#include "sysInclude.h"

#include <stdio.h>

#include <vector>

using std::vector;

// 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

vector<stud\_route\_msg> route; //设置遍历器结构作为路由表

/\*

路由表初始函数

\*/

void stud\_Route\_Init()

{

//在创建全局变量route时已经进行了初始化，无需再初始化

return;

}

/\*

用路由表添加路由的函数

输入：proute ：指向需要添加路由信息的结构体头部，

其数据结构

stud\_route\_msg 的定义如下：

typedef struct stud\_route\_msg

{

unsigned int dest; //目的地址

unsigned int masklen; //子网掩码长度

unsigned int nexthop; //下一跳

} stud\_route\_msg;

\*/

void stud\_route\_add(stud\_route\_msg\* proute)

{

stud\_route\_msg temp; //定义一个temp变量，用来将待添加的路由表项转化为本地字节序

unsigned int dest = ntohl(proute->dest);

unsigned int masklen = ntohl(proute->masklen);

unsigned int nexthop = ntohl(proute->nexthop); //依次将proute的所有字段转化为字节序储存

temp.dest = dest;

temp.masklen = masklen;

temp.nexthop = nexthop; //为temp赋值

route.push\_back(temp); //将temp加入到路由表中

return;

}

/\*

系统处理收到的IP分组的函数

输入：pBuffer：指向接收到的IPv4 分组头部

length：IPv4 分组的长度

返回：0 为成功，1 为失败；

\*/

int stud\_fwd\_deal(char\* pBuffer, int length)

{

int version = pBuffer[0] >> 4; // pBuffer第0个字节内的最开始4个bits为版本号

int head\_length = pBuffer[0] & 0xf; // pBuffer第0个字节内的紧接着4个bits为头部长度

short ttl = (unsigned short)pBuffer[8]; // pBuffer第8个字节内的8个bits为生存时间ttl

short checksum = ntohs(\*(unsigned short\*)(pBuffer + 10)); // pBuffer第10个字节后的16bits为头检验和（short int）

int destination = ntohl(\*(unsigned int\*)(pBuffer + 16)); // pBuffer第10个字节后的16bits为头检验和（short int）

// ttl -= 1;

if (ttl <= 0)

{

//如果出现TTL的错误，调用ip\_DiscardPkt并报道错误类型

fwd\_DiscardPkt(pBuffer, STUD\_FORWARD\_TEST\_TTLERROR);

return 1;

}

if (destination == getIpv4Address())

{

//如果出现目的地址为本机地址，则调用fwd\_LocalRcv提交给上层协议处理

fwd\_LocalRcv(pBuffer, length);

return 0;

}

stud\_route\_msg\* ans\_route = NULL; //定义匹配位置

int temp\_dest = destination; //temp\_dest为目的地址

for (int i = 0; i < route.size(); i++) //遍历路由表

{

unsigned int temp\_sub\_net = route[i].dest & ((1 << 31) >> (route[i].masklen - 1)); //对于路由表中的每一个表项

if (temp\_sub\_net == temp\_dest) //如果目的地址与路由表的某一个表项匹配

{

ans\_route = &route[i]; //记录匹配的位置

break;

}

}

if (!ans\_route)

{

//如果出现没有匹配地址，调用ip\_DiscardPkt并报道错误类型，直接返回1

fwd\_DiscardPkt(pBuffer, STUD\_FORWARD\_TEST\_NOROUTE);

return 1;

}

else //对于匹配成功的情况

{

char\* buffer = new char[length]; //定义一个buffer储存ip分组pBuffer

memcpy(buffer, pBuffer, length);

buffer[8] = ttl - 1; //将ttl减1

memset(buffer + 10, 0, 2);

unsigned long sum = 0;

unsigned long temp = 0;

int i;

//重新计算校验和

//首先对IPv4的数据包包头以两字节为单位，两两相加，计算相加后的和sum

for (i = 0; i < head\_length \* 2; i++)

{

temp += (unsigned char)buffer[i \* 2] << 8;

temp += (unsigned char)buffer[i \* 2 + 1];

sum += temp;

temp = 0;

}

unsigned short low\_of\_sum = sum & 0xffff; //取出低16位

unsigned short high\_of\_sum = sum >> 16; //取出高16位

unsigned short checksum = low\_of\_sum + high\_of\_sum; //低16位与高16位相加得到校验和(未取反)

checksum = ~checksum; //取反得到校验和

unsigned short header\_checksum = htons(checksum);

memcpy(buffer + 10, &header\_checksum, 2); //更新校验和

fwd\_SendtoLower(buffer, length, ans\_route->nexthop); //将封装完成的IP分组通过链路层发送出去

}

return 0;

}