《计算机网络协议开发》实验报告

第七/八次实验

IPv6 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

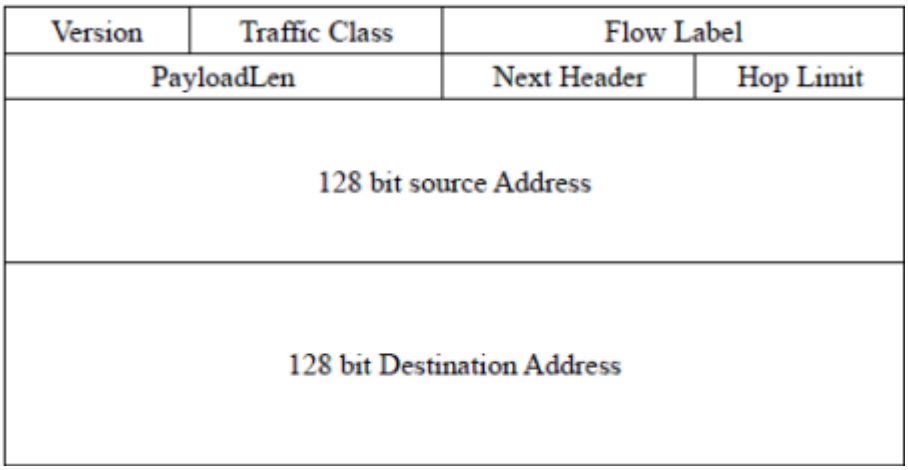
时间：2013/04/20

一、 实验目的

通过实验来了解 IPv6 协议的分组接收和发送流程，进一步理解网络层协议工作原理，并初步掌握网络层协议开发方法；并通过实现 IPv6 分组转发实验，来实现路由器中的 IPv6 协议模块，学习了解路由器是如何为 IPv6 分组选择路由，并逐跳的将 IPv6 分组转发到目的主机。

二、实验设计要点

-1- IPv6 分组的头部格式



Version： 4 位的协议版本号； 为 6

Traffic Class: 流量类型， 占 8 位

Flow Label :流标签， 占 20 位

Payload Length: 16 位无符号整数，IPv6 载荷，即 IPv6 头部后面部分的长度，以 8 bits 为单位。

Next Header：8 位，标识紧跟 IPv6 头部之后的头部类型

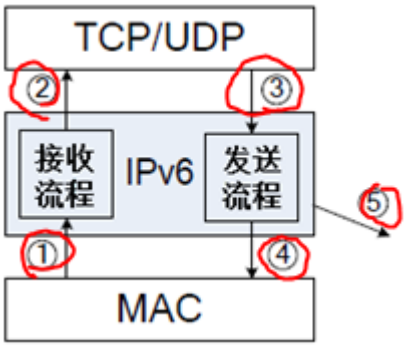
Hop Limit : 8 位无符号整数，每经过一个节点后-1，减为 0 后将该分组丢弃

Source Address : 源 IPv6 地址

Destination Adress: 目的 IPv6 地址

-2- 收发流程与转发流程同 IPv4 类似（略）

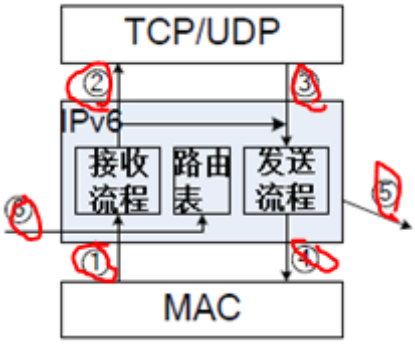
IPv6 收发设计实现：



客户端接收到测试服务器发送来的 IPv6 分组后，调用接收接口函数①，我们需要在这个函数中实现 IPv6 分组接受处理功能，接受处理完成之后，调用接口函数②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⑤丢弃有错误的分组，并报告错误类型。

在上层协议需要发送分组是，会调用函数③。我们需要在这个函数中实现 IPv6 分组封装发送功能。根据所传的参数完成 IPv6 分组的封装并调用接口函数④把分组交给下层来完成发送。

IPv6 转发设计实现：



实验的主要流程和系统接口函数与前面的 IPv6 分组收发实验基本相同。在下层接口函数①中实现分组的接收处理，其主要功能是根据分组中的目的 IPv6 地址查找路由表，对分组进行处理，如果分组需要上交，需要调用接口函数②，如果分组需要丢弃，则调用函数⑤，如果分组需要转发，则进行转发操作，转发操作的实现要点是：首先 HoplImit-1，然后调用发送接口函数④将分组发送出去。

同时有路由表配置的接口函数⑥，要求能够根据系统提供的信息设定本机路由表，试验中只需要简单地设置静态路由信息，作为分组接收和转发处理的判断依据。

三、实验内容

IPv6 头部结构设计：

```
//extern long Subnet(const ipv6_addr* addr, unsigned int masklen);
typedef struct IPv6{
  byte Version;
  unsigned char TrafficClass;
  unsigned int FlowLabel;
  unsigned short PayloadLen;
  unsigned char NextHeader;
  unsigned char HopLimit;
  ipv6_addr S_addr;
  ipv6_addr D_addr;
}Header_v6;
```

路由表实现的链表元素设计：

```
typedef struct Route_node{
    stud_ipv6_route_msg msg;
    Route_node* next;
}Route;

Route *route_first = NULL;

void stud_ipv6_Route_Init()
{
    while(route_first != NULL){
        Route *p = route_first->next;
        delete(route_first);
        route_first = p;
    }
    return;
}
```

判断 IPv6 地址是否相等：

```
bool is_equal_addr(ipv6_addr* t1, ipv6_addr* t2){
    int k=0;
    for(int i=0; i<16; i++){
        if(t1->bAddr[i] == t2->bAddr[i]) k++;
        else break;
    }
    if(k==16)
         return true;
    else
         return false;
}
```

判断目的地址是否和路由表项在一个子网编号中：

```
bool is_equal_subnet(int masklen, ipv6_addr t1, ipv6_addr t2){
    int num = masklen/8;
    int remain = masklen%8;
    for(int i=0; i<num; i++)
        if(t1.bAddr[i] != t2.bAddr[i])  return false;
    char a = t1.bAddr[num] & (0xff << (8-remain));
    char b = t2.bAddr[num] & (0xff << (8-remain));
    if(a == b)
        return true;
    else
        return false;
}
```

【1】 收发实验的关键代码：

```
 Header_v6 head;
 head.Version = (pBuffer[0]>>4) & 0x0f;
 //head.TrafficClass
 //head.FlowLabel
 head.PayloadLen = ntohs(*((unsigned short *)(pBuffer+4)));
 head.NextHeader = pBuffer[6];
 head.HopLimit = pBuffer[7];

 for(int i=0; i<16; i++)   head.S_addr.bAddr[i] = pBuffer[8+i];
 for(int i=0; i<16; i++)   head.D_addr.bAddr[i] = pBuffer[24+i];


if(head.Version != 6){
    ipv6_DiscardPkt(pBuffer,  STUD_IPV6_TEST_VERSION_ERROR);
    return 1;
}

if (head.HopLimit <= 0)
{
    ipv6_DiscardPkt(pBuffer, STUD_IPV6_TEST_HOPLIMIT_ERROR);
    return 1;
}

ipv6_addr* localhost = new ipv6_addr;
getIpv6Address(localhost);
if (Equal(localhost, &head.D_addr)==false)
{
    ipv6_DiscardPkt(pBuffer, STUD_IPV6_TEST_DESTINATION_ERROR);
    return 1;
}

ipv6_SendtoUp(pBuffer, length);



int stud_ipv6_Upsend(char *pData, unsigned short len,
                     ipv6_addr *srcAddr, ipv6_addr *dstAddr,
                     char hoplimit, char nexthead)
{
```

```
{
    char *pBuffer = new char[40+len];
    memset(pBuffer, 0, 40+len);
    pBuffer[0] = 0x60;
    *(unsigned short *)(pBuffer+3) = htons(len);
    pBuffer[6] = nexthead;
    pBuffer[7] = hoplimit;
    //
    for(int i=0; i<16; i++)   pBuffer[8+i] = srcAddr->bAddr[i];
    for(int i=0; i<16; i++)   pBuffer[24+i]  = dstAddr->bAddr[i];

    memcpy(pBuffer+40, pData, len);

      ipv6_SendtoLower(pBuffer, len+40);
    return 0;
}
```

【2】 转发实验的关键代码：

```
int stud_ipv6_fwd_deal(char *pBuffer, int length)
{
    Header_v6 head;
    head.Version = (pBuffer[0]>>4) & 0x0f;
    //head.TrafficClass
    //head.FlowLabel
    head.PayloadLen = ntohs(*((unsigned short *)(pBuffer+4)));
    head.NextHeader = pBuffer[6];
    head.HopLimit = pBuffer[7];
    //
    for(int i=0; i<16; i++)   head.S_addr.bAddr[i] = pBuffer[8+i];
    for(int i=0; i<16; i++)   head.D_addr.bAddr[i] = pBuffer[24+i];

    ipv6_addr* localhost = new ipv6_addr;
    getIpv6Address(localhost);

    if(is_equal_addr(localhost, &head.D_addr)==true){
        ipv6_fwd_LocalRcv(pBuffer, length);
        return 0;
    }
```

中间出错处理部分略；

```
if(is_equal_subnet(p->msg.masklen, head.D_addr, *localhost)==true){
    ipv6_fwd_SendtoLower(pBuffer, length, &head.D_addr);
}
else{
    ipv6_fwd_SendtoLower(pBuffer, length, &p->msg.nexthop);
}
```

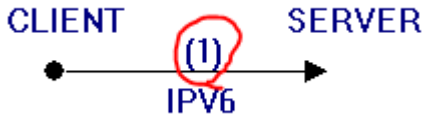
判断是否是最后一跳，进行不同的转发处理。

四、实验结果

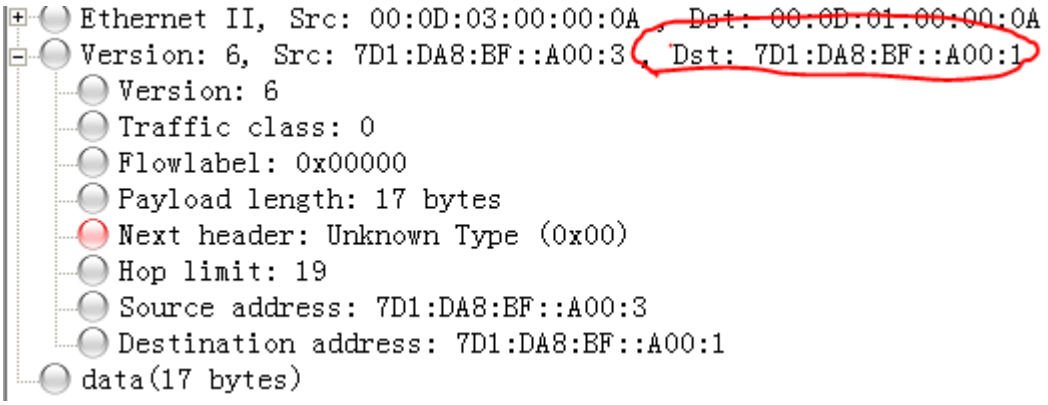
IPv6 分组的基本接收功能：

分析：

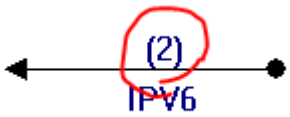
【1】正确发送 IPv6 数据包



编号	时间	源地址	目的地址	协议	数据包描述	实验描述
1	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:3	7D1:DA8:BF::A00:1	IPV6	Version ...	4.1 发送IPv6包
2	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.2 正确接收IPv6包
3	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.3 Hop Limit错的IPv6包
4	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.4 版本号错的IPv6包
5	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::C0A...	IPV6	Version ...	4.5 错误目标地址的IPv6包



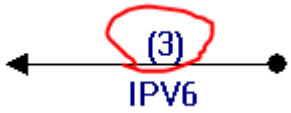
【2】正确接收 IPv6 数据包



编号	时间	源地址	目的地址	协议	数据包描述	实验描述
1	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:3	7D1:DA8:BF::A00:1	IPV6	Version ...	4.1 发送IPv6包
2	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.2 正确接收IPv6包
3	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.3 Hop Limit错的IPv6包
4	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.4 版本号错的IPv6包
5	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::C0A...	IPV6	Version ...	4.5 错误目标地址的IPv6包

```
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version: 6, Src: 7D1:DA8:BF::A00:1 , Dst: 7D1:DA8:BF::A00:3
    Version: 6
    Traffic class: 0
    Flowlabel: 0x00000
    Payload length: 0 bytes
    Next header: Unknown Type (0x00)
    Hop limit: 16
    Source address: 7D1:DA8:BF::A00:1
    Destination address: 7D1:DA8:BF::A00:3
```

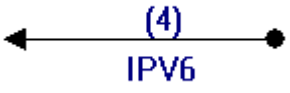
【3】 HopLimit 出错



号	时间	源地址	目的地址	协议	数据包描述	实验描述
1	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:3	7D1:DA8:BF::A00:1	IPV6	Version ...	4.1 发送IPv6包
2	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.2 正确接收IPv6包
3	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.3 Hop Limit错的IPv6包
4	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.4 版本号错的IPv6包
5	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::C0A...	IPV6	Version ...	4.5 错误目标地址的IPv6包

```
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version: 6, Src: 7D1:DA8:BF::A00:1 , Dst: 7D1:DA8:BF::A00:3
    Version: 6
    Traffic class: 0
    Flowlabel: 0x00000
    Payload length: 0 bytes
    Next header: Unknown Type (0x00)
    Hop limit: 0
    Source address: 7D1:DA8:BF::A00:1
    Destination address: 7D1:DA8:BF::A00:3
```

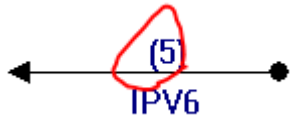
【4】 版本号出错



4	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	4.4 版本号错的IPv6包

```
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version: 5, Src: 7D1:DA8:BF::A00:1 , Dst: 7D1:DA8:BF::A00:3
    Version: 5
    Traffic class: 0
    Flowlabel: 0x00000
    Payload length: 0 bytes
    Next header: Unknown Type (0x00)
    Hop limit: 16
    Source address: 7D1:DA8:BF::A00:1
    Destination address: 7D1:DA8:BF::A00:3
```

【5】 错误的目的地址的数据包



5	Sat Apr 20 19:36:5...	7D1:DA8:BF::A00:1	7D1:DA8:BF::C0A...	IPV6	Version ...	4.5 错误目标地址的IPv6包

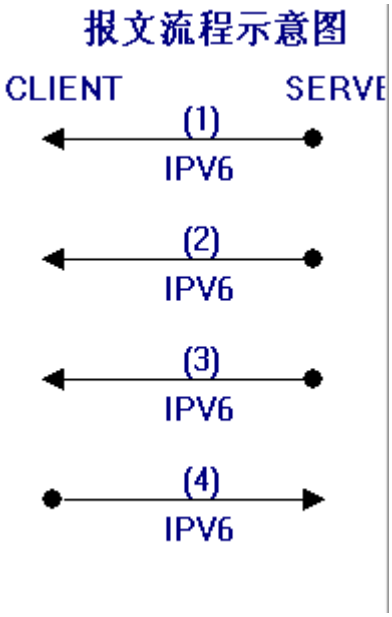
```
Ethernet II, Src: 00:0D:01:00:00:0A , Dst: 00:0D:03:00:00:0A
Version: 6, Src: 7D1:DA8:BF::A00:1 , Dst: 7D1:DA8:BF::C0A8:ED0A
    Version: 6
    Traffic class: 0
    Flowlabel: 0x00000
    Payload length: 0 bytes
    Next header: Unknown Type (0x00)
    Hop limit: 16
    Source address: 7D1:DA8:BF::A00:1
    Destination address: 7D1:DA8:BF::C0A8:ED0A
```

本机的地址是

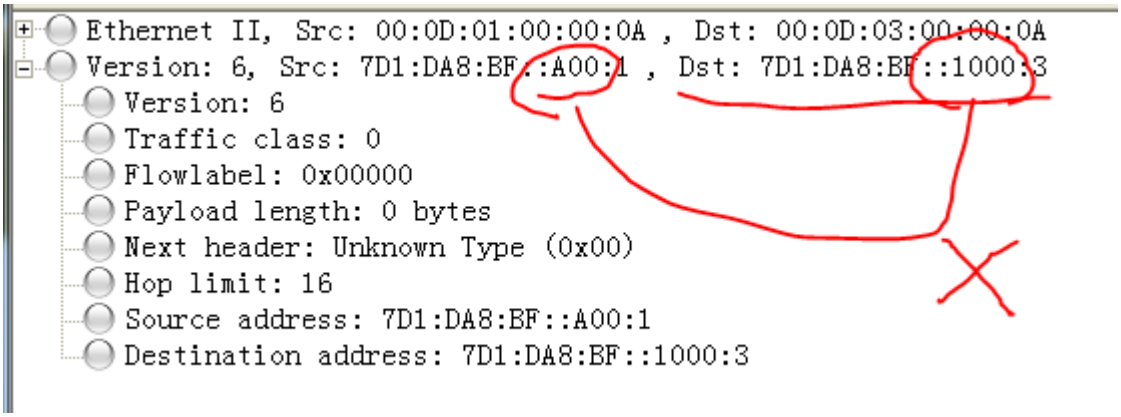
```
7D1:DA8:BF::A00:3
```

IPv6 分组转发功能：

编号	时间	源地址	目的地址	协议	数据包描述	实验描述
1	Sat Apr 20 19:50:0...	7D1:DA8:BF::A00:1	7D1:DA8:BF::A00:3	IPV6	Version ...	5.1 IPv6本地接收实验
2	Sat Apr 20 19:50:0...	7D1:DA8:BF::A00:1	7D1:DA8:BF::1000:3	IPV6	Version ...	5.2 无法获得IPv6路由信息
3	Sat Apr 20 19:50:2...	7D1:DA8:BF::A00:1	7D1:DA8:BF::B00:3	IPV6	Version ...	5.3 IPv6正确转发实验
4	Sat Apr 20 19:50:2...	7D1:DA8:BF::A00:1	7D1:DA8:BF::B00:3	IPV6	Version ...	5.3 IPv6正确转发实验



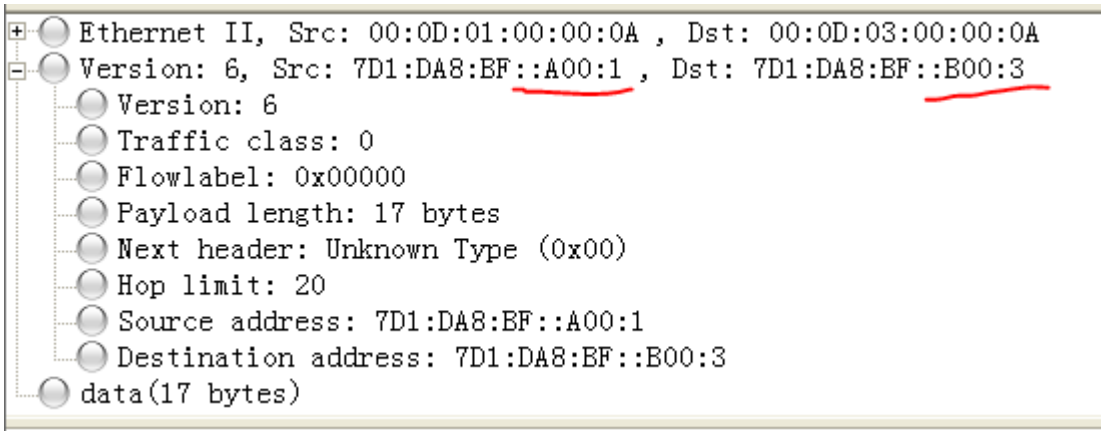
无法获得 IPv6 路由信息



分析：
路由表中没有和目的网络所对应的路由信息，所以会丢弃该分组

正确转发实验

下面的转发的数据包，目的地址不是本机地址，所以需要转发数据报，根据最长掩码匹配原则，匹配到期路由表项，并发送数据包到下一跳。



五、思考问题

-1-IPV6 的分组首部与 IPv4 分组首部的异同：

版本号部分都是 4-bit，相同；

IPV6 的 traffic class 是 8-bit 的一个域，类似于 IPV4 中的 TOS 字段；

V6 头部的长度是固定的，所有没有 IHL 字段；

Payload length 字段是 IPV6 数据报文除了头部之后的长度；

Next header 字段用于区分不同的运输层协议，如 TCP,UDP,SSP； 与 IPv4 总的 protocol 字段功能相同；

Hoplimit 字段同 IPv4 种的 ttl 字段作用相同。

Ipv6 的源地址和目的地址字段都是 128-bit 长度的，可以表示更大的地址范围；

同时部分在 v4 中的字段在 V6 中没有对应的字段：

【1】 分片与重组：IPV6 在中间路由的时候没有考虑分片和重组，这些操作都是在目的端和源地址端进行的，分片和重组是花费较大的操作，把这项功能从路由器转移到端系统可以加速 IP 报文的转发。

【2】 校验和字段：因为传输层和链路层协议已经具有了校验功能，因此这设计者考虑到这 2 部分的校验功能应该足够保证对报文的校验操作，对 IP 数据报的快速处理是减少传输时延的主要因素，而 IPv4 中的校验和字段包含了 TTL 字段，每次转发都需要进行重新计算校验和字段部分，这个操作同分片和重组一样都是花费很大的操作。

-2-分析为什么 IPv6 地址结构这样定义：

IPv6 地址定义根据 RFC4291 的解释：

IPv6 地址类型主要有 3 中，单播，多播，任播；没有广播地址，广播可以通过多播来实现。

子网编号，子网前缀；

一般 IPv6 地址的格式是： ipv6-address/prefix-length

Ipv6 地址是 union 类型，最常用的格式是 unsigned short wAddr[8]的表示形式；

上面是单播地址类型的地址结构；

全球的单播地址

任播地址：

多播地址（略）

我在试验中用 char 数组对地址进行解析的，没有进行打印地址操作，如果在测试中相对地址进行打印操作化，优先采用 long 型和 unsigned short 型来解析 ipv6 的地址。

-3-根据 IPv6 地址结构说明 IPv6 报文头中的 IPv6 地址字段在发送到网络中时是否需要进行字节序转换。

如果采用 char 数组来解析和封装 IPv6 的地址字段可以不考虑字节序的处理，但是如果按照 long 或者 unsigned short 型来解析 IP 地址的话需要进行网络字节序的转换。

-4-解释 IPv6 的路由匹配算法，它和 IPv4 的路由匹配算法有何不同？

IPv6 路由匹配算法：（根据 char 数组的地址结构来比较地址）

```
bool is_equal_subnet(int masklen, ipv6_addr t1, ipv6_addr t2){
    int num = masklen/8;
    int remain = masklen%8;
    for(int i=0; i<num; i++)
        if(t1.bAddr[i] != t2.bAddr[i])  return false;
    char a = t1.bAddr[num] & (0xff << (8-remain));
    char b = t2.bAddr[num] & (0xff << (8-remain));
    if(a == b)
        return true;
    else
        return false;
}
```

IPv4 路由匹配算法：(重载了 set 集合的比较函数，然后匹配路由时，直接查找，调用了 find()库函数就可以了。)

```
class RouteSortCriterion {
public:
    bool operator() (const Route_node &a, const Route_node &b) const {
        unsigned int sub_a = a.dest &  (0xffffffff << (32-a.masklen));
        unsigned int sub_b = b.dest &  (0xffffffff << (32-b.masklen));
        if(sub_a < sub_b) return true;
        else return false;
    }
};
//定义一个set集合
set<Route, RouteSortCriterion> s;

set<Route, RouteSortCriterion>::iterator iter;
int j;
for(j=32; j>0; j--){
    iter = s.find(temp[j]);
    if(iter != s.end()) break;
}
```

六、实验中遇到的问题

【1】最开始的时候处理 IPv6 地址结构的时候遇到了关于 IPv6 地址的字节序转换问题，开始时采用的 long 指针类型来处理的，后来对 IPv6 地址采用了它的 char 数组类型结构，不需要考虑字节序的问题。

【2】还有就是匹配路由表型的时候最开始是想计算子网编号的方法，但是由于数据的范围太大，double 类型的数据不够用，所以后来采用了逐个 char 值比较的方法来实现；

七、 实验的启示/意见和建议

此次实验大约花费了 **10+**小时吧！包括写代码+调代码+写报告