计算机网络原理实验报告——RIP实验

张钰晖 计55 2015011372 yuhui-zh15@mails.tsinghua.edu.cn

零、实验背景

RIP(Routing Information Protocol)是目前因特网上最简单的一种路由协议。RIP采用距离向量路由算法，该算法早在因特网的前身ARPANET网络中就已经被广泛使用。20世纪70年代中期，Xerox公司提出了一套被称为XNS(Xerox Network System)的网络协议系统。这套协议系统包含了XNS RIP协议，这就是现在所使用的RIP的最早原型。

20世纪80年代，加州大学伯克利分校在开发UNIX系统时，在路由守护进程routed程序中设计实现了RIP软件。routed程序被绑定在BSD UNIX系统中一起推出。随着UNIX操作系统的普及，RIP/routed也逐渐被推广，被广泛的应用于早期网络中网络节点之间交换路由信息，成为了中小型网络中最基本的路由协议/软件。

本实验提供了交互实验和C语言编程实现两种方式，通过在NetRiver试验系统上实现RIP，加深对路由协议基本原理和距离向量算法的理解。

一、实验目的

通过交互实验和编程实验，深入理解RIP的工作原理，了解RIP的分组接和发送流程以及路由表的维护，进一步掌握计算机网络中的核心技术——路由技术，并培养路由协议编程开发能力。

二、实验原理

（1）RIP简述

RIP协议的RFC文本在1988年6月被正式推出，它综合了实际应用中许多实现版本的特点，同时为版本的兼容互通性提供了可靠的依据。由于RIPv1中存在着一些缺陷，再加上网络技术的发展，有必要对RIP版本进行相应的改进。1994年11月，RFC1723对RIPv1的分组结构进行了扩展，增加一些新的网络功能。1998年11月，RIPv2的标准RFC文本被正式提出，它在协议分组的路由表项中增加了子网掩码信息，同时增加了安全认证、不同路由协议交互等功能。

随着OSPF、IS-IS等域内路由协议的出现，许多人认为RIP协议软件已经过时。尽管RIP在协议性能和网络适应能力上远远落后于后来提出的路由协议，但是RIP仍然具有自身的特点。首先，在小型的网络环境中，从使用的网络带宽以及协议配置和管理复杂程度上看，RIP的运行开销很小；其次，与其他路由协议相比，RIP使用简单的距离-向量算法，实现更容易；最后，由于历史的原因，RIP的应用范围非常广，在未来的几年中仍然会使用在各种网络环境中。因此，在路由器的设计中，RIP协议是不可缺少的路由协议之一，RIP协议的实现效率高低对路由器系统的路由性能起着重要的作用。

（2）RIP的基本特点

RIP协议使用UDP的520端口进行路由信息的交互，交互的RIP信息分组主要是两种类型：请求（request）分组和响应（response）分组。请求分组用来向相邻运行RIP的路由器请求路由信息，响应分组用来发送本地路由器的路由信息。RIP协议使用距离-向量路由算法，因此发送的路由信息可以用序偶<vector, distance>来表示，在实际分组中，vector用路由的目的地址address表示，而distance用该路由的距离度量值metric表示，metric值规定为从本机到达目的网络路径上经过的路由

器数目，metric的有效值为1到16，其中16表示网络不可到达，可见RIP协议运行的网络规模是有限的。

当系统启动时，RIP协议处理模块在所有RIP配置运行的接口处发出request分组，然后RIP协议就进入了循环等待状态，等待外部RIP分组（包括请求分组和响应分组）的到来；而接收到request分组的路由器则应当发出包含它们路由表信息的response分组。

当发出请求的路由器接收到一个response分组后，它会逐一处理收到的路由表项内容。如果分组中的表项为新的路由表项，那么就会向路由表加入该表项。如果该分组表项已经在路由表中存在，那么首先判断这个收到的路由更新信息是哪个路由器发送过来的。如果就是这个表项的源路由器（即当初发送相应路由信息从而导致这个路由表项的路由器），则无论该现有表项的距离度量值（metric）如何，都需要更新该表项；如果不是，那么只有当更新表项的metric值小于路由表中相应表项metric值时才需要替代原来的表项。

为了保证路由的有效性，RIP协议规定：每隔30秒，重新广播一次路由信息；若连续三次没有收到RIP广播的路由信息，则相应的路由信息失效。

水平分裂算法是一种避免路由回路和加快路由收敛的技术。由于路由器可能受到它自己发送的路由信息，而这种信息是无用的。水平分裂算法规定，路由器不反向通告任何从邻居收到的路由更新信息。

（3）RIPv2的分组结构

RIPv2的分组结构如图所示。每个RIPv2分组都由RIP头部(RIP Header)和最多25个RIP路由项(RIP Entity)组成。如果路由表的路由表项目数目大于25时，就需要多个RIP分组来完成路由信息的传播过程。由图可知，RIP头部包括Command、Version、must be zero 3个字段，一个RIP路由项包括Address Family Identifier、Route Tag、IP Address、Subnet Mask、Next Hop和Metric 6个字段。可以推算出RIPv2分组的最大长度为512字节，其中包括8字节的UDP头部、4字节的RIP头部和最多500字节(20字节x25)路由项。下面对RIPv2的分组字段逐一介绍：

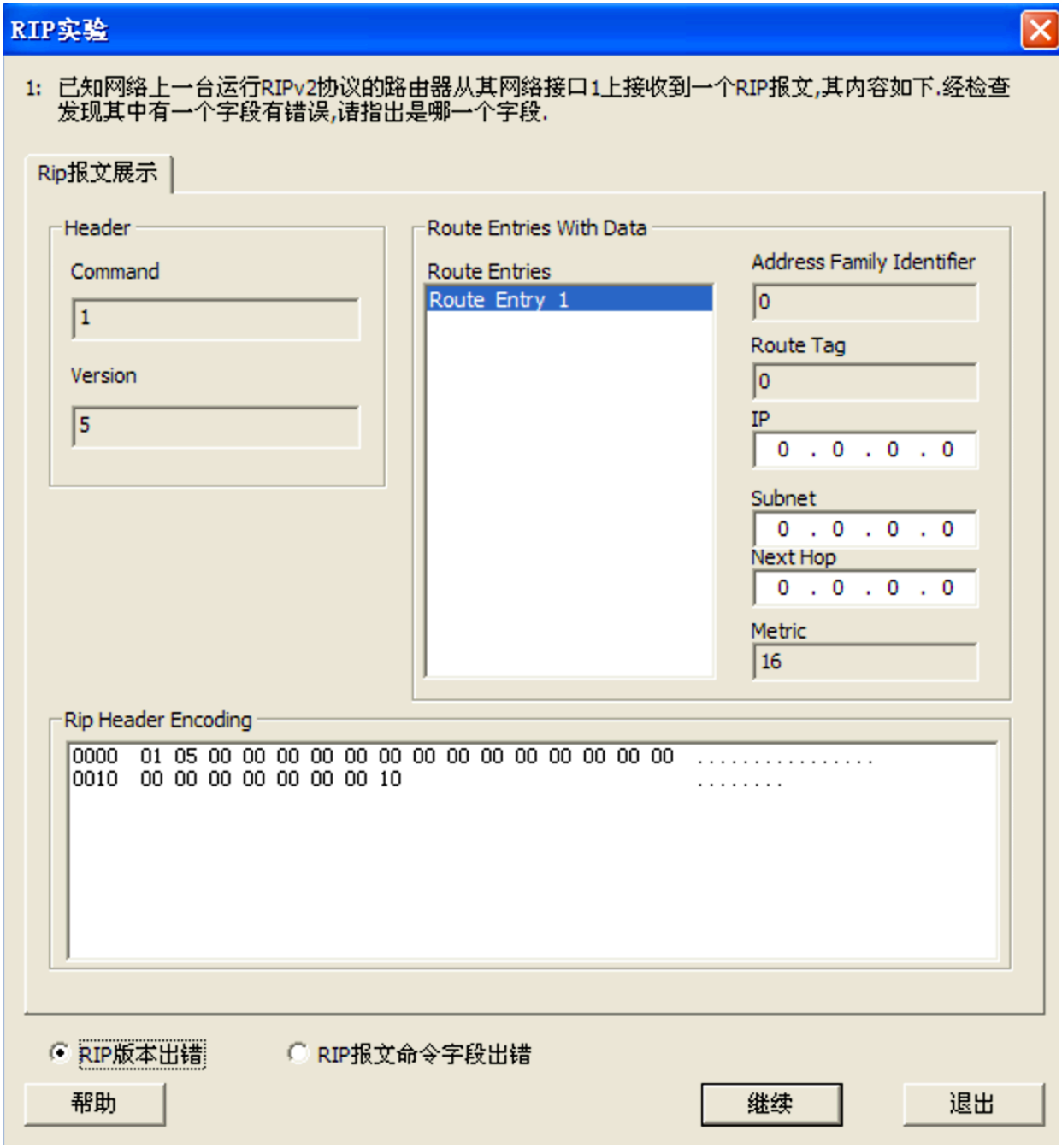
- Command：表示RIP分组的类型，目前RIP只支持两种分组类型，分别是请求分组（request 1）和响应（response 2）分组。
- Version：表示RIP分组的版本信息，RIPv2分组中此字段为2。

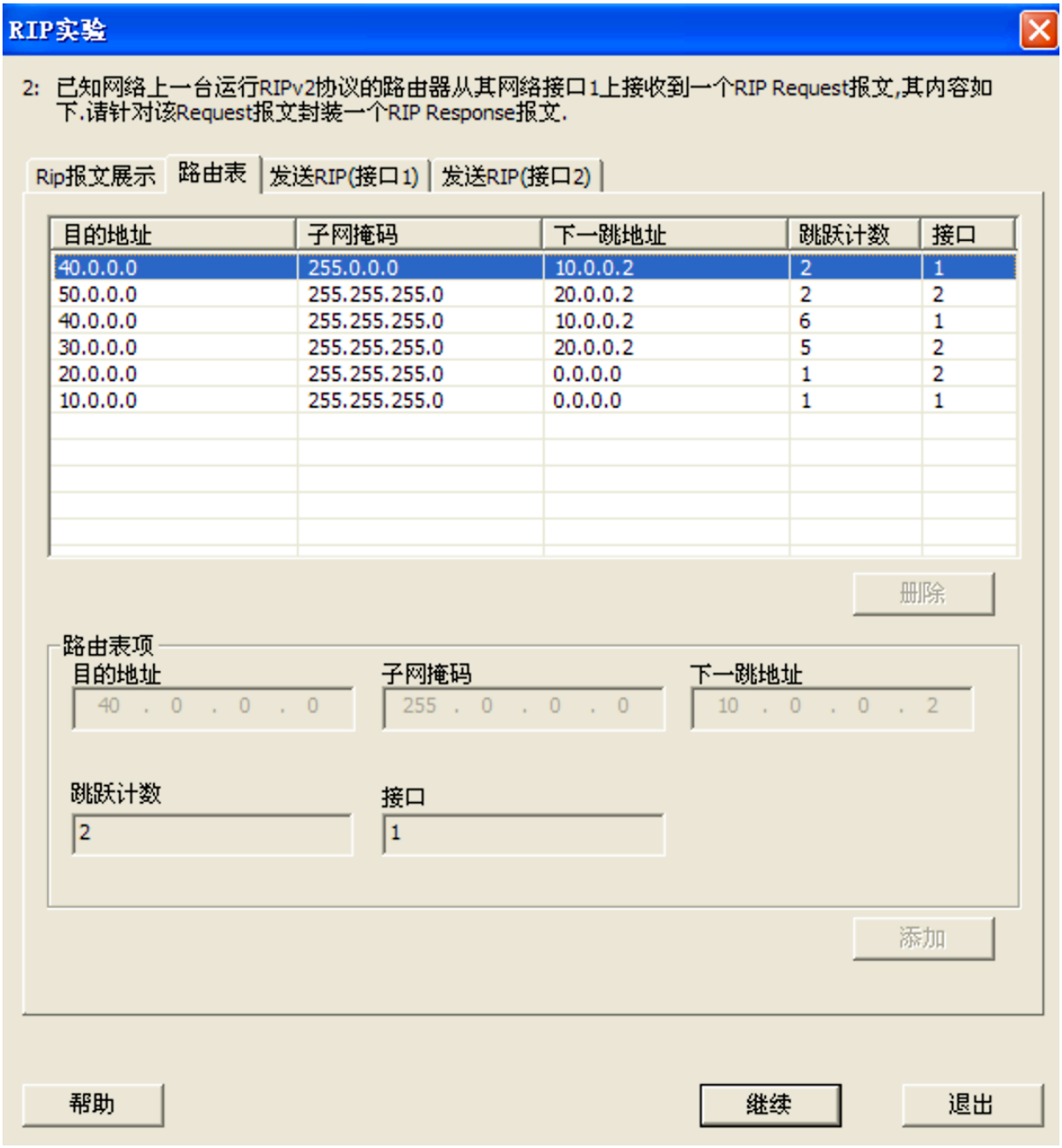
- must be zero：保留字段，取值为0.
- Address Family Identifier：表示路由信息所属的地址族，目前RIP中规定此字段必须为2，表示使用IP地址族。
- IP Address：表示路由信息对应的目的地IP地址，可以是网络地址、子网地址以及主机地址。
- Subnet Mask：子网掩码，应用于IP地址产生非主机部分地址，为0时表示不包括子网掩码部分，使得RIP能够适应更多的环境。
- Next Hop：下一跳IP地址，可以对使用多路由协议的网络环境下的路由进行优化。
- Metric：表示从本路由器到达目的地的距离，目前RIP将路由路径上经过的路由器数作为距离度量值。

一般来说，RIP发送的请求分组和响应分组都符合图的分组结构格式，但是当需要发送请求对方路由器全部路由表信息的请求分组时，RIP使用另一种分组结构，此分组结构中路由信息项的地址族标识符字段为0，目的地址字段为0，距离度量字段为16。

三、交互实验

实验系统内置了交互实验平台，共五道题，通过回答相应问题、构造相应的分组可以更深入的理解RIP协议的原理。


RIP实验
1: 已知网络上一台运行RIPv2协议的路由器从其网络接口1上接收到一个RIP报文,其内容如下.经检查发现其中有一个字段有错误,请指出是哪一个字段.
Rip报文展示
Header
Command
1
Version
5
Route Entries With Data
Route Entries
Route Entry 1
Address Family Identifier
0
Route Tag
0
IP
0 . 0 . 0 . 0
Subnet
0 . 0 . 0 . 0
Next Hop
0 . 0 . 0 . 0
Metric
16
Rip Header Encoding
0000 01 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0010 00 00 00 00 00 00 00 10 ....
RIP版本出错
RIP报文命令字段出错
帮助
继续
退出



学期	序号	学号	姓名	院系	班级	实验名称	实验日期	实验结果	总成绩	程序	报告
2017秋季	1	2015011372	张钰晖	计算机科学与技术系		滑动窗口协议实验	2017-11-04		0		
2017秋季	2	2015011372	张钰晖	计算机科学与技术系		RIP协议实验	2017-12-03		0		
2017秋季	3	2015011372	张钰晖	计算机科学与技术系		RIP协议交互实验	2017-12-05		0		

四、编程实验

（1）实验要求

本实验要求在NetRiver实验系统环境中，用C语言实现。需充分理解RIP，根据RIP的流程设计RIP的分组处理和超时处理函数，并实现如下功能：RIP分组有效性检查；处理Request分组；处理Response分组；路由表项超时删除；路由表项定时发送。

为完成本实验的实验要求，需要完成的内容具体包括：

- 对客户端接收到的RIP分组进行有效性检查：对客户端接收到的RIP协议分组进行合法性检查，丢弃存在错误的分组并指出错误原因；

- 处理Request分组：正确解析并处理RIP协议的Request分组，并能够根据分组的内容以及本地路由表组成相应的Response分组，回复给Request分组的发送者，并实现水平分割；
- 处理Response分组：正确解析并处理RIP协议的Response分组，并根据分组中携带的路由信息更新本地路由表；
- 路由表项超时删除：处理来自系统的路由表项超时消息，并能够删除指定的路由；
- 路由表项定时发送：实现定时对本地的路由进行广播的功能，并实现水平分割。

客户端软件模拟一个网络中的路由器，在其中2个接口运行RIP协议，接口编号为1和2，每个接口均与其他路由器连接，通过RIP协议交互路由信息。

（2）接口函数和全局变量说明

需要实现的接口函数

1. RIP分组处理函数

```
int stud_rip_packet_recv(char *pBuffer, int bufferSize, UINT8 iNo, UINT32
srcAdd)
```

- 参数pBuffer：指向接收到的RIP分组内容的指针。其指向的数据为网络序，是从RIP报头开始的。
- 参数bufferSize：接收到的RIP分组的长度。
- 参数iNo：接收该分组的接口号。
- 参数srcAdd：接收到的分组的源IP地址。
- 返回值：成功为0，失败为-1。

说明：当系统收到RIP分组时，会调用此函数，学生编写此函数，应该实现如下功能：

① 对RIP分组进行合法性检查，若分组存在错误，则调用ip_DiscardPkt函数，并在type参数中传入错误编号。错误编号的宏定义如下：

```
#define STUD_RIP_TEST_VERSION_ERROR  //RIP版本错误
#define STUD_RIP_TEST_COMMAND_ERROR  //RIP命令错误
```

② 对于正确的分组，则根据分组的command域，判断分组类型，即Request或Response分组类型。

③ 对于Request分组，应该将根据本地的路由表信息组成Response分组，并通过rip_sendIpPkt函数发送出去。注意，由于实现水平分割，组Response分组时应该检查该Request分组的来源接口，Response分组中的路由信息不包括来自该来源接口的路由。

④ 对于Response分组，应该提取出该分组中携带的路由表项，针对每个Response分组的路由表项，可进行如下操作：

- 若该Response路由表项为新的表项，即该Response路由表项中的Ip Address与所有本地路由表项的IP Address都不相同，则将其Metric值加1之后添加到本地路由表中。如果Metric加1之后等于16，则表明该Response路由表项已经失效，此时无需添加该表项。
- 若该Response路由表项已经在本地路由表中存在，且两者的Next Hop字段相同，则将其Metric值加1后，更新到本地路由表项的Metric字段。如果Metric加1之后等于16，应置该本地路由表项为无效。
- 若该Response路由表项已经在本地路由表中存在，且两者的Next Hop字段不同，则只有当

Response路由表项中的Metric值小于本地路由表项中的Metric值时，才将其Metric值加1后，更新到本地路由表项的Metric字段，并更新Next Hop字段。如果Metric加1之后等于16，应置该本地路由表项为无效。

2. RIP超时处理函数

```
void stud_rip_route_timeout(UINT32 destAdd, UINT32 mask, unsigned char msgType)
```

- 参数destAdd：路由超时消息中路由的目标地址。
- 参数mask：路由超时消息中路由的掩码。
- 参数msgType：消息类型，包括以下两种定义：

```
#define RIP_MSG_SEND_ROUTE
#define RIP_MSG_DELE_ROUTE
```

说明：RIP协议每隔30秒，重新广播一次路由信息，系统调用该函数并置msgType为RIP_MSG_SEND_ROUTE来进行路由信息广播。该函数应该在每个接口上分别广播自己的RIP路由信息，即通过rip_sendIpPkt函数发送RIP Response分组。由于实现水平分割，分组中的路由信息不包括来自该接口的路由信息。

RIP协议每个路由表项都有相关的路由超时计时器，当路由超时计时器过期时，该路径就标记为失效的，但仍保存在路由表中，直到路由清空计时器过期才被清掉。当超时定时器被触发时，系统会调用该函数并置msgType为RIP_MSG_DELE_ROUTE，并通过destAdd和mask参数传入超时的路由项。该函数应该置本地路由的对应项为无效，即metric值置为16。

系统提供的接口函数

```
void rip_sendIpPkt(unsigned char* pData, UINT16 len, unsigned short dstPort, UINT8 iNo);
```

- 参数pData：指向要发送的RIP分组内容的指针。其指向的数据应该为网络序，是从RIP报头开始的。
- 参数len：要发送的RIP分组的长度 。
- 参数dstPort：要发送的RIP分组的目的端口
- 参数iNo：发送该分组的接口号

说明：该函数用于发送IP分组，可供编写程序时调用。

系统提供的全局变量

```
extern struct stud_rip_route_node *g_rip_route_table;
```

系统以单向链表存储RIP路由表，需要利用此表存储RIP路由，供客户端软件检查。该全局变量为系统中RIP路由表链表的头指针， 其中，stud_rip_route_node结构定义如下：

```c
typedef struct stud_rip_route_node {
    unsigned int dest;
    unsigned int mask;
    unsigned int nexthop;
    unsigned int metric;
    unsigned int if_no;
    struct stud_rip_route_node *next;
};

```

五、思考题

1. RIP协议超时删除处理中，当某个路由表项无效时，其Metric值为何要填写为16，如果不这样填写会出现什么情况?

   答：RIP协议采用距离向量算法，在默认情况下，RIP使用一种非常简单的度量制度：距离就是通往目的站点所需经过的链路数，取值为1~15，数值16表示无穷大，亦即与此路由相对应的目的网络不可达。

   RIP数据包中的度量值字段的长度是32位，就理论而言，RIP路由的度量值最多可达2^32-1跳。虽然度量值字段的取值范围可以很大，但将RIP路由的度量值上限定为15，是为了避免计数到无穷大（即路由环路）问题。在拥有数百台路由器的大型网络中，若将RIP路由的度量值上限（无穷大值）定得过高，一旦发生路由环路，就会使得路由的收敛时间过长。将RIP路由的度量值上限（无穷大值）定义为16，是为了缩短路由收敛时间。此外，把（有效）RIP路由的度量值（跳数）限制为15，还有另外一个原因，那就是RIP路由协议专为小型网络而设计，不适用于数据包转发路径中路由器台数过15的大型网络。

   因此在超时删除处理中，当某个路由表项无效时，将Metric填写为16即认为该表项无效。如果不这样填写会导致该无效的表项被发送给相邻路由器，引发错误。

2. 试比较RIP和OSPF的异同点和适用范围。

   答：RIP协议是一种传统的路由协议，适合比较小型的网络，但是当前Internet网络的迅速发展和急剧膨胀使RIP协议无法适应今天的网络。OSPF协议则是在Internet网络急剧膨胀的时候制定出来的，它克服了RIP协议的许多缺陷。

   相同点：

   RIP协议和OSPF协议都是动态路由协议，都是内部路由协议（在AS内运行）。

   不同点：

   RIP是距离矢量路由协议，OSPF是链路状态路由协议。RIP和OSPF管理距离分别是：120和110

   RIP协议一条路由有15跳（网关或路由器）的限制，如果一个RIP网络路由跨越超过15跳（路由器），则它认为网络不可到达，而OSPF对跨越路由器的个数没有限制。

   OSPF协议支持可变长度子网掩码（VLSM），RIP则不支持，这使得RIP协议对当前IP地址的缺乏和可变长度子网掩码的灵活性缺少支持。

   RIP协议不是针对网络的实际情况而是定期地广播路由表，这对网络的带宽资源是个极大的浪费，特别对大型的广域网。OSPF协议的路由广播更新只发生在路由状态变化的时候，采用IP多路广播来发送链路状态更新信息，这样对带宽是个节约。

RIP网络是一个平面网络，对网络没有分层。OSPF在网络中建立起层次概念，在自治域中可以划分网络域，使路由的广播限制在一定的范围内，避免链路中继资源的浪费。

OSPF在路由广播时采用了授权机制，保证了网络安全。

上述两者的差异显示了OSPF协议后来居上的特点，RIP比较适合于小型网络，最多十几台路由器，用的比较少。OSPF比较流行，也比较复杂，根据链路状况动态更新，其先进性和复杂性使它适应了今天日趋庞大的Internet网，并成为主要的互联网路由协议。

六、问题及解决办法

1. 在超时处理函数中，当消息类型为RIP_MSG_SEND_ROUTE，应该如何操作?

答：应该在每个接口上分别广播自己的RIP路由信息，即通过rip_sendIpPkt函数发送RIP Response分组。由于实现水平分割，分组中的路由信息不包括来自该接口的路由信息。由于本题只有两个接口，故调用发送流程两次即可。

2. 如何置本地路由表项为无效?

答：直接将该表项Metric值置为规定的无效值(16)即可，理由见思考题。

七、实验小结

通过阅读实验说明，笔者理解了RIP协议的基本原理，并且通过编程实现，更深入的了解了RIP协议的细节。尽管这次实验只是一次简单的尝试，但通过本次实验，我体会到了RIP协议的简洁与精妙，在实验中充分理解了RIP协议。

本次实验说明指导手册已经较为详尽，笔者在此基础上对代码增加了详细的注释。本次实验不算很复杂，实现过程中没有遇到特别严重的问题，大部分通过输出调试信息和尝试得到了解决，在这个过程中我对网络数据的传输也得到了更加深刻的理解。

希望今后能通过类似的编程实验更充分的理解计算机网络原理。

八、源代码

```
#include "sysinclude.h"

/*
    - 参数pData：指向要发送的RIP分组内容的指针。其指向的数据应该为网络序，是从RIP报头开始的。
    - 参数len：要发送的RIP分组的长度 。
    - 参数dstPort：要发送的RIP分组的目的端口
    - 参数iNo：发送该分组的接口号
*/
extern void rip_sendIpPkt(unsigned char *pData, UINT16 len,unsigned short dstPort,UINT8 iNo);

/*
    系统以单向链表存储RIP路由表，学生需要利用此表存储RIP路由，供客户端软件检查。该全局变量为系统中RIP路由表链表的头指针， 其中，stud_rip_route_node结构定义如下：
    typedef struct stud_rip_route_node {
        unsigned int dest;
```

```
        unsigned int mask;
        unsigned int nexthop;
        unsigned int metric;
        unsigned int if_no;
        struct stud_rip_route_node *next;
    };
*/
extern struct stud_rip_route_node *g_rip_route_table;

// RIP使用UDP的520端口进行路由信息交互
#define PORT 520

// RIP表头
struct rip_header {
    unsigned char command;
    unsigned char version;
    unsigned short mustbezero;
};

// RIP表项
struct rip_entry {
    unsigned short addressid;
    unsigned short routetag;
    unsigned int dest;
    unsigned int mask;
    unsigned int nexthop;
    unsigned int metric;
};

// RIP分组
struct rip_packet {
    rip_header ripheader;
    rip_entry ripentry[25];
};

/*
    - 参数pBuffer：指向接收到的RIP分组内容的指针。其指向的数据为网络序，是从RIP报头开始的。
    - 参数bufferSize：接收到的RIP分组的长度。
    - 参数iNo：接收该分组的接口号。
    - 参数srcAdd：接收到的分组的源IP地址。
    - 返回值：成功为0，失败为-1。
*/
int stud_rip_packet_recv(char *pBuffer, int bufferSize, UINT8 iNo, UINT32 srcAdd) {
    // 对RIP分组进行合法性检查，若分组存在错误，则调用ip_DiscardPkt()函数，并在type参数中传入错误编号
    unsigned char command = pBuffer[0];
    unsigned char version = pBuffer[1];
```

```
61      if (command != 1 && command != 2) {
62          ip_DiscardPkt(pBuffer, STUD_RIP_TEST_COMMAND_ERROR);
63          return -1;
64      }
65      if (version != 2) {
66          ip_DiscardPkt(pBuffer, STUD_RIP_TEST_VERSION_ERROR);
67          return -1;
68      }
69      // 对于正确的分组，则根据分组的command域，判断分组类型，即Request或Response
    分组类型
70      // 对于Request分组，应该将根据本地的路由表信息组成Response分组，并通过
    rip_sendIpPkt()函数发送出去。
71      if (command == 1) { //request,发送回当前表项
72          rip_packet rippacket;
73          stud_rip_route_node* pointer = g_rip_route_table;
74          while (pointer != NULL) {
75              memset(&rippacket, 0, sizeof(rip_packet));
76              rippacket.ripheader.command = 2;
77              rippacket.ripheader.version = 2;
78              int cnt = 0;
79              while (pointer != NULL && cnt < 25) { //每个rip_packet最大25
    个rip_entry
80                  // 由于实现水平分裂算法，封装Response分组时应该检查该Request分
    组的来源接口，Response分组中的路由信息不包括来自该来源接口的路由
81                  if (pointer->if_no != iNo) {
82                      rippacket.ripentry[cnt].addressid = htons(2);
83                      rippacket.ripentry[cnt].routetag = 0;
84                      rippacket.ripentry[cnt].dest = htonl(pointer-
    >dest);
85                      rippacket.ripentry[cnt].mask = htonl(pointer-
    >mask);
86                      rippacket.ripentry[cnt].nexthop = htonl(pointer-
    >nexthop);
87                      rippacket.ripentry[cnt].metric = htonl(pointer-
    >metric);
88                      ++cnt;
89                  }
90                  pointer = pointer->next;
91              }
92              UINT16 len = 4 + cnt * sizeof(rip_entry);
93              rip_sendIpPkt((unsigned char*)(&rippacket), len, PORT,
    iNo);
94          }
95      }
96      // 对于Response分组，应该提取出该分组中携带的所有路由表项，针对每个Response分
    组的路由表项，进行不同操作
97      else if (command == 2) {
98          rip_packet rippacket = *(rip_packet*)pBuffer;
99          int cnt = (bufferSize - 4) / sizeof(rip_entry);
```

```
100         for (int i = 0; i < cnt; ++i) {
101             rippacket.ripentry[i].dest =
ntohl(rippacket.ripentry[i].dest);
102             rippacket.ripentry[i].mask =
ntohl(rippacket.ripentry[i].mask);
103             rippacket.ripentry[i].nexthop =
ntohl(rippacket.ripentry[i].nexthop);
104             rippacket.ripentry[i].metric =
ntohl(rippacket.ripentry[i].metric);
105             // 根据IP Address与Mask搜索表项
106             stud_rip_route_node* pointer = g_rip_route_table;
107             while (pointer != NULL) {
108                 if (pointer->dest == rippacket.ripentry[i].dest &&
pointer->mask == rippacket.ripentry[i].mask) break;
109                 pointer = pointer->next;
110             }
111             // 若该Response路由表项为新的表项，即该Response路由表项中的Ip
Address与所有本地路由表项的IP Address都不相同，则将其Metric值加1之后添加到本地路
由表中
112             // 如果Metric加1之后等于16，则表明该Response路由表项已经失效，此时
无需添加该表项
113             if (pointer == NULL) {
114                 if (rippacket.ripentry[i].metric + 1 < 16) {
115                     stud_rip_route_node* newnode = new
stud_rip_route_node;
116                     newnode->dest = rippacket.ripentry[i].dest;
117                     newnode->mask = rippacket.ripentry[i].mask;
118                     newnode->nexthop = srcAdd; //nexthop为srcAdd(接收到的
分组的源IP地址)
119                     newnode->metric = rippacket.ripentry[i].metric + 1;
120                     newnode->if_no = iNo;
121                     newnode->next = g_rip_route_table;
122                     g_rip_route_table = newnode;
123                 }
124             }
125             // 若该Response路由表项已经在本地路由表中存在，且两者的Next Hop字段
相同，则将其Metric值加1后，更新到本地路由表项的Metric字段
126             // 如果Metric加1之后等于16，应置该本地路由表项为无效
127             else if (pointer->nexthop == srcAdd) {
128                 pointer->metric = (rippacket.ripentry[i].metric + 1 <
16)? rippacket.ripentry[i].metric + 1: 16;
129                 pointer->if_no = iNo;
130             }
131             // 若该Response路由表项已经在本地路由表中存在，且两者的Next Hop字段
不同，则只有当Response路由表项中的Metric值小于本地路由表项中的Metric值时，才将其
Metric值加1后，更新到本地路由表项的Metric字段，并更新Next Hop字段。
132             // 如果Metric加1之后等于16，应置该本地路由表项为无效
133             else {
```

```
                if (pointer->metric > rippacket.ripentry[i].metric) {
//存在且nexthop与srcAdd不等,比较后更新
                    pointer->metric = (rippacket.ripentry[i].metric + 1
< 16) ? rippacket.ripentry[i].metric + 1: 16;
                    pointer->nexthop = srcAdd;
                    pointer->if_no = iNo;
                }
            }
        }
    }
    return 0;
}


/*
    - 参数destAdd：路由超时消息中路由的目标地址。
    - 参数mask：路由超时消息中路由的掩码。
    - 参数msgType：消息类型，包括以下两种定义：
        #define RIP_MSG_SEND_ROUTE
        #define RIP_MSG_DELE_ROUTE
*/
void stud_rip_route_timeout(UINT32 destAdd, UINT32 mask, unsigned char
msgType)
{
    // RIP协议每隔30秒，重新广播一次路由信息，系统调用该函数并置msgType为
RIP_MSG_SEND_ROUTE来进行路由信息广播。该函数应该在每个接口上分别广播自己的RIP路由
信息，即通过rip_sendIpPkt函数发送RIP Response分组。由于实现水平分割，分组中的路
由信息不包括来自该接口的路由信息
    if (msgType == RIP_MSG_SEND_ROUTE) {
        rip_packet rippacket;
        // 接口号iNo为1
        stud_rip_route_node* pointer = g_rip_route_table;
        while (pointer != NULL) {
            memset(&rippacket, 0, sizeof(rip_packet));
            rippacket.ripheader.command = 2;
            rippacket.ripheader.version = 2;
            int cnt = 0;
            while (pointer != NULL && cnt < 25) {
                if (pointer->if_no != 1) {
                    rippacket.ripentry[cnt].addressid = htons(2);
                    rippacket.ripentry[cnt].routetag = 0;
                    rippacket.ripentry[cnt].dest = htonl(pointer-
>dest);
                    rippacket.ripentry[cnt].mask = htonl(pointer-
>mask);
                    rippacket.ripentry[cnt].nexthop = htonl(pointer-
>nexthop);
                    rippacket.ripentry[cnt].metric = htonl(pointer-
>metric);
```

```
                    ++cnt;
                }
                pointer = pointer->next;
            }
            UINT16 len = 4 + cnt * sizeof(rip_entry);
            rip_sendIpPkt((unsigned char*)(&rippacket), len, PORT, 1);
        }
        // 接口号iNo为2
        pointer = g_rip_route_table;
        while (pointer != NULL) {
            memset(&rippacket, 0, sizeof(rip_packet));
            rippacket.ripheader.command = 2;
            rippacket.ripheader.version = 2;
            int cnt = 0;
            while (pointer != NULL && cnt < 25) {
                if (pointer->if_no != 2) {
                    rippacket.ripentry[cnt].addressid = htons(2);
                    rippacket.ripentry[cnt].routetag = 0;
                    rippacket.ripentry[cnt].dest = htonl(pointer->dest);
                    rippacket.ripentry[cnt].mask = htonl(pointer->mask);
                    rippacket.ripentry[cnt].nexthop = htonl(pointer->nexthop);
                    rippacket.ripentry[cnt].metric = htonl(pointer->metric);
                    ++cnt;
                }
                pointer = pointer->next;
            }
            UINT16 len = 4 + cnt * sizeof(rip_entry);
            rip_sendIpPkt((unsigned char*)(&rippacket), len, PORT, 2);
        }
    }
    // RIP协议每个路由表项都有相关的路由超时计时器，当路由超时计时器过期时，该路径就标记为失效的，但仍保存在路由表中，直到路由清空计时器过期才被清掉。当超时定时器被触发时，系统会调用该函数并置msgType为RIP_MSG_DELE_ROUTE，并通过destAdd和mask参数传入超时的路由项。该函数应该置本地路由的对应项为无效，即metric值置为16
    else if (msgType == RIP_MSG_DELE_ROUTE) {
        stud_rip_route_node* pointer = g_rip_route_table;
        // 根据IP Address与Mask搜索表项
        while (pointer != NULL) {
            if (pointer->dest == destAdd && pointer->mask == mask) break;
            pointer = pointer->next;
        }
        if (pointer != NULL) {
            pointer->metric = 16;
        }
```

```
214     }
215 }
```