



计算机网络实验系统

实验指导书

NetRiver 系列

目 录

1	滑动窗口协议实验	1
1.1	实验目的	1
1.2	实验要求	1
1.3	实验内容	1
1.4	实验帮助	1
1.4.1	处理流程	4
1.4.2	接口函数说明	5
2	IPv4 协议收发实验	7
2.1	实验目的	7
2.2	实验要求	7
2.3	实验内容	7
2.4	实验帮助	7
2.4.1	处理流程	8
2.4.2	IPv4 分组头部格式	9
2.4.3	接口函数说明	10
3	IPv4 协议转发实验	12
3.1	实验目的	12
3.2	实验要求	12
3.3	实验内容	12
3.4	实验帮助	13
3.4.1	路由表维护	15
3.4.2	转发处理流程	15
3.4.3	实验接口函数	15
4	IPv6 协议收发实验	19
4.1	实验目的	19
4.2	实验要求	19

4.3	实验内容	19
4.4	实验帮助	19
4.4.1	处理流程	20
4.4.2	IPv6 分组头部格式	21
4.4.3	数据结构说明	22
4.4.4	接口函数说明	22
5	IPV6 协议转发实验	24
5.1	实验目的	24
5.2	实验要求	24
5.3	实验内容	24
5.4	实验帮助	25
5.4.1	路由表维护	27
5.4.2	转发处理流程	27
5.4.3	实验接口函数	27
6	RIP协议实验	31
6.1	实验目的	31
6.2	实验要求	31
6.3	实验内容	31
6.4	实验帮助	32
6.4.1	RIP协议介绍	32
6.4.2	实验拓扑	35
6.4.3	需要实现的接口函数	35
6.4.4	系统提供的全局变量	37
6.4.5	系统提供的接口函数	37
7	TCP协议实验	38
7.1	实验目的	38
7.2	实验要求	38
7.3	实验内容	39
7.4	实验帮助	40
7.4.1	TCP报文头部格式	42

7.4.2	数据结构	43
7.4.3	接收和发送处理	43
7.4.4	Socket接口各函数处理	44
7.4.5	系统提供的接口函数	45
7.4.6	需要完成的接口函数	46
8	协议状态机实验	50
8.1	实验目的	50
8.2	实验要求	50
8.3	实验内容	50
8.4	实验帮助	50
8.4.1	BGP协议介绍	50
8.4.2	需要实现的接口函数	55
8.4.3	数据结构定义	59
8.4.4	系统提供的接口函数	59
9	IPSEC协议实验	61
9.1	实验目的	61
9.2	实验要求	61
9.3	实验内容	61
9.4	实验帮助	62
9.4.1	IPSEC协议介绍	62
9.4.2	需要实现的接口函数	67
9.4.3	数据结构以及宏定义	75
9.4.4	系统提供的接口函数及全局变量	78
10	移动IP协议实验	83
10.1	实验目的	83
10.2	实验要求	83
10.3	实验内容	83
10.4	实验帮助	84
10.4.1	移动IP协议介绍	84
10.4.2	实验拓扑	88

10.4.3	需要实现的接口函数	88
10.4.4	系统提供的全局变量	92
10.4.5	系统提供的接口函数	93
11	IPV4 协议交互实验	96
11.1	实验目的	96
11.2	实验要求	97
11.3	实验内容	97
11.4	实验帮助	97
12	RIP协议交互实验	99
12.1	实验目的	99
12.2	实验要求	99
12.3	实验内容	99
12.4	实验帮助	100
12.4.1	RIP协议介绍	100
12.4.2	实验拓扑	103
12.4.3	实验界面	103
13	IPV6 协议交互实验	106
13.1	实验目的	106
13.2	实验要求	106
13.3	实验内容	106
13.4	实验帮助	106
14	TCP协议交互实验	108
14.1	实验目的	108
14.2	实验要求	108
14.3	实验内容	108
14.4	实验帮助	108
15	参考文献	111

1 滑动窗口协议实验

1.1 实验目的

计算机网络的数据链路层协议保证通信双方在有差错的通信线路上进行无差错的数据传输，是计算机网络各层协议中通信控制功能最典型的一种协议。

本实验实现一个数据链路层协议的数据传送部分，目的在于使学生更好地理解数据链路层协议中的“滑动窗口”技术的基本工作原理，掌握计算机网络协议的基本实现技术。

1.2 实验要求

在一个数据链路层的模拟实现环境中，用 C 语言实现下面三个数据链路层协议。

- 1) 1 比特滑动窗口协议
- 2) 回退 N 帧滑动窗口协议
- 3) 选择性重传协议

1.3 实验内容

充分理解滑动窗口协议，根据滑动窗口协议，模拟滑动窗口协议中发送端的功能，对系统发送的帧进行缓存并加入窗口等待确认，并在超时或者错误时对部分帧进行重传。

编写停等及退回 N 滑动窗口协议函数，响应系统的发送请求、接收帧消息以及超时消息，并根据滑动窗口协议进行相应处理。

编写选择性重传协议函数，响应系统的发送请求、接受帧消息以及错误消息，并根据滑动窗口协议进行相应处理。

1.4 实验帮助

1) 窗口机制

滑动窗口协议的基本原理就是在任意时刻，发送方都维持了一个连续的允许发送的帧的序号，称为发送窗口；同时，接收方也维持了一个连续的允许接收的帧的序号，称为接收窗口。发送窗口和接收窗口的序号的上下界不

一定要一样，甚至大小也可以不同。不同的滑动窗口协议窗口大小一般不同。发送方窗口内的序列号代表了那些已经被发送，但是还没有被确认的帧，或者是那些可以被发送的帧。下面举一个例子（假设发送窗口尺寸为 2，接收窗口尺寸为 1）：

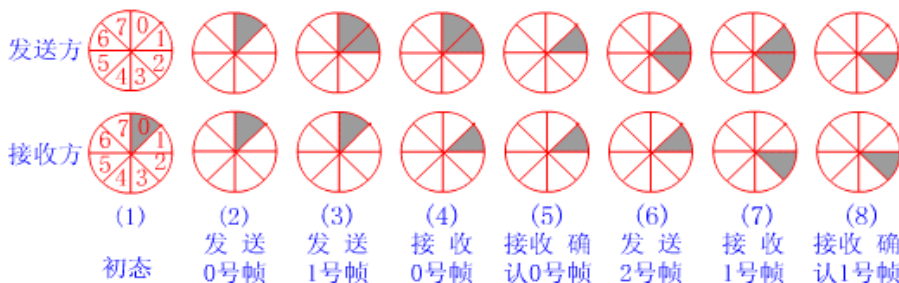


图 6.1 窗口机制示意图

分析：①初始态，发送方没有帧发出，发送窗口前后沿相重合。接收方 0 号窗口打开，等待接收 0 号帧；②发送方打开 0 号窗口，表示已发出 0 帧但尚确认返回信息。此时接收窗口状态不变；③发送方打开 0、1 号窗口，表示 0、1 号帧均在等待确认之列。至此，发送方打开的窗口数已达规定限度，在未收到新的确认返回帧之前，发送方将暂停发送新的数据帧。接收窗口此时状态仍未变；④接收方已收到 0 号帧，0 号窗口关闭，1 号窗口打开，表示准备接收 1 号帧。此时发送窗口状态不变；⑤发送方收到接收方发来的 0 号帧确认返回信息，关闭 0 号窗口，表示从重发表中删除 0 号帧。此时接收窗口状态仍不变；⑥发送方继续发送 2 号帧，2 号窗口打开，表示 2 号帧也纳入待确认之列。至此，发送方打开的窗口又已达规定限度，在未收到新的确认返回帧之前，发送方将暂停发送新的数据帧，此时接收窗口状态仍不变；⑦接收方已收到 1 号帧，1 号窗口关闭，2 号窗口打开，表示准备接收 2 号帧。此时发送窗口状态不变；⑧发送方收到接收方发来的 1 号帧收毕的确认信息，关闭 1 号窗口，表示从重发表中删除 1 号帧。此时接收窗口状态仍不变。

若从滑动窗口的观点来统一看待 1 比特滑动窗口、回退 N 帧和选择性重传这三种协议，它们的差别仅在于各自窗口尺寸的大小不同而已。1 比特滑动窗口协议：发送窗口=1，接收窗口=1；回退 N 帧协议和选择性重传协议：发窗口>1，接收窗口>1。

2) 1 比特滑动窗口协议

当发送窗口和接收窗口的大小固定为 1 时，滑动窗口协议退化为停等协议（stop-and-wait）。该协议规定发送方每发送一帧后就要停下来，等待接收方已正确接收的确认（acknowledgement）返回后才能继续发送下一帧。由于接收方需要判断接收到的帧是新发的帧还是重新发送的帧，因此发送方要为每一个帧加一个序号。由于停等协议规定只有一帧完全发送成功后才能发送新的帧，因而只用一比特来编号就够了。其发送方和接收方运行的流程图如下所示。

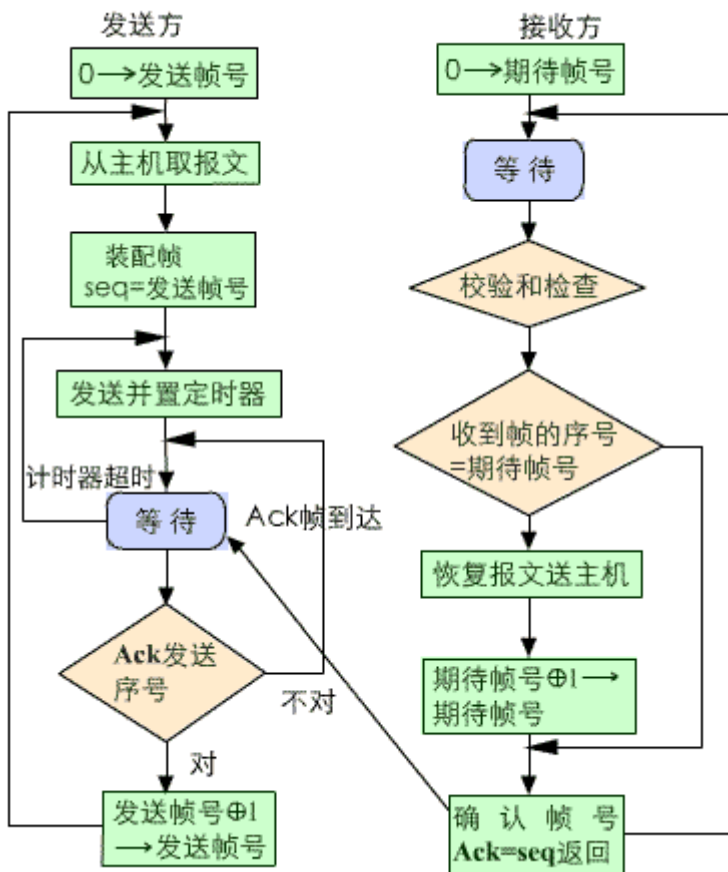


图 6.2 1 比特滑动窗口协议流程图

在两个主机端系统通信的环境中，网络的拓扑可以简化为两台主机直接相连，中间的具体连接方式可以抽象为一条简单的链路，如图 6.3 所示。滑

动窗口实验就是要在实验系统客户端的开发平台上,实现滑动窗口协议发送端的功能。



图 6.3 实验环境网络拓扑结构

1.4.1 处理流程

1) 停等协议和回退 N 帧协议

测试函数包括停等协议测试函数 `stud_slide_window_stop_and_wait` 和回退 N 帧协议测试函数 `stud_slide_window_back_n_frame`, 在下列情况系统会调用学生的测试函数。

当发送端需要发送帧时, 会调用学生测试函数, 并置参数 `messageType` 为 `MSG_TYPE_SEND`, 测试函数应该将该帧缓存, 存入发送队列中。若发送窗口还未打开到规定限度, 则打开一个窗口, 并将调用 `SendFRAMEPacket` 函数将该帧发送。若发送窗口已开到限度, 则直接返回, 相当于直接进入等待状态。

当发送端收到接收端的 `ACK` 后, 会调用学生测试函数, 并置参数 `messageType` 为 `MSG_TYPE_RECEIVE`, 测试函数应该检查 `ACK` 值后, 将该 `ACK` 对应的窗口关闭。由于关闭了窗口, 等待发送的帧可以进入窗口并发送, 因此, 此时若发送队列中存在等待发送的帧应该将一个等待发送的帧发送并打开一个新的窗口。

发送每发送一个帧, 系统都会为他创建一个定时器, 当被成功 `ACK` 后, 定时器会被取消, 若某个帧在定时器超时时间仍未被 `ACK`, 系统则会调用测试函数, 并置参数 `messageType` 为 `MSG_TYPE_TIMEOUT`, 告知测试函数某帧超时, 测试函数应该将根据帧序号将该帧以及后面发送过的帧重新发送。

2) 选择性重传协议

测试函数包括 `stud_slide_window_choice_frame_resend`, 在下列情况下

系统会调用学生的测试函数。

当发送端需要发送帧时，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_SEND`，测试函数应该将该帧缓存，存入发送队列中。若发送窗口还未打开到规定限度，则打开一个窗口，并将调用 `SendFRAMEPacket` 函数将该帧发送。若发送窗口已开到限度，则直接返回，相当于直接进入等待状态。

当发送端收到接收端的 `ACK` 后，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_RECEIVE`，测试函数应该检查 `ACK` 值后，将该 `ACK` 对应的窗口关闭。由于关闭了窗口，等待发送的帧可以进入窗口并发送，因此，此时若发送队列中存在等待发送的帧应该将一个等待发送的帧发送并打开一个新的窗口。

当发送端收到接收端的帧类型为 `NAK`(表示出错的帧号)后，会调用学生测试函数，并置参数 `messageType` 为 `MSG_TYPE_RECEIVE`，测试函数应该检查 `NAK` 值后，系统则会调用测试函数，告知测试函数某帧错误，测试函数应该将根据帧序号将该帧重新发送。

1.4.2 接口函数说明

1.4.2.1 需要实现的接口函数

1) 停等协议测试函数

```
int stud_slide_window_stop_and_wait(char *pBuffer, int bufferSize, UINT8 messageType)
```

参数：

pBuffer: 指向系统要发送或接收到的帧内容的指针，或者指向超时消息中超时帧的序列号内容的指针

bufferSize: pBuffer 表示内容的长度

messageType: 传入的消息类型，可以为以下几种情况

<code>MSG_TYPE_TIMEOUT</code>	某个帧超时
<code>MSG_TYPE_SEND</code>	系统要发送一个帧
<code>MSG_TYPE_RECEIVE</code>	系统接收到一个帧的 <code>ACK</code>

对于 `MSG_TYPE_TIMEOUT` 消息，pBuffer 指向数据的前四个字节为超时帧的序列号，以 `UINT32` 类型存储，在与帧中的序列号比较时，请注意字节序，并进行必要的转换。

对于 MSG_TYPE_SEND 和 MSG_TYPE_RECEIVE 类型消息, pBuffer 指向的数据的结构如以下代码中 frame 结构的定义。

```
typedef enum {data,ack,nak} frame_kind;
typedef struct frame_head
{
    frame_kind kind;        //帧类型
    unsigned int seq;       //序列号
    unsigned int ack;       //确认号
    unsigned char data[100]; //数据
};
typedef struct frame
{
    frame_head head;        //帧头
    unsigned int size;      //数据的大小
};
```

2) 回退 N 帧协议测试函数

```
int stud_slide_window_back_n_frame(char *pBuffer, int bufferSize, UINT8
messageType)
```

参数:

该函数参数定义同停等协议测试函数。

3) 选择性重传协议测试函数

```
int stud_slide_window_choice_frame_resend(char *pBuffer, int bufferSize,
UINT8 messageType)
```

该函数参数定义同停等协议测试函数。

1.4.2.2 系统提供的接口函数

1) 发送帧函数

```
extern void SendFRAMEPacket(unsigned char* pData, unsigned int len);
```

参数:

pData: 指向要发送的帧的内容的指针

len: 要发送的帧的长度

2 IPv4 协议收发实验

2.1 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。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。

2.2 实验要求

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；
- 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

2.3 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。

2.4 实验帮助

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。在后面 IPv4 分组的转发实验中，我们还将深入学习 IPv4 地址在分组转发过程中对选择转发路径的重要作用。

在两个主机端系统通信的环境中，网络的拓扑可以简化为两台主机直接相连，中间的具体连接方式可以抽象为一条简单的链路，如图 1.1 所示。IPv4 分组收发实验就是要在实验系统客户端的开发平台上，实现 IPv4 分组的接收和发送功能。

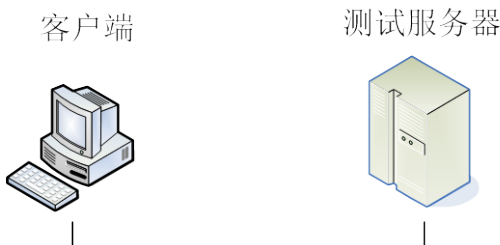


图 1.1 实验环境网络拓扑结构

2.4.1 处理流程

客户端接收到测试服务器发送来的 IPv4 分组后，调用接收接口函数 `stud_ip_recv()`（图 1.2 中接口函数 1）。学生需要在这个函数中实现 IPv4 分组接收处理的功能。接收处理完成后，调用接口函数 `ip_SendtoUp()` 将需要上层协议进一步处理的信息提交给上层协议（图 1.2 中接口函数 2）；或者调用函数 `ip_DiscardPkt()` 丢弃有错误的分组并报告错误类型（图 1.2 中函数 5）。

在上层协议需要发送分组时，会调用发送接口函数 `stud_ip_Upsend()`（图 1.2 中接口函数 3）。学生需要在这个函数中实现 IPv4 分组封装发送的功能。根据所传参数完成 IPv4 分组的封装，之后调用接口函数 `ip_SendtoLower()` 把分组交给下层完成发送（图 1.2 中接口函数 4）。

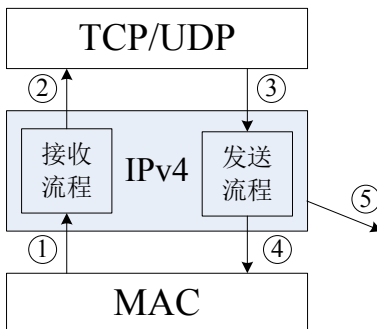


图 1.2 实验接口函数示意图

2.4.1.1 接收流程

在接口函数 `stud_ip_recv()` 中，需要完成下列处理步骤（仅供参考）：

- 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，并说明错误类型。
- 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。

2.4.1.2 发送流程

在接口函数 `stud_ip_Upsend()` 中，需要完成下列处理步骤（仅供参考）：

- 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。

2.4.2 IPv4 分组头部格式

IPv4 分组头部格式如图 1.3 所示，具体字段的定义请参考教材[1]和文献[2]。

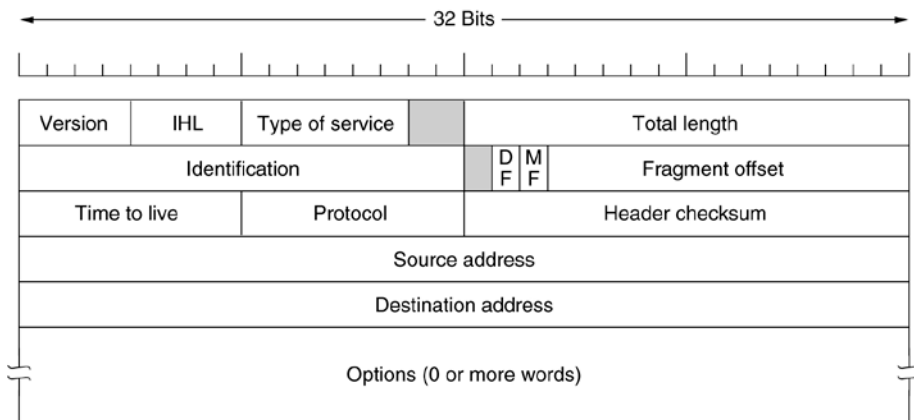


图 1.3 IPv4 分组头部格式

2. 4. 3 接口函数说明

2. 4. 3. 1 需要实现的接口函数

1) 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

2) 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

2. 4. 3. 2 系统提供的接口函数

1) 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:

<code>STUD_IP_TEST_CHECKSUM_ERROR</code>	IP 校验和出错
<code>STUD_IP_TEST_TTL_ERROR</code>	TTL 值出错
<code>STUD_IP_TEST_VERSION_ERROR</code>	IP 版本号错
<code>STUD_IP_TEST_HEADLEN_ERROR</code>	头部长度错
<code>STUD_IP_TEST_DESTINATION_ERROR</code>	目的地址错

2) 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

3) 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

4) 获取本机 IPv4 地址

`unsigned int getIpv4Address()`

参数: 无

3 IPv4 协议转发实验

3.1 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。

3.2 实验要求

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。

3.3 实验内容

实验内容主要包括：

- 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。

3.4 实验帮助

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，以此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。本实验要求按照路由器协议栈的 IPv4 协议功能进行设计实现，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。

实验的主要流程和系统接口函数与前面“IP 实验”基本相同。在下层接收接口函数 `Stud_fwd_deal()` 中（图 2.1 中接口函数 1），实现分组接收处理。主要功能是根据分组中目的 IPv4 地址结合对应的路由信息对分组进行处理。分组需要上交，则调用接口函数 `Fwd_LocalRcv()`（图 2.1 中接口函数 2）；需要丢弃，则调用函数 `Fwd_DiscardPkt()`（图 2.1 中函数 5）；需要转发，则进行转发操作。转发操作的实现要点包括，TTL 值减 1，然后重新计算头校验和，最后调用发送接口函数 `Fwd_SendtoLower()`（图 2.1 中接口函数 4）将分组发送出去。注意，接口函数 `Fwd_SendtoLower()` 比前面实验增加了一个参数 `pNxtHopAddr`，要求在调用时传入下一跳的 IPv4 地址，此地址是通过查找路由表得到的。

另外，本实验增加了一个路由表配置的接口（图 2.1 中函数 6），要求能够根据系统所给信息来设定本机路由表。实验中只需要简单地设置静态路由信息，以作为分组接收和发送处理的判断依据，而路由信息的动态获取和交互，在有关路由协议的实验（RIP 协议）中会重点涉及。

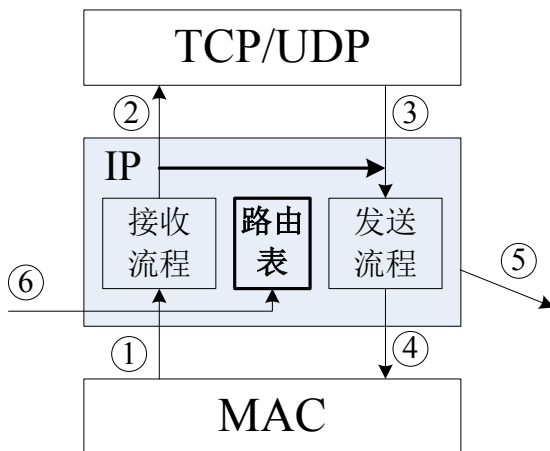


图 2.1 IPv4 分组转发实验接口示意图

与前面 IP 实验不同的是，在本实验中分组接收和发送过程中都需要引入路由表的查找步骤。路由器的主要任务是进行分组转发，它所接收的多数分组都是需要进行转发的，而不像主机协议栈中 IPv4 模块只接收发送给本机的分组；另外，路由器也要接收处理发送给本机的一些分组，如路由协议的分组（RIP 实验中会涉及到）、ICMP 分组等。如何确定对各种分组的处理操作类型，就需要根据分组的 IPV4 目的地址结合路由信息进行判断。

一般而言，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。在前面实验中，发送流程中没有查找路由表来确定下一跳地址的步骤，这项工作由系统来完成了，在本实验中则作为实验内容要求学生实现。需要进一步说明的是，在转发路径中，本路由器可能是路径上的最后一跳，可以直接转发给目的主机，此时下一跳的地址就是 IPv4 分组的目的地址；而非最后一跳的情况下，下一跳的地址是从对应的路由信息中获取的。因此，在路由表中转发类型要区分最后一跳和非最后一跳的情况。

路由表数据结构的设计是非常重要的，会极大地影响路由表的查找速度，进而影响路由器的分组转发性能。本实验中虽然不会涉及大量分组的处

理问题，但良好且高效的数据结构无疑会为后面的实验奠定良好的基础。链表结构是最简单的，但效率比较低；树型结构的查找效率会提高很多，但组织和维护有些复杂，可以作为提高的要求。具体数据结构的设计，可以在实践中进一步深入研究。

3.4.1 路由表维护

需要完成下列分组接收处理步骤：

- 1) `stud_Route_Init()`函数中，对路由表进行初始化。
- 2) `stud_route_add()`函数中，完成路由的增加。

3.4.2 转发处理流程

在 `stud_fwd_deal()`函数中，需要完成下列分组接收处理步骤：

- 1) 查找路由表。根据相应路由表项的类型来确定下一步操作，错误分组调用函数 `fwd_DiscardPkt()`进行丢弃，上交分组调用接口函数 `fwd_LocalRcv()`提交给上层协议继续处理，转发分组进行转发处理。注意，转发分组还要从路由表项中获取下一跳的 IPv4 地址。
- 2) 转发处理流程。对 IPv4 头部中的 TTL 字段减 1，重新计算校验和，然后调用下层接口 `fwd_SendtoLower()`进行发送处理。

3.4.3 实验接口函数

本小节列出了实验时会用到的各种接口函数，其中有一些函数是需要学生来完成的，有一些函数则是已经实现好了，供学生在实验时直接使用的。

表 2.1 列出了所有的接口函数及其简要说明，更详细的说明在后面描述。

函数名	说明	是否需要学生实现
<code>stud_fwd_deal</code>	系统处理收到的 IP 分组的函数，当接收到一个 IP 分组的时候，实验系统会调用该函数进行处理	是
<code>fwd_SendtoLower</code>	将封装完成的 IP 分组通过链路层发送出去的函数。	否
<code>fwd_LocalRcv</code>	将 IP 分组上交本机上层协议的函数，即当分组需要上交上层函数的时候调用本函数。	否

fwd_DiscardPkt	丢弃 IP 分组的函数。当需要丢弃一个 IP 分组的时候调用。	否
stud_route_add	向路由表添加路由的函数。系统将调用该函数向路由表添加一条 IPv4 路由。	是
stud_Route_Init	路由表初始化函数，系统初始化的时候将调用此函数对路由表进行初始化操作。	是
getIpv4Address	获取本机的 IPv4 地址，用来判断分组地址和本机地址是否相同	否

表 2.1 函数接口表

1) stud_fwd_deal()

int stud_fwd_deal(char * pBuffer, int length)

参数：

pBuffer: 指向接收到的 IPv4 分组头部

length: IPv4 分组的长度

返回值：

0 为成功，1 为失败；

说明：

本函数是 IPv4 协议接收流程的下层接口函数，实验系统从网络中接收到分组后会调用本函数。调用该函数之前已完成 IP 报文的合法性检查，因此学生在本函数中应该实现如下功能：

- a. 判定是否为本机接收的分组，如果是则调用 fwd_LocalRcv()；
- b. 按照最长匹配查找路由表获取下一跳，查找失败则调用 fwd_DiscardPkt()；
- c. 调用 fwd_SendtoLower() 完成报文发送；
- d. 转发过程中注意 TTL 的处理及校验和的变化；

2) fwd_LocalRcv()

void fwd_LocalRcv(char *pBuffer, int length)

参数:

pBuffer: 指向分组的 IP 头

length: 表示分组的长度

说明:

本函数是 IPv4 协议接收流程的上层接口函数, 在对 IPv4 的分组完成解析处理之后, 如果分组的目的地址是本机的地址, 则调用本函数将正确分组提交上层相应协议模块进一步处理。

3) fwd_SendtoLower()

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

参数:

pBuffer: 指向所要发送的 IPv4 分组头部

length: 分组长度 (包括分组头部)

nexthop: 转发时下一跳的地址。

说明:

本函数是发送流程的下层接口函数, 在 IPv4 协议模块完成发送封装工作后调用该接口函数进行后续发送处理。其中, 后续的发送处理过程包括分片处理、IPv4 地址到 MAC 地址的映射 (ARP 协议)、封装成 MAC 帧等工作, 这部分内容不需要学生完成, 由实验系统提供支持。

4) fwd_DiscardPkt()

void fwd_DiscardPkt(char * pBuffer, int type)

参数:

pBuffer: 指向被丢弃的 IPV4 分组头部

type: 表示错误类型, 包括 TTL 错误和找不到路由两种错误, 定义如下:

STUD_FORWARD_TEST_TTLERROR

STUD_FORWARD_TEST_NOROUTE

说明:

本函数是丢弃分组的函数, 在接收流程中检查到错误时调用此函数将分组丢弃。

5) stud_route_add()

void stud_route_add(stud_route_msg *proute)

参数:

`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;
    unsigned int  masklen;
    unsigned int  nexthop;
} stud_route_msg;
```

说明:

本函数为路由表配置接口, 系统在配置路由表时需要调用此接口。此函数功能为向路由表中增加一个新的表项, 将参数所传递的路由信息添加到路由表中。

6) `stud_Route_Init()`

`void stud_Route_Init()`

参数:

无

说明:

本函数将在系统启动的时候被调用, 学生可将初始化路由表的代码写在这里。

7) `getIpv4Address()`

`UINT32 getIpv4Address()`

说明:

本函数用于获取本机的 IPv4 地址, 学生调用该函数即可返回本机的 IPv4 地址, 可以用来判断 IPV4 分组是否为本机接收。

返回值::

本机 IPv4 地址

除了以上的函数以外, 学生可根据需要自己编写一些实验需要的函数和数据结构, 包括路由表的数据结构, 对路由表的搜索、初始化等操作函数。

4 IPv6 协议收发实验

4.1 实验目的

现有的互联网是在 IPv4 协议的基础上运行。IPv6 是下一版本的互联网协议，它的提出最初是因为随着互联网的迅速发展，IPv4 定义的有限地址空间将被耗尽，地址空间的不足必将影响互联网的进一步发展。为了扩大地址空间，拟通过 IPv6 重新定义地址空间。IPv4 采用 32 位地址长度，只有大约 43 亿个地址，估计在 2005~2010 年间将被分配完毕，而 IPv6 采用 128 位地址长度，几乎可以不受限制地提供地址。

本实验通过设计实现主机协议栈中的 IPv6 协议，让学生深入了解网络层协议的基本原理，学习 IPv6 协议基本的分组接收和发送流程。

4.2 实验要求

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 IPv6 分组收发模块。要求实现的主要功能包括：

- 1) IPv6 分组的基本接收处理；
- 2) IPv6 分组的封装发送；
- 3) 不要求实现 IPv6 协议中的选项和分片处理功能。

4.3 实验内容

- 1) 实现 IPv6 分组的基本接收处理功能

对于接收到的 IPv6 分组，检查目的地址是否为本地地址，并检查 IPv6 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 IPv6 分组的封装发送

根据上层协议所提供的参数，封装 IPv6 分组，调用系统提供的发送接口函数将分组发送出去。

4.4 实验帮助

在两个主机端系统通信的环境中，网络的拓扑可以简化为两台主机直接相连，中间的具体连接方式可以抽象为一条简单的链路，如图 4.1 所示。IPv6

分组收发实验就是要在实验系统客户端的开发平台上，实现 IPv6 分组的接收和发送功能。



图 4.1 实验环境网络拓扑结构

4.4.1 处理流程

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 `stud_ipv6_recv()`（图 4.2 中接口函数 1）。学生需要在这个函数中实现 IPv6 分组接收处理的功能。接收处理完成后，调用接口函数 `ipv6_SendtoUp()` 将需要上层协议进一步处理的信息提交给上层协议（图 4.2 中接口函数 2）；或者调用函数 `ipv6_DiscardPkt()` 丢弃有错误的分组并报告错误类型（图 4.2 中函数 5）。

在上层协议需要发送分组时，会调用发送接口函数 `stud_ipv6_Upsend()`（图 4.2 中接口函数 3）。学生需要在这个函数中实现 IPv6 分组封装发送的功能。根据所传参数完成 IPv6 分组的封装，之后调用接口函数 `ipv6_SendtoLower()` 把分组交给下层完成发送（图 4.2 中接口函数 4）。

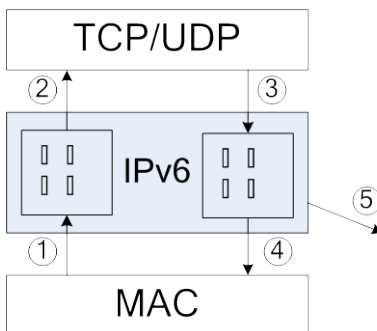


图 4.2 实验接口函数示意图

4.4.1.1 接收流程

在接口函数 `stud_ipv6_recv()` 中，需要完成下列处理步骤（仅供参考）：

- 1) 检查所接收到的 IPv6 分组头部的字段，包括版本号（Version）、有效载荷长度（Payload length）、跳数限制（Hop limit）字段。对于出错的分组调用 `ipv6_DiscardPkt()` 丢弃，并说明错误类型。
- 2) 检查 IPv6 分组是否该由本机接收。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 `ipv6_DiscardPkt()` 丢弃，并说明错误类型。
- 3) 提取得到上层协议类型，调用 `ipv6_SendtoUp()` 接口函数，交给系统进行后续接收处理。

4.4.1.2 发送流程

在接口函数 `stud_ipv6_Upsend()` 中，需要完成下列处理步骤(仅供参考)：

- 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 IPv6 协议标准填写 IPv6 分组头部各字段。注意，各字段内容都要转换成网络字节序。
- 3) 完成 IPv6 分组的封装后，调用 `ipv6_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。

4.4.2 IPv6 分组头部格式

IPv6 分组头部格式如图 4.3 所示，具体字段的定义请参考教材[1]和文献[2]。

Version	Traffic Class	Flow Label	
PayloadLen		Next Header	Hop Limit
128 bit source Address			
128 bit Destination Address			

图 4.3 IPv6 分组头部格式

- Version: 4 位协议版本号，为 6。
- Traffic Class: 传输类型，占 8 位。

- Flow Label: 流量标签: 占 20 位。
- Payload Length: 16 位无符号整数, IPv6 载荷, 也就是说跟在 IPv6 头部后面的部分, 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, 每经过一个节点减 1, 减为 0 后该报文被丢弃。
- Source Address: 源地址, 报文的产生者。
- Destination Address: 目的地址: 报文的接收者。

4.4.3 数据结构说明

系统提供了 IPv6 地址的结构定义如下:

```
typedef union
{
    char      bAddr[16];
    unsigned short wAddr[8];
    long dwAddr[4];
} ipv6_addr;
```

4.4.4 接口函数说明

4.4.4.1 需要实现的接口函数

1) 接收接口

int stud_ipv6_recv(char * pBuffer, unsigned short length)

参数:

pBuffer: 指向接收缓冲区的指针, 指向 IPv6 分组头部

length: IPv6 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

2) 发送接口

int stud_ipv6_Upsend(char* pData, unsigned short len, ipv6_addr srcAddr, ipv6_addr dstAddr, byte char hoplimit, char nextthead)

参数:

pData: IPv6 上层协议数据
 len: IPv6 上层协议数据长度
 srcAddr: 源 IPv6 地址
 dstAddr: 目的 IPv6 地址
 hoplimit: 跳数限制
 nexthead: 上层协议类型

返回值:

0: 成功发送 IP 分组
 -1: 发送分组失败

4.4.4.2 系统提供的接口函数

8) 丢弃分组

extern void ipv6_DiscardPkt(char * pBuffer, int type)

参数:

pBuffer: 指向被丢弃的分组

type: 分组被丢弃的原因, 有以下可能

STUD_IPV6_TEST_VERSION_ERROR	IP 版本号错
STUD_IPV6_TEST_DESTINATION_ERROR	目的地址错
STUD_IPV6_TEST_HOPLIMIT_ERROR	跳数限制错

9) 发送分组

extern void ipv6_SendtoLower(char *pBuffer, int length)

参数:

pBuffer: 指向待发送的 IPv6 分组

length: 待发送的 IPv6 分组长度

10) 上层接收

extern void ipv6_SendtoUp(char *pBuffer, int length)

参数:

pBuffer: 指向要上交的上层协议报文头

length: 上交报文长度

11) 获取本机 IPv6 地址

extern void getIpv6Address(ipv6_addr *paddr)

参数:

paddr: 指向本机 IPv6 地址结构的指针

5 IPv6 协议转发实验

5.1 实验目的

通过前面的实验，我们已经深入了解了 IPv6 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 IPv6 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 IPv6 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 IPv6 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 IPv6 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。IPv6 分组转发是路由器最为重要的功能。

本实验设计实现路由器中的 IPv6 协议，可以在原有 IPv6 分组收发实验的基础上，增加 IPv6 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端的。大家在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。

5.2 实验要求

在前面 IPv6 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 IPv6 分组，根据其目的 IPv6 地址查找本机的路由表，对该分组进行如下的几类操作：

- 1) 丢弃查不到路由的分组；
- 2) 向上层协议上交目的地址为本机地址的分组；
- 3) 根据路由查找结果，向相应接口转发其余的分组。

5.3 实验内容

实验内容主要包括：

- 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 IPv6 地址来确定分组处理行为（丢弃、上交或转发），转发情况下需获得下一跳的 IPv6 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) IPv6 分组的接收和发送。

对前面实验中所完成的代码进行修改，在路由器协议栈的 IPv6 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 IPv6 分组收发实验。

3) IPv6 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进一步处理。

5.4 实验帮助

分组转发是路由器最重要功能。分组转发的依据是路由信息，以此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。本实验要求按照路由器协议栈的 IPv6 协议功能进行设计实现，接收处理所有收到的分组（而不只是目的地之为本机地址的分组），并根据预先设定的路由信息，对分组进行转发、接收或丢弃。

实验的主要流程和系统接口函数与前面 IPv6 分组收发实验基本相同。在下层接收接口函数 `stud_ipv6_fwd_deal()` 中（图 5.1 中接口函数 1），实现分组接收处理。主要功能是根据分组中目的 IPv6 地址查找路由表，根据路由表查找结果进行后续处理。分组需要上交，则调用接口函数 `ipv6_fwd_LocalRcv()`（图 5.1 中接口函数 2）；需要丢弃，则调用函数 `ipv6_fwd_DiscardPkt()`（图 5.1 中函数 5）；需要转发，则进行转发操作。转发操作的实现要点包括，Hop Limit 值减 1，然后调用发送接口函数 `ipv6_fwd_SendtoLower()`（图 5.1 中接口函数 4）将分组发送出去。注意，接口函数 `ipv6_fwd_SendtoLower()` 比前面实验增加了一个参数 `nexthop`，要求在调用时传入下一跳的 IPv6 地址，此地址是通过查找路由表得到的。

另外，本实验增加了一个路由表配置的接口（图 5.1 中函数 6），要求能够根据系统所给信息来设定本机路由表。实验中只需要简单地设置静态路由信息，以作为分组接收和发送处理的判断依据。

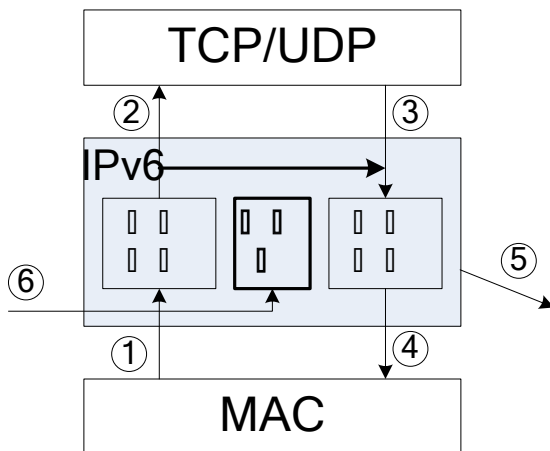


图 5.1 IPv6 分组转发实验接口示意图

与前面 IPv6 分组收发实验不同的是，在本实验中分组接收和发送过程中都需要引入路由表的查找步骤。路由器的主要任务是进行分组转发，它所接收的多数分组都是需要进行转发的，而不像主机协议栈中 IPv6 模块只接收发送给本机的分组。；另外，路由器也要接收处理发送给本机的一些分组，如路由协议的分组、ICMP 分组等。如何确定对各种分组的处理操作类型，就需要根据路由表进行判断。

一般而言，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 IPv6 分组后，要通过其目的地址匹配地址段来查找相应的路由信息，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 IPv6 地址。发送 IPv6 分组时，也要拿目的地址来查找路由表，得到下一跳的 IPv6 地址，然后调用发送接口函数进一步处理。在前面实验中，发送流程中没有查找路由表来确定下一跳地址的步骤，这项工作由系统来完成了，在本实验中则作为实验内容要求学生实现。需要进一步说明的是，在转发路径中，本路由器可能是路径上的最后一跳，可以直接转发给目的主机，此时下一跳的地址就是 IPv6 分组的目的地址；而非最后一跳的情况下，下一跳的地址是从对应的路由信息中获取的。因此，在路由表中转发类型要区分最后一跳和非最后一跳的情况。

路由表数据结构的设计是非常重要的，会极大地影响路由表的查找速度，进而影响路由器的分组转发性能。本实验中虽然不会涉及大量分组的处理问题，但良好且高效的数据结构无疑会为后面的实验奠定良好的基础。链

表结构是最简单的，但效率比较低；树型结构的查找效率会提高很多，但组织和维护有些复杂，可以作为提高的要求。具体数据结构的设计，可以在实践中进一步深入研究。

5.4.1 路由表维护

需要完成下列分组接收处理步骤：

- 1) `stud_ipv6_route_Init()` 函数中，对路由表进行初始化。
- 2) `stud_ipv6_route_add()` 函数中，完成路由的增加。

5.4.2 转发处理流程

在 `stud_ipv6_fwd_deal()` 函数中，需要完成下列分组接收处理步骤：

- 1) 查找路由表。根据相应路由表项的类型来确定下一步操作，错误分组调用函数 `ipv6_fwd_DiscardPkt()` 进行丢弃，上交分组调用接口函数 `ipv6_fwd_LocalRcv()` 提交给上层协议继续处理，转发分组进行转发处理。注意，转发分组还要从路由表项中获取下一跳的 IPv4 地址。
- 2) 转发处理流程。对 IPv6 头部中的 Hop Limit 字段减 1，然后调用下层接口 `ipv6_fwd_SendtoLower()` 进行发送处理。

5.4.3 实验接口函数

本小节列出了实验时会用到的各种接口函数，其中有一些函数是需要学生来完成的，有一些函数则是已经实现好了，供学生在实验时直接使用的。表 1 列出了所有的接口函数及其简要说明，更详细的说明在后面描述。

1) `stud_ipv6_fwd_deal()`

`int stud_ipv6_fwd_deal(char * pBuffer, int length)`

参数：

`pBuffer`：指向所及受到的 IPv6 分组头部

`length`：为 IPv6 分组的长度

返回值：

处理该包的返回值，0 为成功，1 为失败；

说明：

本函数是 IPv6 协议接收流程的下层接口函数，实验系统从网络中接收到分组后会调用本函数。调用该函数之前已完成 IPv6 报文的合法性检查，因此学生在本函数中应该实现如下功能：

- a. 判定是否为本机接收的分组，如果是则调用 `ipv6_fwd_LocalRcv()`；

- b. 按照最长匹配查找路由表获取下一跳，查找失败则调用 `ipv6_fwd_DiscardPkt()`;
- c. 调用 `ipv6_fwd_SendtoLower()` 完成报文发送;
- d. 转发过程中注意 Hop Limit 的处理;

2) `ipv6_fwd_LocalRcv()`

`void ipv6_fwd_LocalRcv(char *pBuffer, int length)`

参数:

`pBuffer`: 指向报文的 IPv6 头

`length`: 表示报文的长度

说明:

本函数是 IPv6 协议接收流程的上层接口函数，在对 IPv6 的分组完成解析处理之后，如果分组的目的地址是本机的地址，则调用本函数将正确分组提交上层相应协议模块进一步处理。

3) `ipv6_fwd_SendtoLower()`

`void ipv6_fwd_SendtoLower(char *pBuffer, int length, ipv6_addr *nexthop)`

参数:

`pBuffer`: 指向所要发送的 IPv6 分组的起始地址

`length`: 分组的整个长度（包括分组头部）

`nexthop`: 为转发处理时下一跳的地址。

说明:

本函数是发送流程的下层接口函数，在 IPv6 协议模块完成发送封装工作后调用该接口函数进行后续发送处理。其中，后续的发送处理过程包括分片处理、IPv6 地址到 MAC 地址的映射（ND 协议）、封装成 MAC 帧等工作，这部分内容不需要学生完成，由实验系统提供支持。

4) `ipv6_fwd_DiscardPkt()`

`void ipv6_fwd_DiscardPkt(char *pBuffer, int type)`

参数:

`pBuffer`: 指向被丢弃的报文

`type`: 表示错误类型，包括 TTL 错误和找不到路由两种错误，定义如下:

```
#define STUD_IPV6_FORWARD_TEST_HOPLIMIT_ERROR
```

```
#define STUD_IPV6_FORWARD_TEST_NOROUTE
```

说明：

本函数是丢弃分组的函数，在接收流程中检测到错误时调用此函数将分组丢弃。

5) stud_ipv6_route_add()

```
void stud_ipv6_route_add(stud_ipv6_route_msg *proute)
```

参数：

proute: 需要添加的路由，其数据结构 stud_ipv6_route_msg 的定义如下：

```
typedef    stud_route_msg
{
    ipv6_addr  dest;
    UINT32    masklen;
    ipv6_addr  nexthop;
} stud_ipv6_route_msg;
```

说明：

本函数为路由表配置接口，系统在配置路由表时需要调用此接口。此函数功能为向路由表中增加一个新的表项，将参数所传递的路由信息添加到路由表中。

6) stud_ipv6_Route_Init()

```
void stud_ipv6_Route_Init()
```

参数：

说明：

本函数用于对路由表进行初始化，由学生编写，在系统初始化的时候将调用该函数，对学生自己写的路由表数据结构进行初始化操作。

7) getIpv6Address()

```
void getIpv6Address(ipv6_addr* pAddr)
```

说明：

本函数用于获取本机的 IPv6 地址，学生调用该函数即可返回本机的 IPv6 地址。

参数：

pAddr: 返回的 IPv6 地址结构的指针

返回值:：

无

除了以上的函数以外，学生还需要自己编写一些实验需要的函数和数据结构，包括路由表的数据结构，对路由表的搜索、初始化等操作函数。

6 RIP 协议实验

6.1 实验目的

通过简单实现路由协议 RIP，深入理解计算机网络中的核心技术——路由技术，并了解计算机网络的路由转发原理。

6.2 实验要求

充分理解 RIP 协议，根据 RIP 协议的流程设计 RIP 协议的报文处理和超时处理函数。能够实现如下功能：

- 1) RIP 报文有效性检查
- 2) 处理 Request 报文
- 3) 处理 Response 报文
- 4) 路由表项超时删除
- 5) 路由表项定时发送

6.3 实验内容

- 1) 对客户端接收到的 RIP 报文进行有效性检查

对客户端接收到的 RIP 协议报文进行合法性检查，丢弃存在错误的报文并指出错误原因；

- 2) 处理 Request 报文

正确解析并处理 RIP 协议的 Request 报文，并能够根据报文的内容以及本地路由表组成相应的 Response 报文，回复给 Request 报文的发送者，并实现水平分割；

- 3) 处理 Response 报文

正确解析并处理 RIP 协议的 Response 报文，并根据报文中携带的路由信息更新本地路由表；

- 4) 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；

- 5) 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。

6.4 实验帮助

6.4.1 RIP 协议介绍

6.4.1.1 协议简述

RIP 协议采用的是距离-向量路由算法, 该算法早在 Internet 的前身 ARPANET 网络中就已经被广泛采用。在 70 年代中期, Xerox 公司根据它们对互联网的研究成果提出了一套被称为 XNS (Xerox Network System) 的网络协议软件。这套协议软件包含了 XNS RIP 协议, 该协议就是现在所使用的 RIP 协议的最早原型。80 年代加州大学伯克利分校在开发 Unix 系统的同时在 routed 程序中设计实现了 RIP 协议软件。routed 程序被绑定在 BSD Unix 系统中一起推出, 被广泛的应用于早期网络中的机器之间交换路由信息。尽管 RIP/routed 没有非常突出的优点, 但是由于 Unix 操作系统的普及, RIP/routed 也逐渐被推广出来, 为许多人所接受, 成为了中小型网络中最基本的路由协议/程序。

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 的报文结构进行了扩展, 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, 它在协议报文的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。

6. 4. 1. 2 RIPv2 协议的报文结构



图 7.1 RIPv2 的报文结构

RIPv2 的报文结构如图 7.1 所示。每个报文都包括一个报文命令字段、一个报文版本字段、一个路由域字段、一个地址类字段、一个路由标记字段以及一些路由信息项（一个 RIP 报文中最多允许 25 个路由信息项），其中每个字段后括号中的数字表示该字段所占的字节数。RIP 报文的最大长度为 $4+20*25=504$ 字节，加上 UDP 报头的 8 字节，一共是 512 字节。如果路由表的路由表项数目大于 25 时，那么就需要多个 RIP 报文来完成路由信息的传播过程。下面对报文字段进行逐一介绍：

- ◆ **命令字段：**表示 RIP 报文的类型，目前 RIP 只支持两种报文类型，分别是请求报文（request 1）和响应（response 2）报文。
- ◆ **版本字段：**表示 RIP 报文的版本信息，RIPv2 报文中此字段为 2。
- ◆ **路由域字段：**是一个选路守护程序的标识符，它指出了这个数据报的所有者。在一个 Unix 实现中，它可以是选路守护程序的进程号。该域允许管理者在单个路由器上运行多个 RIP 实例，每个实例在一个选路域内运行。
- ◆ **地址类字段：**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 地址字段：**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。

- ◆ **子网掩码字段：**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。
- ◆ **下一站 IP 地址字段：**下一驿站，可以对使用多路由协议的网络环境下的路由进行优化。
- ◆ **度量值字段：**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 发送的请求报文和响应报文都符合图 7.1 的报文结构格式，但是当需要发送请求对方路由器全部路由表信息的请求报文时，RIP 使用另一种报文结构，此报文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。

6. 4. 1. 3 RIP 协议的基本特点

协议规定，RIP 协议使用 UDP 的 520 端口进行路由信息的交互，交互的 RIP 信息报文主要是两种类型：请求（request）报文和响应（response）报文。请求报文用来向相邻运行 RIP 的路由器请求路由信息，响应报文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，在实际报文中，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 报文，然后 RIP 协议就进入了循环等待状态，等待外部 RIP 协议报文（包括请求报文和响应报文）的到来；而接收到 request 报文的路由器则应当发出包含它们路由表信息的 response 报文。

当发出请求的路由器接收到一个 response 报文后，它会逐一处理收到的路由表项内容。如果报文中的表项为新的路由表项，那么取出报文中路由表项的各个字段，并且取 response 报文的源地址作为下一跳的值，向路由表加入该表项。如果该报文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。

此外，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。

6.4.1.4 水平分割

水平分割是一种避免路由环的出现和加快路由汇聚的技术。由于路由器可能收到它自己发送的路由信息，而这种信息是无用的，水平分割技术不反向通告任何从终端收到的路由更新信息，而只通告那些不会由于计数到无穷而清除的路由。

6.4.2 实验拓扑

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。

6.4.3 需要实现的接口函数

1) RIP 报文处理函数

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

参数：

pBuffer：指向接收到的 RIP 报文内容的指针

bufferSize：接收到的 RIP 报文的长度

iNo：接收该报文的接口号

srcAdd：接收到的报文的源 IP 地址

返回值：

0

说明：

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

对 RIP 报文进行合法性检查，若报文存在错误，则调用 ip_DiscardPkt(char * pBuffer ,int type)函数，pBuffer：指向被丢弃分组的指针（从 IP 头部开始），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
```

根据报文的 `command` 域，判断报文类型。

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

对于 Response 报文，应该提取出该报文中携带的路由信息，对于本地路由表中已存在的项要判断该条路由信息的 `metric` 值，若为 16，则应置本地路由表中对应路由为无效，否则若更新表项的 `metric` 值小于路由表中相应表项 `metric` 值时就替代原来的表项。注意要将 `metric` 值加 1。对于本地路由表中不存在的项，则将 `metric` 值加 1 后将该路由项加入本地路由表，注意，若 `metric` 值加 1 后为 16 说明路由已经失效，则不用添加。

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

`mask`: 路由超时消息中路由的掩码

`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 报文。由于实现水平分割，报文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。

6.4.4 系统提供的全局变量

1) RIP 路由表

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

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;
};
```

6.4.5 系统提供的接口函数

1) 发送 RIP 报文函数

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

参数：

pData: 指向要发送的 RIP 报文内容的指针

len: 要发送的 RIP 报文的长度

dstPort: 要发送的 RIP 报文的目的端口

iNo: 发送该报文通过的接口的接口号

7 TCP 协议实验

7.1 实验目的

传输层是互联网协议栈的核心层次之一，它的任务是在源节点和目的节点间提供端到端的、高效的数据传输功能。TCP 协议是主要的传输层协议，它为两个任意处理速率的、使用不可靠 IP 连接的节点之间，提供了可靠的、具有流量控制和拥塞控制的、端到端的数据传输服务。TCP 协议不同于 IP 协议，它是有状态的，这也使其成为互联网协议栈中最复杂的协议之一。网络上多数的应用程序都是基于 TCP 协议的，如 HTTP、FTP 等。本实验的主要目的是学习和了解 TCP 协议的原理和设计实现的机制。

TCP 协议中的状态控制机制和拥塞控制算法是协议的核心部分。TCP 协议的复杂性主要源于它是一个有状态的协议，需要进行状态的维护和变迁。有限状态机可以很好的从逻辑上表示 TCP 协议的处理过程，理解和实现 TCP 协议状态机是本实验的重点内容。另外，由于在网络层不能保证分组顺序到达，因而在传输层要处理分组的乱序问题。只有在某个序号之前的所有分组都收到了，才能够将它们一起提交给应用层协议做进一步的处理。

拥塞控制算法对于 TCP 协议及整个网络的性能有着重要的影响。目前对于 TCP 协议研究的一个重要方向就是对于拥塞控制算法的改进。希望通过学习、实现和改进 TCP 协议的拥塞控制算法，增强大家对计算机网络进行深入研究的兴趣。

另外，TCP 协议还要向应用层提供编程接口，即网络编程中所普遍使用的 Socket 函数。通过实现这些接口函数，可以深入了解网络编程的原理，提高网络程序的设计和调试能力。

TCP 协议是非常复杂的，不可能在一个实验中完成所有的内容。出于工作量和实现复杂度的考虑，本实验对 TCP 协议进行适当的简化，只实现客户端角色的、“停一等”模式的 TCP 协议，能够正确的建立和拆除连接，接收和发送 TCP 报文，并向应用层提供客户端需要的 Socket 函数。

7.2 实验要求

实验要求主要包括：

- 1) 了解 TCP 协议的主要内容, 并针对客户端角色的、“停一等”模式的 TCP 协议, 完成对接收和发送流程的设计。
- 2) 实现 TCP 报文的接收流程, 重点是报文接收的有限状态机。
- 3) 实现 TCP 报文的发送流程, 完成 TCP 报文的封装处理。
- 4) 实现客户端 Socket 函数接口。

注意, 以下功能暂不做要求, 有能力的同学可顺序选做:

- 1) 滑动窗口。由于将 TCP 协议简化为“停一等”模式, 实际上发送窗口和接收窗口的大小都为 1, 因此也就不存在乱序接收的问题。发送流程也得到了很大的简化, 不再需要复杂的状态控制。可以进一步实现复杂的滑动窗口控制机制。
- 2) 拥塞控制。“停一等”模式的 TCP 协议, 其发送和接收窗口都为 1, 不存在拥塞控制的问题。可以在实现复杂的滑动窗口机制的基础上, 进一步设计实现拥塞控制算法。具体内容可参考教材[1]。
- 3) 往返时延估计。TCP 协议超时重传的间隔时间长短会影响到 TCP 协议甚至整个网络的性能, 如何确定重传时间间隔也是当前计算机网络领域的一个研究热点。现在主要采用的是一种动态自适应的算法, 根据网络性能的连续测量情况, 来不断地调整超时时间间隔。具体内容可参考教材[1]。

7.3 实验内容

实验内容主要包括:

- 1) 设计保存 TCP 连接相关信息的数据结构 (一般称为 TCB, Transmission Control Block)。
- 2) TCP 协议的接收处理。

学生需要实现 `stud_tcp_input()` 函数, 完成检查校验和、字节序转换功能 (对头部中的选项不做处理), 重点实现客户端角色的 TCP 报文接收的有限状态机。不采用捎带确认机制, 收到数据后马上回复确认, 以满足“停一等”模式的需求。

- 3) TCP 协议的封装发送。

学生需要实现 `stud_tcp_output()` 函数, 完成简单的 TCP 协议的封装发送功能。为保证可靠传输, 要在收到对上一个报文的确认后才能够继续发送。

4) TCP 协议提供的 Socket 函数接口

实现与客户端角色的 TCP 协议相关的 5 个 Socket 接口函数，
stud_tcp_socket()、**stud_tcp_connect()**、**stud_tcp_recv()**、
stud_tcp_send()和 **stud_tcp_close()**，将接口函数实现的内容与发送和接收流程有机地结合起来。

实验内容（1）、（2）和（3）完成后，其正确性用测试例 1——*针对分组交互的测试*来验证；实验内容（4）的正确性用测试例 2——*socket API 测试*来验证。

7.4 实验帮助

从设计实现的角度出发，TCP 协议主要考虑以下几个因素：

- 1) 状态控制
- 2) 滑动窗口机制
- 3) 拥塞控制算法
- 4) 性能问题（RTT 估计）
- 5) Socket 接口

将这些内容完全实现具有很大的难度，因此我们对其进行简化。本实验中要求实现一个客户端角色的 TCP 协议，采用“停一等”模式，即发送窗口和接收窗口的大小都为 1，并向上层提供客户端所必需的 Socket 接口函数。

客户端角色的 TCP 协议只能够主动发起建立连接请求，而不能监听端口等待其它主机的连接。因而相比于标准 TCP 协议的状态控制，客户端角色的 TCP 协议的状态机得到简化，如图 3.1 所示。

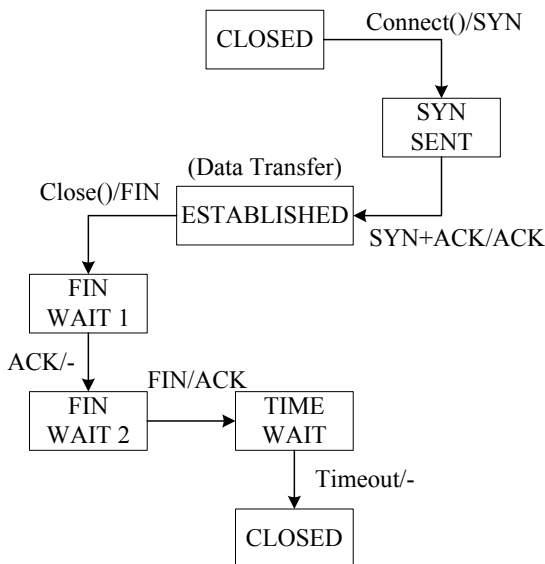


图 3.1 客户端角色的 TCP 协议状态机

从 **CLOSED** 状态开始，调用 Socket 函数 `stud_tcp_connect()` 发送建立连接请求（SYN 报文）后转入 **SYN-SENT** 状态，等待服务器端的应答和建立连接请求；收到服务器端的 SYN+ACK 后，以 ACK 应答，完成“三次握手”的过程，转为 **ESTABLISHED** 状态。在 **ESTABLISHED** 状态，客户端与服务器端可以通过函数 `stud_tcp_recv()` 和 `stud_tcp_send()` 进行数据交互。完成数据传输后，客户端通过 `stud_tcp_close()` 函数发送关闭连接请求（FIN 报文），转入 **FIN-WAIT1** 状态；收到应答 ACK 后进入 **FIN-WAIT2** 状态，此时状态为半关闭，等待服务器端关闭连接。收到服务器端的 FIN 报文后，需要发送确认 ACK，状态改变为 **TIME-WAIT**；等待一段时间后，转为初始的 **CLOSED** 状态，连接完全断开。

上述流程是一个客户端 TCP 建立和拆除连接的最简单的正常流程，然而实际的 TCP 协议要处理各种各样的异常情况，这也是 TCP 状态控制非常复杂的一个原因。本实验中对此不做复杂处理，对接收到的异常报文（如报文类型不符、序号错误等情况）调用系统所提供的函数 `tcp_DiscardPkt()` 丢弃即可。

“停一等”模式的 TCP 协议，其发送窗口和接收窗口大小都为 1。接收到的报文中，只有序列号正确的报文才会被处理，并根据报文内容和长度对接

收窗口进行滑动。发送一个报文后就不能继续发送，而要等待对方对此报文的确认，收到确认后才能继续发送。如果一段时间后没有收到确认，则需要重传（本实验暂不涉及重传机制）。

实现一个完整的 TCP 协议，还需要向应用层提供 Socket 接口函数。作为客户端角色的 TCP 协议，必须实现的 Socket 接口函数包括：

stud_tcp_socket()、**stud_tcp_connect()**、**stud_tcp_recv()**、**stud_tcp_send()** 和 **stud_tcp_close()**。实现 Socket 接口函数，主要工作是将 Socket 接口函数和 TCP 报文的接收和发送流程结合起来，根据应用层需求发送或接受特定类型的报文或读写数据。实验的接口如下图 3.2 所示。

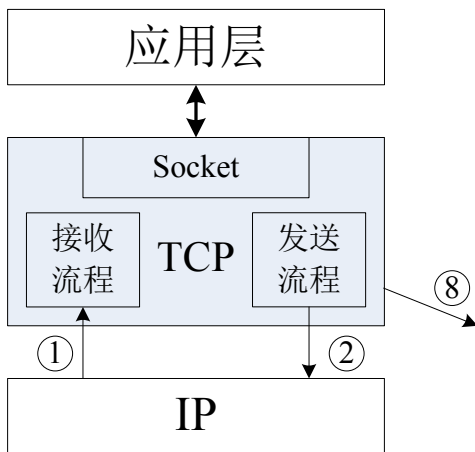


图 3.2 TCP 实验接口示意图

下面详细说明设计实现时所涉及的要点，包括数据结构的设计、报文的接收和发送处理流程、实验模块的接口函数说明等。

7.4.1 TCP 报文头部格式

TCP 报文头部格式如图 3.3 所示，具体字段的定义请参考教材[1]及文献[2]。

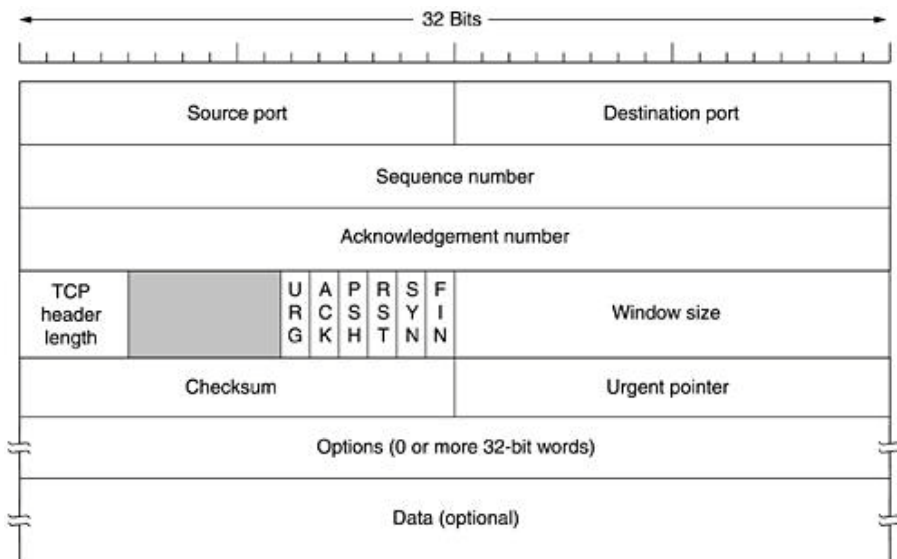


图 3.3 TCP 报文头部格式

7.4.2 数据结构

一般情况下，TCP 协议通过一个数据结构来控制每个 TCP 连接的发送和接收动作，该数据结构被称为传输控制块（Transmission Control Block，TCB）。TCP 为每个活动的连接维护一个 TCB 结构，所有 TCP 连接对应的 TCB 结构可以组织成一个 TCB 的链表结构进行管理。TCB 中包含了有关 TCP 连接的所有信息，包括两端的 IP 地址和端口号、连接状态信息、发送和接收窗口信息等。

TCB 的数据结构由学生来设计完成，本实验指导书中并不做具体规定，学生可根据设计实现的需要来具体定义 TCB 结构。

7.4.3 接收和发送处理

在 `stud_tcp_input()` 函数中，需要完成下列分组接收处理步骤：

- 1) 检查校验和。关于 TCP 头部中校验和的计算和检查请参考教材[1]和参考文献[4]的相关部分。
- 2) 字节序转换。
- 3) 检查序列号。如果序列号不正确，则调用 `tcp_DiscardPkt`。
- 4) 将报文交由输入有限状态机处理。
- 5) 有限状态机对报文进行处理，转换状态。

- 6) 根据当前的状态并调用 `stud_tcp_output` 函数完成 tcp 建连、数据传递时返回 ACK、tcp 断连等工作

在 `stud_tcp_output()` 函数中, 分组发送流程的实现要点:

- 1) 判断需要发送的报文类型, 并针对特定类型做相应的处理。
- 2) 判断是否可以发送 (发送窗口不为 0)。
- 3) 构造 TCP 数据报文并发送。填写 TCP 报文各字段的内容和数据, 转换字节序, 计算校验和, 然后调用发送流程的下层接口函数 `sendIpPkt()` 发送。

7. 4. 4 Socket 接口各函数处理

在 `stud_tcp_socket()` 函数中, 分组发送流程的实现要点:

- 1) 在此函数中要先创建新的 TCB 结构, 并对成员变量进行初始化。
- 2) 为每个 TCB 结构分配唯一的套接口描述符。

在 `stud_tcp_connect()` 函数中, 分组发送流程的实现要点:

- 1) 设定目的 IPv4 地址和端口, 源 IPv4 地址和端口。
- 2) 初始化 TCB 结构中的相关变量。
- 3) 设定 TCB 中的输入状态为 **SYN-SENT**, 及其它相关变量, 准备发送 SYN 报文。
- 4) 调用发送流程的下层接口函数 `stud_tcp_output()` 发送 SYN 报文 (发送类型为 `PACKET_TYPE_SYN`)。
- 5) 等待“三次握手”完成后返回, 建立连接成功; 或者出错返回。

`stud_tcp_send()` 函数的实现要点:

- 1) 判断是否处于 **ESTABLISHED** 状态。
- 2) 将应用层协议的数据拷贝到 TCB 的输入缓冲区。
- 3) 调用 `stud_tcp_output()` 发送 TCP 的数据报文 (发送类型为 `PACKET_TYPE_DATA`)。

`stud_tcp_recv()` 函数的实现要点:

- 1) 判断是否处于 **ESTABLISHED** 状态。
- 2) 从 TCB 的输入缓冲区读出数据。
- 3) 将数据交给应用层协议。

在 `stud_tcp_close()` 函数中，分组发送流程的实现要点：

- 1) 在正常情况下 (**ESTABLISHED** 状态)，进行相应状态转换；非正常情况下 (**SYN-SENT** 状态)，直接删除 TCB 结构后退出。
- 2) 调用发送流程下层接口函数 `stud_tcp_output()` 发送 FIN 报文（发送类型为 `PACKET_TYPE_FIN`）。
- 3) 等待回应的 ACK 报文，收到后成功返回；或者出错返回。

7.4.5 系统提供的接口函数

1) TCP 处理中由于某种原因丢弃报文

`void tcp_DiscardPkt(char * pBuffer, int type);`

参数：

`char * pBuffer`：指向被丢弃的报文；

`int type`：报文丢弃的原因，有以下三种可能：

`STUD_TCP_TEST_SEQNO_ERROR`

序列号错误

`STUD_TCP_TEST_SRCPORT_ERROR`

源端口错误

`STUD_TCP_TEST_DSTPORT_ERROR`

目的端口错误

说明：

本函数给出了丢弃报文的函数接口

返回值：

无

2) IP 报文发送函数：

`void tcp_sendIpPkt(unsigned char* pData, uint16 len, unsigned int srcAddr, unsigned int dstAddr, uint8 ttl)`

参数：

`pData`：IP 上层协议数据；

`len`：IP 上层协议数据长度；

`srcAddr`：源 IP 地址；

`dstAddr`：目的 IP 地址；

`ttl`：跳极限；

说明：

本函数提供了发送 IP 报文的接口

返回值：

无

下列函数仅用于 **socket** 接口函数：

3) IP 数据报文主动接收：

`int waitIpPacket(char *pBuffer, int timeout)`

参数：

`char *pBuffer`：接收缓冲区的指针

`int timeout`：等待时间

返回值：

如果正确接收则返回接收到的数据长度，否则返回 -1

说明：

本函数用于学生代码中主动接收 IP 分组，如果在设定时间内正确接收到分组，则将该分组内容复制到 `pBuffer` 中，否则返回 -1

4) 客户端获得本机 IPv4 地址：

`UINT32 getIpv4Address();`

5) 客户端获得服务器 IPv4 地址：

`UINT32 getServerIpv4Address();`

6) 全局变量

`int gSrcPort = 2005;`

`int gDstPort = 2006;`

`int gSeqNum = 0;`

`int gAckNum = 0;`

7. 4. 6 需要完成的接口函数

7. 4. 6. 1 接收和发送函数

一、针对分组交互的测试

1) TCP 分组接收函数：

`int stud_tcp_input(char *pBuff, unsigned shortlen, unsigned int srcAddr, unsigned int dstAddr)`

参数:

char *pBuff: 指向接收缓冲区的指针, 从 TCP 头开始

unsigned short len: 缓冲区数据长度

unsigned int srcAddr: 源 IP 地址

unsigned int dstAddr: 目的 IP 地址

返回值:

如果成功则返回 0, 否则返回-1

说明:

所有接收到的 TCP 报文都将调用本函数传递给学生代码, 本函数中学生需要维护一个状态机, 并根据状态机得变迁调用 stud_TCP_send 向服务器发送对应的报文, 如果出现异常, 则需要调用 tcp_sendReport 函数向服务器报告处理结果。

2) TCP 分组发送函数:

void stud_tcp_output(char *pData, unsigned short len, unsigned char flag, unsigned short srcPort, unsigned short dstPort, unsigned int srcAddr, unsigned int dstAddr)

参数:

char *pData: 数据指针

unsigned short len: 数据长度

unsigned char flag: 分组类型

unsigned short srcPort: 源端口

unsigned short dstPort: 目的端口

unsigned int srcAddr: 源 IP 地址

unsigned int dstAddr: 目的 IP 地址

其中, 分组类型 flag 为以下可能:

PACKET_TYPE_DATA

数据

PACKET_TYPE_SYN

SYN 标志位开

PACKET_TYPE_SYN_ACK

SYN、ACK 标志位开

PACKET_TYPE_ACK

ACK 标志位开

PACKET_TYPE_FIN

FIN 标志位开

PACKET_TYPE_FIN_ACK

FIN、ACK 标志位开

输出:

无

说明:

学生需要在此函数中自行申请一定的空间, 并封装 TCP 头和相关的数
据, 此函数可以由接收函数调用, 也可以直接由解析器调用, 此函数中将调
用 tcp_sendIpPkt 完成分组发送。

7. 4. 6. 2 Socket 接口函数

1) 获得 socket 描述符

int stud_tcp_socket(int domain, int type, int protocol)

参数:

int domain: 套接字标志符, 缺省为 INET

int type: 类型, 缺省为 SOCK_STREAM

int protocol: 协议, 缺省为 IPPROTO_TCP

返回值:

如果正确建连则返回 socket 值, 否则返回-1

说明:

返回以后在系统调用中可能用到的 socket 描述符, 或者在错误的时候
返回-1

2) TCP 建立连接函数

int stud_tcp_connect(int sockfd, struct sockaddr_in* addr, int addrlen)

参数:

int sockfd: 套接字标志符

struct sockaddr_in* addr: socket 地址结构指针

int addrlen socket: 地址结构的大小

返回值:

如果正确发送则返回 0, 否则返回-1

说明:

在本函数中要求发送 SYN 报文, 并调用 waitIpPacket 函数获得
SYN_ACK 报文, 并发送 ACK 报文, 直至建立 tcp 连接。

3) TCP 报文发送函数

int stud_tcp_send(int sockfd, const unsigned char* pData, unsigned short datalen,
int flags)

参数:

int sockfd: 套接字标志符
 const unsigned char* pData: 数据缓冲区指针
 unsigned short datalen: 数据长度
 int flags: 标志

返回值:

如果正确接收则返回 0，否则返回-1

说明:

本函数向服务器发送数据“this is a tcp test”，在本函数内要调用 waitIpPacket 函数获得 ACK。

4) TCP 报文接收函数

int stud_tcp_recv(int sockfd, const unsigned char* pData, unsigned short datalen, int flags)

参数:

int sockfd: 套接字标志符
 const uint8* pData: 数据缓冲区指针
 uint16 dataLen: 数据长度
 int flags: 标志

返回值:

如果正确接收则返回 0，否则返回-1

说明:

本函数接收从服务器发送的数据，并调用在本函数内要调用 sendIpPkt 函数发送 ACK。

5) TCP 关闭连接函数

int stud_tcp_close(int sockfd)

参数:

int sockfd: 连接描述符

返回值:

如果正常关闭则返回 0，否则返回-1

说明:

在本函数中要求发送 FIN 报文，并调用 waitIpPacket 函数获得 FIN_ACK 报文，并发送 ACK 报文，直至关闭 tcp 连接

8 协议状态机实验

8.1 实验目的

通过协议状态机实验，使学生加深对协议状态机描述的认识，同时对实现协议状态机有一个初步的认识；实验状态机取材于 BGP 路由协议，通过这个实验，同学们也可以增进对于路由协议的理解。

8.2 实验要求

- 1) 协议状态变迁；

8.3 实验内容

- 1) 协议状态变迁；

根据系统的各种输入事件，进行 BGP 状态的变迁，并根据 BGP 协议在适当情况下进行相应的处理。

8.4 实验帮助

8.4.1 BGP 协议介绍

8.4.1.1 协议简述

BGP 是一种自治系统间的动态路由发现协议，它的基本功能是在自治系统间自动交换无环路的路由信息。与 OSPF 和 RIP 等在自治区域内部运行的协议对应，BGP 是一类 EGP(Edge Gateway Protocol)协议，而 OSPF 和 RIP 等为 IGP(Interior Gateway Protocol)协议。

BGP 是在 EGP 应用的基础上发展起来的。EGP 在此以前已经作为自治区域间的路由发现协议，广泛应用于 NFSNET 等主干网络上。但是，EGP 被路由环路问题所困扰。BGP 通过在路由信息中增加自治区域(AS)路径的属性，来构造自治区域的拓扑图，从而消除路由环路并实施用户配置的策略。同时，随着 INTERNET 的飞速发展，路由表的体积也迅速增加，自治区域间路由信息的交换量越来越大，都影响了网络的性能。BGP 支持无类型的区域间路由 CIDR(Classless Interdomain Routing)，可以有效的减少日益增大的路由表。

BGP 运行时时刻分别与本自治区域外和区域内的 BGP 伙伴建立连接(使用 Socket)。与区域内伙伴的连接称为 IBGP(Internal BGP)连接，与自治区域外

的 BGP 伙伴的连接称为 EBGP(External BGP)连接。本地的 BGP 协议对 IBGP 和 EBGP 伙伴使用不同的机制处理。

8.4.1.2 BGP 协议的报文类型

BGP 有 4 种类型的消息。分别为 OPEN, UPDATE, KEEPALIVE 和 NOTIFY。它们有相同的消息头。

1. 消息头结构:



Marker (16bytes)

Length

Type

Marker : (16 字节) 鉴权信息

Length : (2 字节) 消息的长度

Type : (1 字节) 消息的类型

0 : OPEN

1 : UPDATE

2 : NOTIFICATION

3 : KEEPALIVE

2. OPEN 消息结构:

消息头加如下结构：



Version

My Autonomous System

Hold Time

BGP Identifier

OptParmLen

Optional Parameters (n bytes)

Version : (1 字节) 发端 BGP 版本号

My Autonomous System : (2 字节无符号整数) 本地 AS 号

Hold Time : (2 字节无符号整数) 发端建议的保持时间

BGP Identifier : (4 字节) 发端的路由器标识符

OptParmLen : (1 字节) 可选的参数的长度

Optional Parameters : (变长) 可选的参数

3. KEEPALIVE 消息结构

KEEPALIVE 消息只有一个消息头。

4. NOTIFY 消息结构

消息头加如下结构:

1	2	3	4
Error code	Errsubcode	Data	

Errorcode: (1 字节) 错误代码

错误代码

1

2

3

4

5

6

错误类型

消息头错

OPEN 消息错

UPDATE 消息错

保持时间超时

状态机错

退出

Errsubcode : (1 字节) 辅助错误代码, 略。

Data : (变长) 依赖于不同的错误代码和辅助错误代码。用于诊断错误原因。

5. UPDATE 消息结构

消息头加如下结构:

Unfeasible Routes Len : (2 字节无符号整数) 不可达路由长度

Withdrawn Routes : (变长) 退出路由

Path Attribute Len : (2 字节无符号整数) 路径属性长

Path Attributes : (变长) 路径属性(以下详细说明)

Network Layer Reachability Information : (变长) 网络可达信息(信宿)

其中退出路由和信宿地址的表示方法为一 的二元组。length 一个字节, 指示地址前缀的长度。prefix 为地址前缀, 长度 1 至 4 字节。

8. 4. 1. 3 BGP 伙伴的有限状态机 (FSM)

BGP 有限状态机有 6 种状态:

1—Idle

2—Connect

- 3—Active
- 4—OpenSent
- 5—OpenConfirm
- 6—Established

BGP 事件:

- 1—Start
- 2—ManualStop
- 3—BGP Transport connection open
- 4—BGP Transport connection closed
- 5—BGP Transport connection open failed
- 6—BGP Transport fatal error
- 7—ConnectRetry timer expired
- 8—Hold Timer expired
- 9—KeepAlive timer expired
- 10—Receive OPEN message
- 11—Receive KEEPALIVE message
- 12—Receive UPDATE message
- 13—Receive NOTIFICATION message

● Idle 状态

BGP 总是从 IDLE 状态开始, 此时拒绝所有外来连接。当一个 Start 事件 (IE1) 发生 (如操作者手动配置 BGP 进程或复位现有的 BGP 进程等), BGP 进程将初始化所有 BGP 资源, 起定时器, 并初始化与邻居间的 TCP 连接, 监听来自邻居的 TCP 初始化, 并将自己的状态置为 Connect。

● Connect 状态

在此状态, BGP 进程等待 TCP 建连完成。若 TCP 建连成功, BGP 进程将清除 ConnectRetry 定时器, 完成初始化, 并向邻居发送 OPEN 报文, 且将自己状态置为 OpenSent 状态。若 TCP 建连不成功, BGP 进程将继续监听来自邻居的连接, 复位 ConnectRetry 定时器, 并将自己状态转为 Active。

若在此状态 ConnectRetry 定时器超时, 此定时器将被复位, 进程将再次试图与邻居建立 TCP 连接, 状态仍保持在 Connect 状态。其他事件 (除了 IE1) 都将使 BGP 状态回到 Idle 状态。

● Active 状态

在此状态, BGP 进程试图初始化与邻居的 TCP 连接, 若 TCP 建连成功, BGP 进程清除 ConnectRetry 定时器, 完成初始化, 并向邻居发送 OPEN 报文, 且将自己状态置为 OpenSent 状态。

若在此状态 ConnectRetry 定时器超时, 进程将回到 Connect 状态, 并复位 ConnectRetry 定时器, 且初始化 TCP 连接, 监听 TCP 连接。

其他事件 (除了 IE1) 都将使 BGP 状态回到 Idle 状态。

- **OpenSent 状态**

在此状态, Open 报文已经被发送, BGP 进程将等待来自邻居的 OPEN 报文。当收到来自邻居的 Open 报文时, BGP 进程将检查报文的所有的域, 若存在错误, 则发送 Notification 报文, 并将自己状态置为 Idle。

若 Open 报文中没有错误, 则发送一个 KeepAlive 报文, 并置 KeepAlive 定时器。并将自己状态置为 OpenConfirm。

若发生 TCP 断连, 本地进程将关闭 BGP 连接, 重置 ConnectRetry 定时器, 并开始监听新的连接。自己的状态被置为 Active。其他事件 (除了 IE1) 都将使 BGP 状态回到 Idle 状态。

- **OpenConfirm 状态**

在此状态, BGP 进程等待 KeepAlive 报文或 Notification 报文。若收到 KeepAlive 报文, 则将自己状态转到 Established 状态。

若在收到 KeepAlive 报文之前 KeepAlive 计时器超时, 则发送一个 KeepAlive 消息, Hold 定时器超时, 则发送一个 Notification 报文, 其中错误代码为 Hold Timer Expired, 并将状态转为 Idle。

若收到 Notification 报文, 则转换状态到 Idle 态。

若 Hold 定时器超时, 或错误发生, 或 Stop 事件发生, 将发送一个 Notification 报文到邻居, 关闭 BGP 连接, 并转换状态到 Idle 态。

- **Established 状态**

在此状态, BGP 对等体的连接被完全建立, 此时可以交互各种 BGP 报文, 若收到 Notification 报文, 则转换状态到 Idle 态。

若 Hold 定时器超时, 则发送一个 Notification 报文, 其中错误代码为 “Hold Timer Expired”, 并将自己状态转为 Idle 态。

若 Keepalive 定时器超时, 则发送一个 Keepalive 报文, 并复位 Keepalive 定时器。

若收到 BGP Stop 事件, 则发送 Notification 报文, 其中错误代码为 Cease, 并将自己状态转为 Idle。

其他事件(除了 IE1)都将发送 Notification 报文, 其中错误代码为 Finite State Machine Error, 并将 BGP 状态回到 Idle 状态。

状态机如图 8.1 所示:

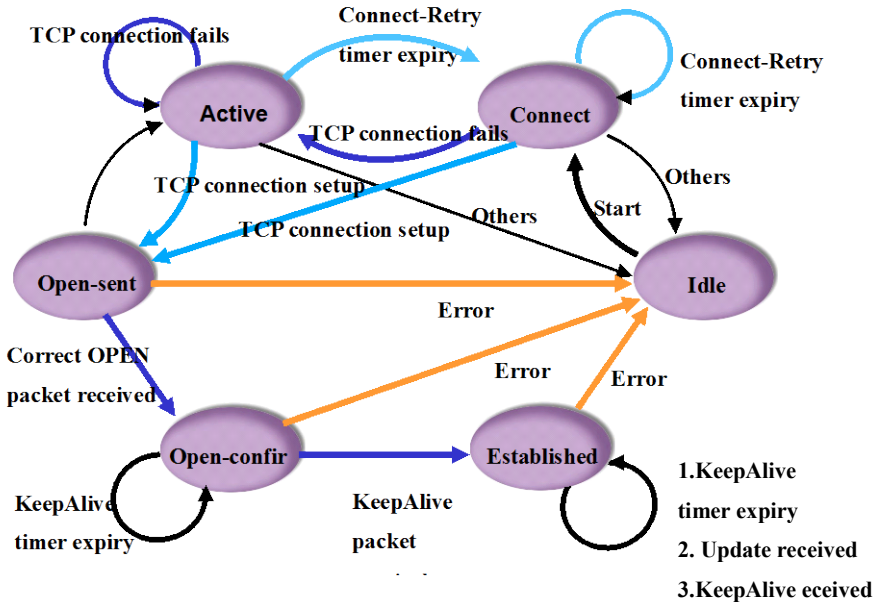


图 8.1 BGP 有限状态机

一个典型的过程为: Idle(启动消息)→Connect(TCP 连接成功, 发 OPEN)→OpenSent(收到 OPEN 消息, 协商 成功)→OpenConfirm(收到 KEEPALIVE 消息)→Established(TCP 连接关闭, 有错误, 或处理 UPDATE 消息失败, 或收到 NOTIFICATION 消息)→Idle

8.4.2 需要实现的接口函数

1) 收到 Open 报文事件处理函数

BOOLEAN stud_bgp_FsmEventOpen (BgpPeer *pPeer, BYTE * pBuf, unsigned int len)

参数:

pPeer: 指向当前对等体的指针

pBuf: 指向接收到的 open 报文

len: open 报文长度

返回值:

0: 成功处理 Open 报文

-1: 处理过程出现错误

说明:

当系统收到 BGP Open 报文时, 会调用此函数, 学生编写此函数, 应该实现如下功能:

对 Open 报文进行检查, 根据当前状态, 调整 BGP 状态机的状态。并在需要的时候发送 KeepAlive 报文。

2) 收到 Keepalive 报文事件处理函数

BOOLEAN stud_bgp_FsmEventKeepAlive (BgpPeer *pPeer, BYTE *pBuf , unsigned int len)

参数:

pPeer: 指向当前对等体的指针

pBuf: 指向接收到的 keepalive 报文

len: keepalive 报文长度

返回值:

0: 成功处理 Keepalive 报文

-1: 处理过程出现错误

说明:

当系统收到 BGP Keepalive 报文时, 会调用此函数, 学生编写此函数, 应该实现对 BGP 状态进行相应的改变。

3) 收到 Notification 报文事件处理函数

BOOLEAN stud_bgp_FsmEventNotification (BgpPeer *pPeer, BYTE *pBuf, unsigned int len)

参数:

pPeer: 指向当前对等体的指针

pBuf: 指向接收到的 Notification 报文

len: Notification 报文长度

返回值:

0: 成功处理 Notification 报文

-1: 处理过程出现错误

说明:

当系统收到 BGP Notification 报文时, 会调用此函数, 学生编写此函数, 应该实现对 BGP 状态进行相应的改变。

4) 收到 Update 报文事件处理函数

BOOLEAN stud_bgp_FsmEventUpdate (BgpPeer *pPeer, BYTE *pBuf, unsigned int len)

参数:

pPeer: 指向当前对等体的指针

pBuf: 指向接收到的 Update 报文

len: Update 报文长度

返回值:

0: 成功处理 Update 报文

-1: 处理过程出现错误

说明:

当系统收到 BGP Update 报文时, 会调用此函数, 学生编写此函数, 应该实现对 BGP 状态进行相应的改变。

5) TCP 连接异常事件处理函数

void stud_bgp_FsmEventTcpException (BgpPeer *pPeer, BYTE msgType)

参数:

pPeer: 指向当前对等体的指针

msgType: TCP 异常类型。可能的取值如下:

BGP_TCP_CLOSE /*tcp dose*/

BGP_TCP_FATAL_ERROR /*BGP Transport fatal error*/

BGP_TCP_RETRANSMISSION_TIMEOUT/*retransmission

timeout*/

返回值:

无

说明:

当 TCP 出现异常时, 系统会调用此函数。学生编写此函数, 根据 BGP

状态机的需求，进行相应操作，以及状态的改变。

6) 定时器超时事件处理函数

```
void stud_bgp_FsmEventTimerProcess ( BgpPeer * pPeer, BYTE msgType)
```

参数:

pPeer: 指向当前对等体的指针

msgType: 定时器消息类型，可能的取值如下:

BGP_CONNECTRETRY_TIMEOUT

BGP_HOLD_TIMEOUT

BGP_KEEPALIVE_TIMEOUT

返回值:

无

说明:

当定时器超时时，系统会调用此函数，根据 msgType 传入超时定时器的类型。学生应根据协议需求，进行相应操作，以及状态的改变。

7) Bgp 开始事件处理函数

```
void stud_bgp_FsmEventStart ( BgpPeer *pPeer )
```

参数:

pPeer: 指向当前对等体的指针

返回值:

无

说明:

当发生 BGP Start 事件 (IE1) 时，系统调用此函数，学生根据协议要求编写代码，进行 BGP 状态的改变，并在适当的条件下调用试图建连函数初始化 TCP 连接。

8) Bgp 结束事件处理函数

```
void stud_bgp_FsmEventStop ( BgpPeer *pPeer )
```

参数:

pPeer: 指向当前对等体的指针

返回值:

无

说明:

当发生 BGP Stop 事件 (IE2) 时，系统调用此函数，学生根据协议要求

编写代码，进行 BGP 状态的改变。

9) 收到连接结果事件处理函数

```
void stud_bgp_FsmEventConnect (BgpPeer *pPeer )
```

参数:

pPeer: 指向当前对等体的指针

返回值:

无

说明:

BGP 的 TCP 建连成功时，系统将调用此函数。学生需要在此函数中实现 BGP 状态的改变，以及在必要的时候构造 BGP Open 报文，并通过 bgp_FsmSendTcpData 函数发送出去。

8.4.3 数据结构定义

系统定义了 BGP 对等体结构定义，如下:

```
struct BgpPeer
{
    DWORD bgp_dwMyRouterID;    // 本路由器的路由器 ID;
    WORD bgp_wMyAS ; // 向对方建立时，所用的 AS。联盟相关
    DWORD bgp_dwCfgHoldtime; //设置的 holdtime 时间值
    BYTE bgp_byState;          //协议状态机
    BYTE bgp_bAuth;            // 是否有认证信息
}
```

8.4.4 系统提供的接口函数

1) 试图建连函数

```
void bgp_FsmTryToConnectPeer ( )
```

参数:

无

返回值:

无

说明:

学生根据 BGP 协议要求，在需要试图建连时，调用此函数开始 TCP 连接的初始化。

2) TCP 报文发送函数

void bgp_FsmSendTcpData (char *pBuf, DWORD dwLen)

参数:

pBuf: 传送的报文内容

dwLen: 发送的报文长度

返回值:

无

说明:

学生根据 BGP 协议要求，在需要发送 BGP 报文时调用此函数来发送 BGP 报文。

9 IPSEC 协议实验

9.1 实验目的

通过 IPSEC 协议实验,使学生 加深对 IPSEC 协议的认识,同时对 IPSEC 协议实现的工作模式、AH 协议和 ESP 协议有一个初步的认识,通过这个实验,同学们也可以增进对于网络安全协议的理解。

9.2 实验要求

- 1) 传输模式报文处理 (AH 协议)
- 2) 隧道模式报文处理 (ESP 协议)
- 3) 混合模式报文处理 (AH 协议+ESP 协议)
- 4) 3des 加密算法实现 (ESP 协议)

9.3 实验内容

- 1) 传输模式报文处理 (AH 协议) ;

根据系统提供的已经建立安全联盟的信息,对于接收到的报文和发送的报文,在 IPSEC 传输模式的工作状态下,根据 AH 协议使用系统提供的认证算法对报文进行相应的处理。

- 2) 隧道模式报文处理 (ESP 协议);

根据系统提供的已经建立安全联盟的信息,对于接收到的报文和需要发送的报文,在 IPSEC 隧道模式的工作状态下,根据 ESP 协议使用系统提供的认证和加密算法对报文进行相应的处理。

- 3) 混合模式报文处理 (AH 协议+ESP 协议) ;

根据系统提供的已经建立安全联盟的信息,对于接收到的报文和需要发送的报文,在 IPSEC 混合模式的工作状态下,根据 AH 协议和 ESP 协议使用系统提供的认证和加密算法对报文进行相应的处理。

- 4) 3DES 加密算法实现 (ESP 协议);

根据系统提供的已经建立安全联盟的信息,对于接收到的报文和需要发送的报文,在 IPSEC 传输模式的工作状态下,根据 ESP 协议通过使用系统的 3des 加密算法对报文进行相应的处理。

9.4 实验帮助

9.4.1 IPSEC 协议介绍

9.4.1.1 协议简述

IPsec (Internet Protocol Security) 即 Internet 安全协议, 是 IETF 提供 Internet 安全通信的一系列规范, 它提供私有信息通过公用网的安全保障。IPSec 适用于目前的版本 IPv4 和下一代 IPv6。由于 IPSec 在 TCP/IP 协议的核心层——IP 层实现, 因此可以有效地保护各种上层协议, 并为各种安全服务提供一个统一的平台。IPSec 也是被下一代 Internet 所采用的网络安全协议。IPSec 协议是现在 VPN 开发中使用的最广泛的一种协议, 它有可能在将来成为 IPVPN 的标准。

IPsec 使用两个不同的协议——AH 和 ESP 来确保通信的认证、完整性和机密性。它既可以保护整个 IP 数据报也可以只保护上层协议。适当的模式称为: 隧道模式和传送模式。在隧道模式下, IP 数据报被 IPsec 协议完全加密成新的数据报; 在传送模式下, 仅仅是有效负荷被 IPsec 协议将 IPsec 头插入 IP 头和上层协议头之间来搬运。

为保护 IP 数据报的完整性, IPsec 协议使用了“散列信息认证代码”(HMAC: hash message authentication codes)。为了得到这个“散列信息认证代码”, IPsec 使用了像 MD5 和 SHA 这样的散列算法根据一个密钥和数据报的内容来生成一个“散列”。这个“散列信息认证代码”包含在 IPsec 协议头并且数据报接受者可以检查“散列信息认证代码”(当然前提是可以访问密钥)。

为了保证数据报的机密性, IPsec 协议使用对称加密算法。IPsec 标准需要 NULL 和 DES 执行者。如今经常使用像 3DES、AES 和 Blowfish 这样更加强的算法。

为了避免拒绝服务攻击 (DoS: denial of service attacks), IPsec 协议使用了一个滑动窗口, 每个数据包被分配到一个序号, 并且只接受在不在窗口中的或新的数据包。旧的数据包立即丢弃。这可以避免重复攻击, 即: 攻击者记录原始的数据并稍后再次重发。本实验的设计中没有加入抗重播服务。

通信的双方为了能够加密和解密 IPsec 数据包, 他们需要一种方法来保存通信的密钥、算法和 IP 地址等有关信息。所有的这些用来保护 IP 数据报的参数保存在一个“安全联盟”(SA: security association) 中。“安全

联盟”依次保存在一个“安全联盟”数据库（SAD: security association database）中。

每个“安全联盟”定义了以下参数：

- 1) 结果 IPsec 头中的源 IP 地址和目标 IP 地址。这是 IPsec 双方保护数据包的 IP 地址
- 2) IPsec 协议（AH 还是 ESP）
- 3) IPsec 协议用的加密算法和密钥
- 4) IPsec 协议用的认证算法和密钥
- 5) 安全参数索引（SPI）。这是一个 32 位的数据，用来识别“安全联盟”（SA）
- 6) IPsec 模式（隧道模式或者传输模式）

因为“安全联盟”定义了源 IP 地址和目标 IP 地址，在一个全双工 IPsec 通信中，它只能保护一个方向的通信。如果想保护双方的通信，IPsec 需要两个单向的“安全联盟”。

“安全联盟”仅仅声明 IPsec 支持保护通信。需要定义另外的信息来声明什么时候通信需要保护。这些信息保存在“安全策略”（SP: security policy）中，“安全策略”又依次保存在“安全策略库”（SPD: security policy database）中。

一个“安全策略”通常声明下列参数：

- 1) 被保护的数据包的源地址和目标地址。在传输模式下，这同 SA 的地址相同。在隧道模式下，他们可能不一样。
- 2) 被保护的协议或端口。一些 IPsec 执行者不允许定义被保护的协议，这种情况下将保护所有涉及到的 IP 地址之间的通信。
- 3) 用来保护数据包的“安全联盟”

9.4.1.2 AH 协议、ESP 协议及 3DES 加密算法介绍

IPsec 协议包括两个协议：认证头协议（AH: Authentication Header）和封装安全负载协议（ESP: Encapsulated Security Payload）。两者都独立于 IP 协议。认证头协议使用 IP 协议的 51，封装安全负载协议使用 IP 协议 50。

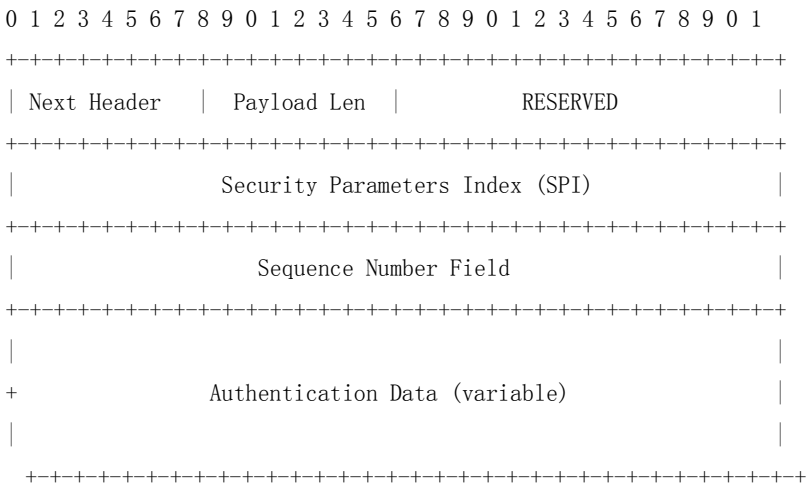
1. AH 协议

认证头协议保护 IP 双数据报的完整性。为了达到这一点，认证头协议计算了一个“散列信息认证代码”（HMAC）来保护完整性。当计算一个“散

列信息认证代码”时，认证头协议基于它的密钥、有效负载和 IP 头的不可变的部分（如 IP 地址）。然后它将认证头放到数据包里。

认证头共长 24 个字节。第一个字节是“下一个头”字段，这个字段指定了下边协议的头。在隧道模式下，一个完整的 IP 数据报被封装，因此这个字段的值是 4。在传送模式下，当被封装的是 TCP 数据报时相应的值是 6。下一个字节指定了的有效负载的长度。这个字段下边是两个保留字节。再下边两个字节指定了 32 位长的安全参数索引（SPI：Security Parameter Index）。安全参数索引指定了解安全负载封装使用的“安全联盟”。32 位长的序号防止重复攻击。最后 96 位保存了“散列信息认证代码”（HMAC）。“散列信息认证代码”保护了数据包的完整性，因为只有双方知道可以创建和检查“散列信息认证代码”的密钥。

因为认证头协议保护了 IP 数据包括 IP 头的不可变的部分(如 IP 地址)，认证头协议不允许网络地址转换（NAT：Network Address Translation）。网络地址转换将 IP 头的 IP 地址（通常是源 IP 地址）改为不同的 IP 地址，当改变后“散列信息认证代码”就不再有效了。IPsec 协议的“横越网络地址转换”（NAT-Traversal）扩展围绕这一限制实现了一条道路。下图显示了认证头的结构。



➤ **Next Header**

该字段长度为 8 位，指明在 AH 协议头后的有效载荷的协议类型。

➤ **Payload len**

该字段长度为 8 位，指明在 AH 协议头后的有效载荷的长度。

➤ **RESERVED**

该字段长度为 16 位，是预留位，默认值为全零。

➤ **Security Parameters Index**

该字段长度为 32 位，它与目的地址和 AH 认证协议唯一的标识一个安全连接。

➤ **Sequence Number Field**

该字段长度为 32 位，是一个累加的值，为这个安全连接提供了抗重播功能，初始化为 0，第一个包的值为 1。

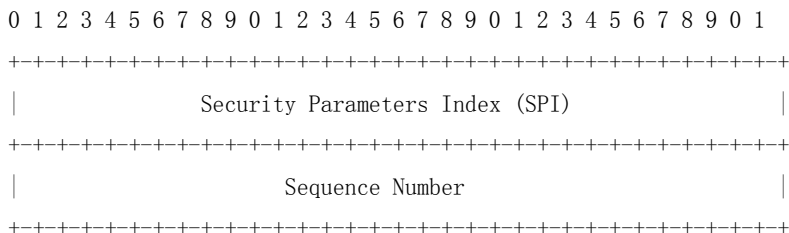
➤ **Authentication Data**

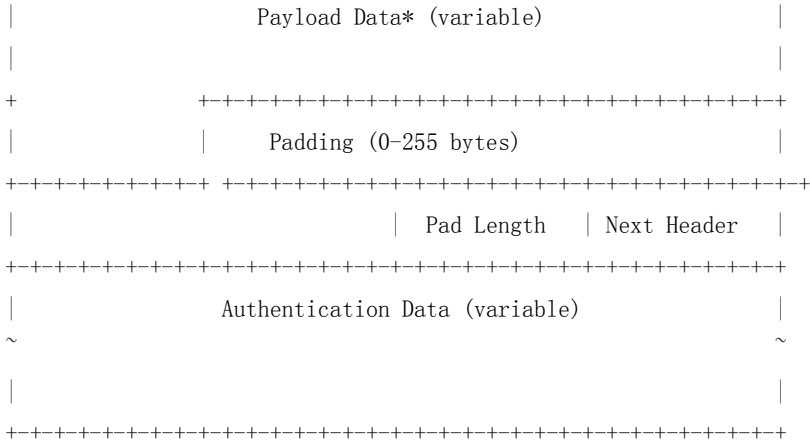
该字段为可变长度，但是必须是 32 的整数倍，用来存放报文完整性校验值。

2. ESP 协议

安全负载封装协议既可以使用“散列信息认证代码”来确保数据包的完整性，又可以使用加密算法保证机密性。在加密数据包并计算出“散列信息认证代码”后，生成安全负载封装头并加入数据包。安全负载封装头的第一个双字节指定了安全参数索引（SPI），这个“索引”指定了解安全负载封装数据包用到的“安全联盟”。下一个双字节保存序号。序号被用来预防重复攻击。第三个双字节指定了加密程序使用的“初始向量”（IV：Initialization Vector）。不使用“初始向量”对称加密算法可能被多次攻击。“初始向量”可以确保两个相同的负载被加密成不同的负载。

IPsec 使用密码块来进行加密处理。因此，如果负载的长度不是声的整倍长时可能需要填补。这时需要增加填补长度。下边的两个字节是填补长度。“下一个头”字段指定了下一个头。“安全负载封装”头的最后 96 位长是“散列信息认证代码”，用来确保数据包的完整性。这个“散列信息认证代码”仅仅统计数据包的负载。IP 头不包括在计算之列。





➤ Security Parameters Index (SPI)

SPI 是一个任意的 32 位值，它与目的 IP 地址和安全协议（ESP）结合，唯一地标识这个数据报的 SA。

➤ Sequence Number

这个无符号的、32 位字段包含一个单调递增的计数器值（序列号）。

➤ Payload Data

有效载荷数据是变长字段，它包含下一个头字段描述的数据。

➤ Padding

供加密使用的填充字段，字节数为（0-255）。

➤ Pad Length

填充字段的长度。

➤ Next Header

该字段为 8 位字节，标识有效载荷字段中数据的类型。

➤ Authentication Data

验证字段用来存放 ESP 报文的完整性校验值，该字段可选，只有 SA 选择验证服务的时候，才包含验证数据字段。

3. 3DES 加密算法

3DES 是 DES 加密算法的一种模式，它使用 3 条 64 位的密钥对数据进行三次加密。数据加密标准（DES）是美国的一种由来已久的加密标准，它使用对称密钥加密法，并于 1981 年被 ANSI 组织规范为 ANSI X.3.92。DES 使用 56 位密钥和密码块的方法，而在密码块的方法中，文本被分成 64 位大小

的文本块然后再进行加密。比起最初的 DES，3DES 更为安全。

3DES（即 Triple DES）是 DES 向 AES 过渡的加密算法（1999 年，NIST 将 3-DES 指定为过渡的加密标准），是 DES 的一个更安全的变形。它以 DES 为基本模块，通过组合分组方法设计出分组加密算法，其具体实现如下：设 $E_k()$ 和 $D_k()$ 代表 DES 算法的加密和解密过程， K 代表 DES 算法使用的密钥， P 代表明文， C 代表密表，这样，

加密过程为： $C = E_{k3}(D_{k2}(E_{k1}(P)))$

解密过程为： $P = D_{k1}(E_{k2}(D_{k3}(C)))$

K_1 、 K_2 、 K_3 决定了算法的安全性，若三个密钥互不相同，本质上就相当于用一个长为 168 位的密钥进行加密。多年来，它在对付强力攻击时是比较安全的。若数据对安全性要求不那么高， K_1 可以等于 K_3 。在这种情况下，密钥的有效长度为 112 位。

(1) 先使用密钥 1 通过 des 加密算法对明文 1 加密，得到密文 1

(2) 再用密钥 2 通过 des 解密算法对密文 1 解密，得到明文 2（肯定不是原来的明文）

(3) 最后再用密钥 3 对明文 2 用 des 加密算法加密，得到密文 2。

注意：本实验设计中的安全联盟只提供两个密钥，即在加密过程中密钥 3 用密钥 1 代替，这样总共密钥就是 128 位了，并且在 ESP 协议使用 3DES 加密算法对报文加密时没有加入初始化向量。

9.4.2 需要实现的接口函数

1) 收到 ipsec 报文事件处理函数

```
int stud_ipsec_input(char * pBuffer,unsigned int len)
```

参数：

pBuffer：指向当前接收报文的指针,从 IP 报头开始

len：接收到的 ipsec 报文长度

返回值：

0：成功解封装 ipsec 报文

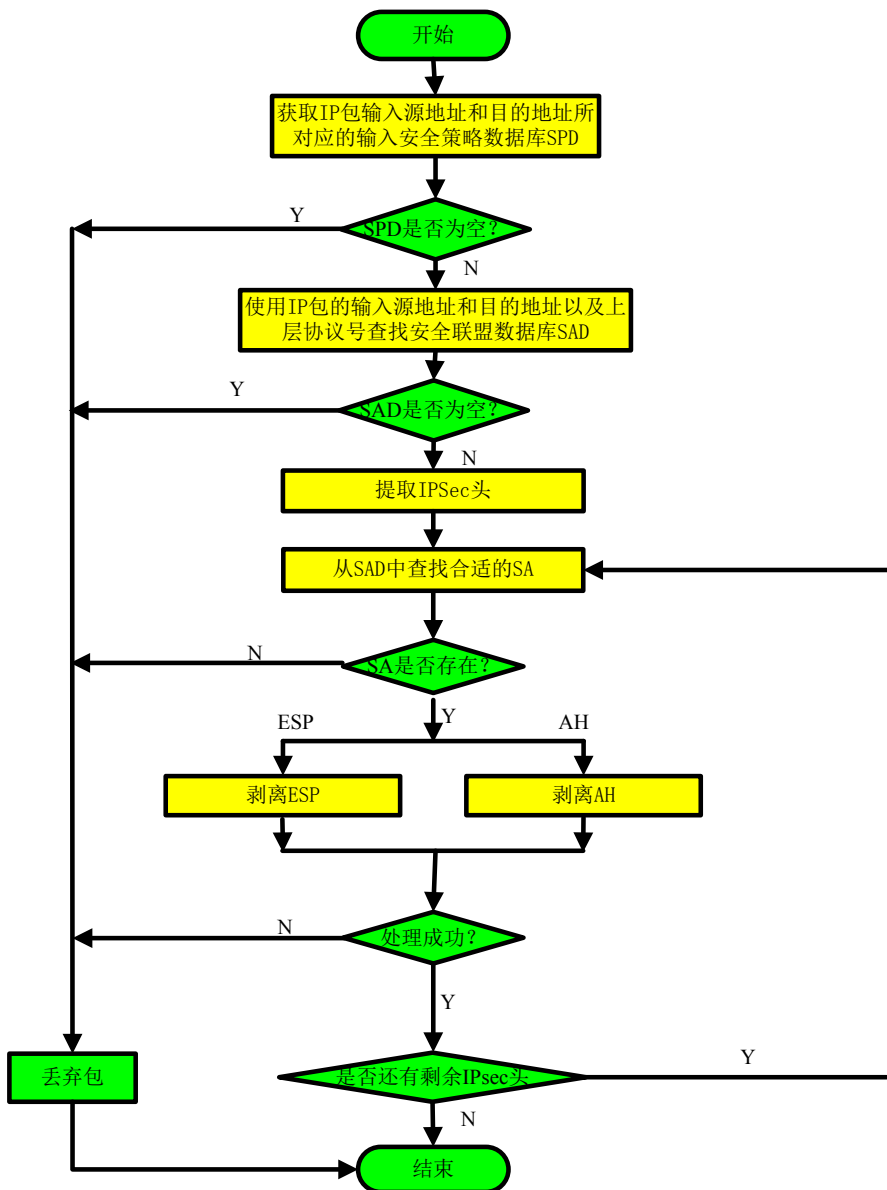
1：解封装过程出现错误

说明：

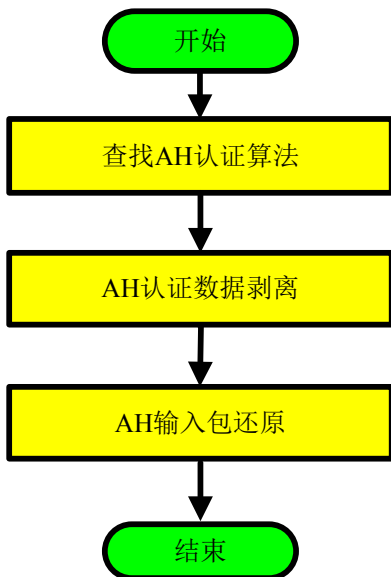
当系统收到 ipsec 报文时，会调用此函数，学生编写此函数，应该实现如下功能：

根据系统提供的安全联盟数据库，按照报文源地址、目的地址和上层协

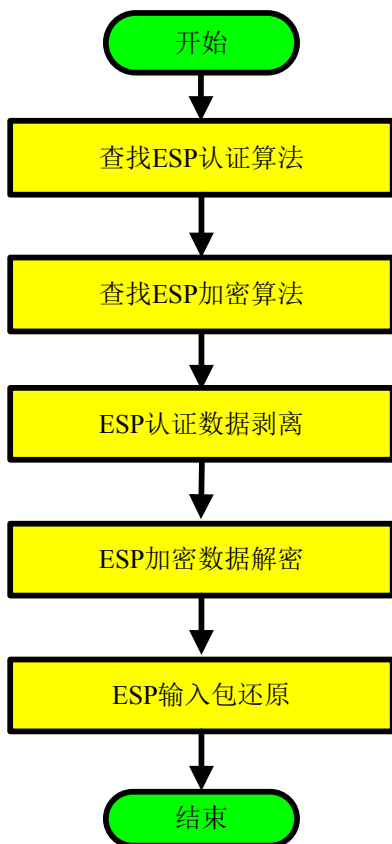
议号查找与之匹配的安全策略，确定与该安全策略对应的安全联盟束，再根据源地址、目的地址和上层协议号查找于之对应的安全联盟，从而根据找到的安全联盟中的模式、认证加密算法和密钥处理该报文。（注意：有可能被多次保护，需要按照先后顺序对报文进行认证和解密），具体流程图如下图所示：



AH 输入处理流程图:



ESP 输入处理流程图:



2) 发送 ipsec 报文事件处理函数

int stud_ipsec_output(char * pBuffer,unsigned int len)

参数:

pBuffer: 指向当前接收报文的指针,从 IP 报头开始

len: 接收到报文长度

返回值:

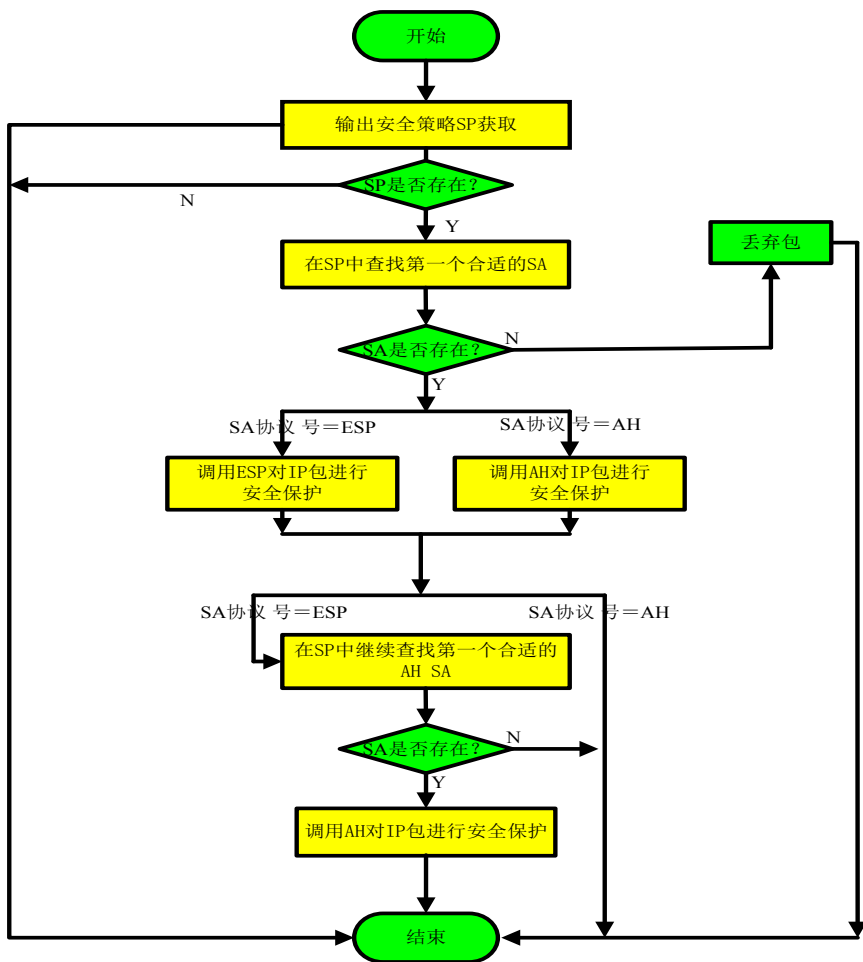
0: 成功封装 ipsec 报文

1: 封装过程出现错误

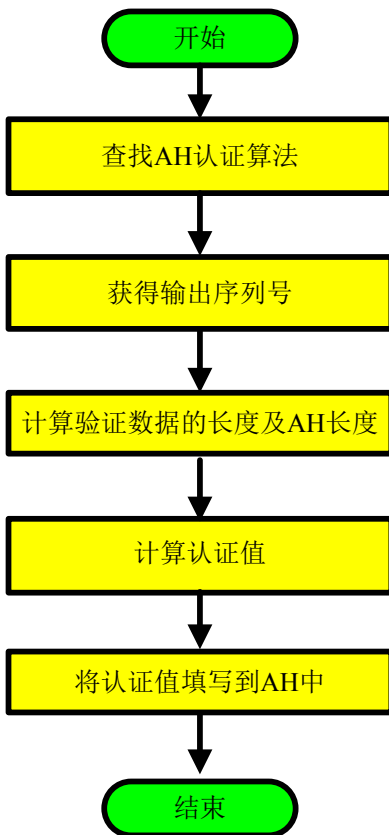
说明：

当系统收到普通报文时，会调用此函数，学生编写此函数，应该实现如下功能：

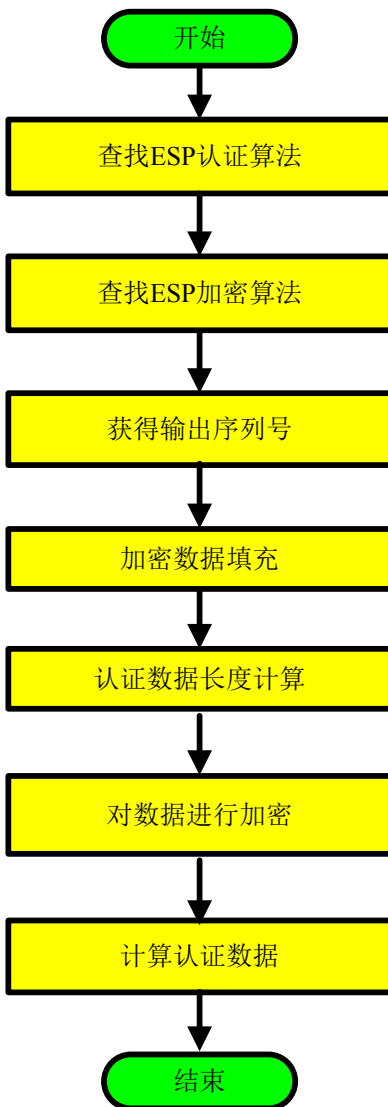
根据系统提供的安全策略数据库，按照报文源地址、目的地址和上层协议号查找于之匹配的安全策略，确定与该安全策略对应的安全联盟束，再根据源地址、目的地址和上层协议号查找于之对应的安全联盟，从而根据找到的安全联盟中的模式、认证加密算法和密钥处理该报文。（注意：有可能一个数据流命中多个安全联盟，需要按照命中的先后顺序对数据流进行安全保护），处理流程图如下图所示：



AH 输出处理流程图:



ESP 输出处理流程图:



9.4.3 数据结构以及宏定义

系统定义了安全策略头结构定义，如下：

```
typedef struct ipsecSpdHeader  
{
```

```
struct ipsecPolicySelector *spSelector;    //指向当前接口安全策略
数据库的头指针
```

```
}IPSEC_SPD_HEADER;
```

系统定义安全策略结构定义，如下：

```
typedef struct ipsecPolicySelector
```

```
{
```

```
    struct ipsecPolicySelector *next;    //指向下一条安全策略条目
```

```
    DWORD srcAddr;    //安全策略中数据流的源地址
```

```
    DWORD dstAddr;    //安全策略中数据流的目的地址
```

```
    DWORD srcPref;    //安全策略中数据流源地址的前缀
```

```
    DWORD dstPref;    //安全策略中数据流目的地址的前缀
```

```
    struct ipsecAssoc *spSA; //指向当前接口安全联盟数据库的头指
针
```

```
}IPSEC_POLICY_SELECTOR;
```

系统定义的安全联盟密钥结构定义，如下：

```
typedef struct ipsecSaKey
```

```
{
```

```
    BYTE keyNum;    /* the number of keys */
```

```
    unsigned short keyBits;    /* bits of one key */
```

```
    BYTE key[1];    /* keybits/8 */
```

```
}IPSEC_SA_KEY;
```

系统定义的安全联盟结构定义，如下：

```
typedef struct ipsecAssoc
```

```
{
```

```
    struct ipsecAssoc *next;    //指向下一条安全联盟的指针
```

```
    struct ipsecAssocSelector selector;    //安全联盟对应的数据流
```

```
    DWORD srcAddr;    //当前安全联盟的起始地址
```

```
    DWORD dstAddr;    //安全联盟的目的地址
```

```
    BYTE proto;    //安全联盟所使用的协议（AH 或者 ESP）
```

```
    BYTE mode;    //安全联盟的工作模式（隧道模式或者传输模式）
```

```
    DWORD spi;    //安全联盟的索引号
```

```
    BYTE algAuth;    //安全联盟所使用的认证算法名称
```

```

        BYTE  algEncry;           //安全联盟所使用的加密算法名称
        struct ipsecSaKey  *keyAuth;    //安全认证密钥
        struct ipsecSaKey  *keyEncry;    //安全联盟加密密钥 1（DES 加
加密算法的密钥，3DES 加密算法的密钥 1）
        struct ipsecSaKey *keyEncry2;    //安全联盟加密密钥 2（3DES 加
加密算法的密钥 2）
    
```

```

        DWORD  ivLen;           //安全联盟初始化向量长度（单位为字节）
    
```

```

        BYTE  iv[20];           //安全联盟初始化向量
    
```

```

}IPSEC_ASSOC;
    
```

系统定义了加密算法结构如下：

```

typedef struct espAlgorithm
    
```

```

{
    
```

```

        unsigned int  ivLen;           //初始化向量长度（单位为字节）
    
```

```

        const char  *name;             //算法名称
    
```

```

        /*加密算法函数指针*/
    
```

```

        int (* encrypt)(unsigned char *plainText, unsigned int plainLen,
unsigned char *encryText, unsigned int *encryLen, char *pKeys, char *iv);
    
```

```

        /* 解密算法函数指针 */
    
```

```

        int (* decrypt)(unsigned char *pEncryText, unsigned int encryLen,
unsigned char *pPlainText, unsigned int *pPlainLen, char *pKeys, char *iv);
    
```

```

        /*填充函数指针 */
    
```

```

        int (* padding)(unsigned char *pOriginalText, unsigned int originalLen,
unsigned char **pPaddingText, unsigned int *paddedLen, unsigned char
nxtHdr);
    
```

```

}ESP_ALGORITHM;
    
```

系统定义了认证算法结构如下：

```

typedef struct ahAlgorithm
    
```

```

{
    
```

```

        unsigned int  digestSize;      //认证摘要数据长度（单位为比特）
    
```

```

        unsigned int  dataLen;         //源数据长度（单位为比特）
    
```

```

        const char  *name;             //认证算法名称
    
```

```

        void (*hashInit)(IKE_MD5_CTX *ctx_hash);/* initialize context */
    
```

```
void (*hashUpdate)(IKE_MD5_CTX *ctx_hash, const unsigned char
*input, unsigned int len);    /* add input to hash */
void (*hashFinal)(unsigned char *output, IKE_MD5_CTX *ctx_hash);
/* finalize hash */
```

```
}AH_ALGORITHM;
```

系统定义了 ESP 协议结构如下：

```
typedef struct ipsec_esp
```

```
{
```

```
    unsigned int    spi;                /* 安全索引号*/
```

```
    unsigned int    seq;                /* 序列号*/
```

```
}IPSEC_ESP;
```

系统定义了 AH 协议结构如下：

```
typedef struct ipsec_ah
```

```
{
```

```
    BYTE    ah_nxt;                    //下一头
```

```
    BYTE    ah_len;                    //认证数据长度
```

```
    unsigned short    ah_reserved;        //保留位
```

```
    DWORD    ah_spi;                    //安全索引号
```

```
    DWORD    ah_seq;                    //序列号
```

```
}IPSEC_AH;
```

系统定义了如下宏：

```
#define IPSEC_AUTH_MD5 1                //MD5 认证算法
```

```
#define IPSEC_ENCRY_DES 1                //DES 加密算法
```

```
#define IPSEC_ENCRY_3DES 2                //3DES 加密算法
```

```
#define IPSEC_MODE_TUNNEL 1                //隧道模式
```

```
#define IPSEC_MODE_TRANSPORT 2            //传输模式
```

```
#define IPSEC_PROTO_AH 1                //AH 协议
```

```
#define IPSEC_PROTO_ESP 2                //ESP 协议
```

9.4.4 系统提供的接口函数及全局变量

1) 系统接收报文函数

```
void ipsec_Receive_process(char *pBuffer,int dwLen)
```

参数：

pBuffer: 指向当前解封装完毕报文的指针,从 IP 报头开始

len: 接收到解封装完毕报文长度

返回值:

无

说明:

学生在完成对 IPSEC 报文解封装后,将报文交给该函数进行处理。

2) 系统发送报文函数

void ipsec_Send_process(char *pBuffer, int iLen)

参数:

pBuffer: 指向当前封装完毕报文的指针,从 IP 报头开始

len: 接收到解封装完毕报文长度

返回值:

无

说明:

学生在完成对普通报文的封装后,将报文交给该函数进行处理。

3) 认证算法查找函数

AH_ALGORITHM *ahAlgorithmLookup(unsigned char algAuth)

参数:

algAuth: 认证算法名称

返回值:

存储有关认证算法信息的结构体指针

说明:

学生在从安全联盟中获得当前所使用的认证算法名称,通过该函数获得系统提供的认证算法函数接口以及相关信息。

4) 加密算法查找函数

ESP_ALGORITHM *espAlgorithmLookup(unsigned char algEncry)

参数:

algEncry: 加密算法名称

返回值:

存储有关加密算法信息的结构体指针

说明:

学生在从安全联盟中获得当前所使用的加密算法名称,通过该函数获得

系统提供的加密算法函数接口以及相关信息。

5) AH 协议认证摘要计算函数

```
int ah4CalculateICV(unsigned char *pPacketCpy, unsigned short totalLen,  
unsigned char *pOutput, const AH_ALGORITHM *pAlg,  
IPSEC_ASSOC *pSA)
```

参数:

pPacketCpy: 指向源报文拷贝的指针 (请注意是源数据的拷贝, 从 IP 报头开始)

totalLen: 源报文长度

pOutput: 指向存储计算出报文摘要信息的指针

pAlg: 认证算法结构体指针

pSA: 安全联盟指针

返回值:

失败: -1

成功: 0

说明:

学生根据系统提供的这个函数来计算当前 AH 协议报文中的认证数据摘要。

6) ESP 协议认证摘要计算函数

```
int espAuthCalculate(unsigned char *pData, unsigned short dataLen,  
unsigned char *pOutput, AH_ALGORITHM *pAlg, IPSEC_ASSOC  
*pSA)
```

参数:

pData: 指向源报文 ESP 报头的指针

dataLen: 需要计算认证摘要的报文长度

pOutput: 指向存储计算出的报文摘要信息的指针

pAlg: 认证算法结构体指针

pSA: 安全联盟指针

返回值:

失败: -1

成功: 0

说明:

学生根据系统提供的这个函数来计算当前 ESP 协议报文中的认证数据摘要。

7) ESP 协议加密函数

```
int (* encrypt)(unsigned char *plainText, unsigned int plainLen, unsigned char *encryText, unsigned int *encryLen, char *pKeys, char *iv);
```

参数:

plainText: 指向需要加密的原文头指针(如果保护模式为隧道模式从 IP 报头开始,如果保护模式为传输模式从 IP 报文的有效载荷开始)

plainLen: 原文长度

encryText: 指向有效载荷加密后的指针

encryLen: 获取有效载荷加密后的数据长度的整型指针

pKeys: 指向加密密钥的指针

iv: 指向初始化向量的指针

返回值:

失败: -1

成功: 0

说明:

学生根据安全联盟提供的函数指针,利用系统提供的加密算法获得需要加密的报文部分的密文。

8) ESP 协议解密函数

```
int (* decrypt)(unsigned char *pEncryText, unsigned int encryLen, unsigned char *pPlainText, unsigned int *pPlainLen, char *pKeys, char *iv);
```

参数:

encryText: 指向有效载荷密文的指针,指向 ESP 头部之后的数据

encryLen: 密文的长度

plainText: 指向解密后的原文指针

plainLen: 获取有效载荷解密后数据长度的整型指针

pKeys: 指向加密密钥的指针

iv: 指向初始化向量的指针

返回值:

失败: -1

成功: 0

说明:

学生根据安全联盟提供的函数指针, 利用系统提供的解密算法获得需要解密报文部分的原文。

9) ESP 协议填充函数

```
int (* padding)(unsigned char *pOriginalText, unsigned int originalLen,
unsigned char **pPaddingText, unsigned int *paddedLen, unsigned char
nxtHdr);
```

参数:

pOriginalText: 指向源数据的指针(如果保护模式为隧道模式从 IP 报头开始,如果保护模式为传输模式从 IP 报文的有效载荷开始)

originalLen: 源数据的长度

pPaddingText: 获取填充后的报文拷贝指针 (在函数内部为其分配内存, 在函数外部需要释放)

plainLen: 获取有效载荷填充之后数据长度的整型指针

nxtHdr: 被加密报文的协议号 (需要加入 ESP 协议填充头中下一头字段值)

返回值:

失败: -1

成功: 0

说明:

学生根据安全联盟提供的函数指针, 由于 ESP 算法的加密数据块必须为 32bit 的整数倍, 利用系统提供的填充函数, 获取填充后的报文拷贝。

10) struct ipsecSpdHeader *g_ipsec_spd_in

说明: 该全局变量为接收 IPsec 报文时的安全策略数据库, 数据流根据目的地址、源地址查询安全策略数据库, 以决定该数据流是否被命中, 如果命中使用该安全策略中 IPSEC 安全联盟对应的信息对报文解除 IPSEC 协议保护。

11) struct ipsecSpdHeader *g_ipsec_spd_out

说明: 该全局变量为转发正常报文时的安全策略数据库, 数据流根据目的地址、源地址查询安全策略数据库, 以决定该数据流是否被命中, 如果命中使用该安全策略中 IPSEC 安全联盟提供的安全协议进行保护。

10 移动 IP 协议实验

10.1 实验目的

移动 IP 是一种在全球因特网上提供移动功能的方案。它使移动节点在切换链路时仍然可保持现有通信, 并且, 移动 IP 提供了一种 IP 路由机制, 可以使移动节点以一个永久的 IP 地址连接到任何链路上。

本实验分“角色”(即 MN(移动节点)、FA(外地代理)和 HA(家乡代理))的实现移动 IP 的三个技术:代理搜索、注册和包传送。

通过本实验, 学生可以理解移动 IP 的工作原理, 掌握移动 IP 的主要技术。

10.2 实验要求

充分理解移动 IP 协议, 了解它的三个“角色”的主要作用, 能够实现以下功能:

- 1) 代理搜索-移动节点功能的实现
- 2) 注册-移动节点功能的实现
- 3) 注册-外地代理功能的实现
- 4) 注册-家乡代理功能的实现
- 5) 包传送-家乡代理功能的实现
- 6) 包传送-外地代理功能的实现

10.3 实验内容

- 1) 实现代理搜索过程 MN 的主要功能

根据系统提供的参数组装并发送代理请求报文, 收到代理应答消息后, 判断出该代理是 FA 还是 HA, 如果是 FA 提交转交地址(本实验默认采用外地代理地址作为转交地址);

- 2) 实现注册过程中 MN 的主要功能

根据系统提供的参数组装并发送注册请求消息, 收到注册应答消息后, 根据报文的内容修改 MN 的路由表;

3) 实现注册过程中 FA 的主要功能

- (a) 收到注册请求报文后, 通过查找路由表将该报文重新进行组装, 然后中继到 HA;
- (b) 从 HA 收到注册应答后, 判断本次注册过程是否成功, 成功则需要修改 FA 的路由表, 最后将该应答消息中继到 MN。

4) 实现注册过程中 HA 的主要功能

对于收到的注册请求报文, 进行合法性检查 (只需检查要注册的 MN 是否合法), 如果合法, 则修改它的绑定表, 最后发送注册应答消息给 FA。

5) 实现包传送中 HA 的主要功能

如果收到目的地址是 MN 家乡地址的数据包, 查找绑定表, 将该包进行 IP 封装, 然后发送给 FA; 否则, 查找路由表, 进行正常的 IP 转发。

6) 实现包传送中 FA 的主要功能

如果收到目的地址是 MN 家乡地址的数据包, 查找路由表, 将该包进行解封封装, 然后发送给 MN。

如果收到来自移动节点的数据包: 必须路由移动节点发来的数据报, 必须验证 IP 头部检验和, 减少 IP 的生存期 (Time To Live), 重新计算 IP 头部检验和, 并把这样的数据报转发。

10.4 实验帮助

10.4.1 移动 IP 协议介绍

10.4.1.1 协议简述

越来越多的网络用户通过无线技术接入网络, 它有很多优点。然而也存在着不足之处, 当用户移到一个新网络时连接就会断开。移动 IP 是一种网络标准, 它实现跨越不同网络的无缝漫游功能。用户跨越不同网络边界时, 使用移动 IP 技术可以确保如网络技术、多媒体业务和虚拟专用网络等各种应用的永不中断连接。

采用传统 IP 技术的主机在移动到另外一个网段或者子网地时候, 由于不同的网段对应于不同的 IP 地址, 用户不能使用原有 IP 地址进行通信, 必须修改主机 IP 地址为所在子网的 IP 地址, 而且由于各种网络设置, 用户一

般不能继续访问原有网络的资源，其他用户也无法通过该用户原有的 IP 地址访问该用户。

所谓移动 IP 技术，是指移动用户可在跨网络随意移动和漫游中，使用基于 TCP/IP 协议的网络时，不用修改计算机原来的 IP 地址，同时，继续享有原网络中一切权限。简单地说，移动 IP 就是实现网络全方位的移动或者漫游。

移动 IP 应用于所有基于 TCP/IP 网络环境中，它为人们提供了无限广阔的网络漫游服务。譬如：在用户离开北京总公司，出差到上海分公司时，只要简单的将移动终端(便携笔记本电脑、PDA 等所有基于 IP 的设备)连接至上海分公司网络上，不需要做其他任何改动，那么用户就可以享受到跟在北京总公司里一样的所有操作。用户依旧能使用北京总公司的相关打印机；诸如此类的种种操作，让用户感觉不到自己身在外地。换句话说：移动 IP 的应用让用户的“家”网络随处可以安“家”。

10.4.1.2 移动 IP 的几种报文结构

1. 代理请求消息和代理应答消息

1) 代理请求消息

代理请求消息由 20 字节的 IP 头加上 8 字节的路由器广播消息组成。见下表：

		服务类型	总长度	
标识			标记	片偏移
生存时间		校验和		
源地址 = 移动节点的家乡地址				
目的地址 = 255. 255. 255. 255（广播）或 224. 0. 0. 2（组播）				
类型 = 10	Code = 0	校验和		
保留				

2) 代理广播消息

代理广播消息由 IP 报头、ICMP 路由器广播和移动代理广播扩展以及其他一些可选扩展组成，本实验不考虑可选扩展。本实验中，ICMP 路由器广播域中地址数为 1。

		服务类型	总长度	
标识			标记	片偏移
生存时间		校验和		
源地址 = 链路上的家乡或外地代理地址				
目的地址 = 255.255.255.255（广播）或 224.0.0.2(组播)				
类型 = 10	Code = 0	校验和		
地址数	地址宽度	生存时间		
路由器地址[1]				
优选级[1]				
.....				
类型=16	长度	序号		
注册生存时间		保留		
转交地址				

2. 注册请求和注册应答消息报文格式

1) 注册请求消息

注册请求消息由 IP 头、UDP 头、注册请求定长部分和认证扩展组成。

0	1	2	3	4
		服务类型	总长度	
标识			标记	片偏移

生存时间		校验和
源地址 = 移动节点的家乡地址		
目的地址 = 255.255.255.255（广播）或 224.0.0.2（组播）		
源端口		目的端口=434
长度		校验和
类型 = 1	ignore	生存时间
移动节点家乡地址		
家乡代理地址		
转发地址		
标识		
类型	长度	安全参数
索引		
认证算法		

2) 注册应答消息

类型 = 3	Code	生存时间
移动节点家乡地址		
家乡代理地址		

标识

10.4.2 实验拓扑

本实验的所有测试例，学生代码作为 MN、FA 或 HA 这三个功能实体中的某一个“角色”，系统则作为其他两个“角色”。

10.4.3 需要实现的接口函数

1) 移动节点发送代理请求报文函数

```
int stud_MN_icmp_send()
```

参数：无

返回值：

0：成功发送报文

1：发送失败

说明：本函数和下一个函数是实现代理搜索中移动节点功能的函数。本函数作用是发送一条代理请求消息。另外注意 ignore 字段应填为 0。

学生组装注册请求报文，然后调用系统给出的相关函数将报文发送出去。

2) 移动节点接收代理应答报文函数

```
int stud_MN_icmp_recv(char *buffer,unsigned short len)
```

参数：

buffer：指向接收到的 icmp 报文内容的指针

len：接收到的报文的长度

返回值：

1：外地代理

2：家乡代理

说明：当系统接收到代理应答报文时，调用该函数，当根据该广播消息的源地址得出是外地代理，那么提交转交地址，函数返回值为1；否则为家乡代理，返回值为2。在本函数中调用getHAHaddress()函数获取家乡代理的地址，调用submitCareofadd(unsigned int careofadd)提交转交地址。

3) 移动节点发送注册应答报文函数


```
int stud_MN_send_Regi_req(char* pdata,unsigned char len,unsigned short  
ttl,unsigned int HA_addr,unsigned int FA_addr,unsigned int MN_Haddr,unsigned  
int iden_low,unsigned int iden_high)
```

参数:

pdata: 要发送的注册请求消息所携带的数据;
Len: 要发送的注册请求消息所携带的数据的长度;
ttl: 生存时间;
HA_addr: 家乡代理地址;
FA_addr: 外地代理地址;
MN_Haddr: 移动节点家乡地址;
Iden_low: 标识号的低 32 位;
Iden_high: 标识号的高 32 位

返回值:

0: 成功发送报文
1: 发送失败

说明: 本函数和下一函数是移动节点注册功能的实现函数。本函数的功能是发送一条注册请求消息。

4) 移动节点接收注册应答报文函数

```
int stud_MN_recv_Regi_rep(char *pbuffer,unsigned short len,unsigned int if_no)
```

参数:

pbuffer: 指向接收的报文内容的指针
len: 接收的报文的长度
if_no:接收报文的接口号

返回值:

0: 成功发送报文
1: 发送失败

说明: 系统调用该函数完成移动节点接收注册应答消息, 首先根据注册消息中的 CODE 值判断此次注册过程是成功还是失败, 如果 CODE 值既不为 1 又不为 0, 那么函数返回 1, 否则修改 MN 的路由表, 函数返回 0。

5) 外地代理接收注册请求报文函数

```
int stud_FA_recv_Regi_req(char *pbuffer,unsigned short len, unsigned int mask,  
unsigned int if_no)
```

参数:

Pbuffer: 指向接收的报文内容的指针

Len: 接收的报文的长度

Mask: MN 的掩码

If_no: 接收报文的接口号

返回值:

0: 成功接收报文

1: 接收失败

说明: 本函数是完成 FA 接收注册请求消息的函数。当接收到报文后, FA 将该报文中继到 HA, 即需要改变原来的 IP 包头的源和目的地址, 调用相应的系统函数将该注册消息中继到 HA, 并且要记录下 MN 地址、掩码、接收该报文的接口号。

6) 外地代理接收注册应答报文函数

int stud_FA_rcv_Regi_rep(char *pbuffer,unsigned short len)

参数:

Pbuffer: 指向接收的报文内容的指针

Len: 接收的报文的长度

返回值:

0: 成功接收报文

1: 接收失败

说明: 本函数是实现 FA 处理从 HA 接收注册应答报文的功能的函数。当接收到报文后, 首先根据报文的 CODE 值判断本次注册过程是成功还是失败, (如果成功则需要增加一条到 MN 的特定主机路由)最后调用相应的系统函数将消息中继到 MN。

7) 家乡代理接收注册请求报文函数

int stud_HA_rcv_Regi_req(char *pbuffer,unsigned short len,unsigned int mask)

参数:

Pbuffer: 指向接收的报文内容的指针

Len: 接收的报文的长度

mask: FA 的掩码

返回值:

0: 成功接收报文

1: 接收失败

说明：本函数是 HA 注册功能的实现函数。本函数的功能是从系统（FA）接收一条注册请求报文，首先判断所要注册的 MN 地址是不是本地地址（本地 MN 地址由系统函数 `unsigned int getMNHaddress()` 提供），如果不是，则将注册应答消息中 CODE 值置为 131，代表本次注册过程失败；反之如果是，则将 CODE 值置为 0，代表本次注册过程成功。最后调用 `int`

`HA_regi_sendIpPkt (unsigned char type,unsigned char code,unsigned int destadd,unsigned int srcadd,unsigned int Mn_addr,unsigned int Ha_addr)` 将注册应答消息发送出去。

8) 家乡代理接收发往 MN 的报文函数

`int stud_packertrans_HA_rcv(char *pbuffer,unsigned short len)`

参数：

Pbuffer: 指向接收的报文内容的指针

Len: 接收的报文的长度

返回值：

0: 成功接收报文

1: 接收失败

说明：本函数是家乡代理包传送功能的实现函数。当接收到一个数据包后，首先查看该包的目地地址是否是 MN 的地址，如果是，则查找“绑定”表，将该包进行 IP 封装，然后发送出去；否则，查找路由表，调用系统函数 `HA_forward_ipv4packet(pbuffer,len)` 进行正常的 IP 转发。

扩展：如果家乡代理收到的数据报是经过封装的数据包（参考 RFC2002 4.2.3 家乡代理方面的考虑），作如下逻辑处理：

if (数据报是封装数据报)

if （内层目的地址==外层目的地址）

if （外层源地址==当前转交地址）（丢弃）

if （外层源地址!=当前转交地址）（外层目的地址改成转交地址，直接转发，不用重新封装数据报）

else （内层目的地址!=外层目的地址）

家乡代理再次封装数据包（递归封装，家乡代理把整个数据报转发到移动节点，就像转发其它数据报一样（不管是否经过封装。）

9) 外地代理接收发往 MN 的封装报文函数

`int stud_packertrans_FA_recv(char *pbuffer,unsigned short len)`

参数：

Pbuffer: 指向接收的报文内容的指针

Len: 接收的报文的长度

返回值：

0: 成功接收报文

1: 接收失败

说明：本函数是外地代理包传送功能的实现函数。当接收到一个数据包后，首先由包头的协议号判断是否是封装包，如果是，则进行解封装，然后查找“路由”表，然后调用相应的系统函数 `FA_send_ipv4_toMN(char *pBuffer, int length)` 发送给 MN。

如果收到来自移动节点的数据包：必须路由移动节点发来的数据报，必须验证 IP 头部检验和，减少 IP 的生存期（Time To Live），重新计算 IP 头部检验和，并把这样的数据报转发。

10. 4. 4 系统提供的全局变量

`extern struct stud_MN_route_node *g_MN_route_table;`

该全局变量为系统中MN路由表链表的头指针。

`extern struct stud_FA_route_node *g_FA_route_table;`

该全局变量为系统中FA路由表链表的头指针。

`extern struct stud_HA_route_node *g_HA_route_table;`

该全局变量为系统中HA路由表链表的头指针。

系统以单向链表存储MN、FA、HA的路由表，学生根据需要修改对应的表项，供客户端软件检查。

其中，`stud_MN_route_node`、`stud_FA_route_node`、`stud_HA_route_node` 分别定义如下：

`typedef struct stud_MN_route_node`

{

`unsigned int dest;`

```

        unsigned int mask;
        unsigned int nexthop;
        unsigned int if_no;
        struct stud_MN_route_node *next;
    };
typedef struct stud_FA_route_node
{
    unsigned int dest;
    unsigned int mask;
    unsigned int nexthop;
    unsigned int if_no;
    struct stud_FA_route_node *next;
};
typedef struct stud_HA_route_node
{
    unsigned int dest;
    unsigned int mask;
    unsigned int nexthop;
    unsigned int if_no;
    struct stud_HA_route_node *next;
};
两个宏定义：
#define FA    1
#define HA    2

```

10. 4. 5 系统提供的接口函数

1) 移动节点发送代理请求消息的系统函数

`void icmp_sendIpPkt(unsigned char* pData, unsigned short len)`

参数：

Pbuffer: 指向要发送的报文内容的指针

Len: 发送的报文的长度

说明：在“移动节点代理请求功能的实现”测试例中，学生完成报文的组装后调用该函数发送代理请求消息。

2) 发送移动节点注册请求消息的系统函数

```
void MN_regi_sendIpPkt(unsigned char* pData, unsigned short len)
```

参数:

Pbuffer: 指向要发送的报文内容的指针

Len: 发送的报文的长度

说明: 在“移动节点注册功能的实现”测试例中, 学生完成报文的组装后调用该函数发送注册请求消息。

3) 发送外地代理注册请求报文的系统函数

```
Void FA_sendregi_req(char* pData,unsigned short len,unsigned int  
destadd,unsigned int srcadd)
```

参数:

pData: 指向注册请求消息后面所携带的附加信息的指针 (除去 IP 头和 UDP 头)

Len: 发送的报文的长度

destadd: 目的 IPv4 地址

srcadd: 源 IPv4 地址

说明: 本函数用于“外地代理注册功能的实现”测试例中。学生将收到的注册请求报文“中继”到家乡代理时调用该函数进行处理。

4) 发送外地代理注册应答报文的系统函数

```
Void FA_sendregi_rep(char* pData,unsigned short len,unsigned int  
destadd,unsigned int srcadd)
```

参数:

pData: 指向注册请求消息后面所携带的附加信息的指针 (除去 IP 头和 UDP 头)

Len: 发送的报文的长度

destadd: 目的 IPv4 地址

srcadd: 源 IPv4 地址

说明: 本函数用于“外地代理注册功能的实现”测试例中。学生将收到的注册应答报文进行重新“组装”, 发送到移动节点时调用该函数进行处理。

5) 家乡代理发送注册应答报文的函数

```
void HA_regi_sendIpPkt(unsigned char type,unsigned char code,unsigned int  
destadd,unsigned int srcadd,unsigned int Mn_addr,unsigned int Ha_addr)
```

参数:

type: 报文的类型, 是请求消息还是应答消息

code: CODE 值, 说明注册是成功还是失败

destadd: 目的 IPv4 地址

srcadd: 源 IPv4 地址

Mn_addr: MN 家乡 IPv4 地址

Ha_addr: 家乡代理 IPv4 地址

说明: 本函数用于“家乡代理注册功能的实现”测试例中。学生将要发送的注册应答消息调用该函数进行处理。

6) 家乡代理发送封装包的函数

int HA_send_Encap_packet(char* pBuffer,unsigned short len,unsigned int dstAddr,unsigned int srcAddr)

参数:

pBuffer: 指向要发送的报文的内容的指针

len: 报文的长度

dstAddr: 目的 IPv4 地址

srcAddr: 源 IPv4 地址

说明: 封装之后的包调用本函数进行发送。

7) 外地代理发送解封装包的函数

void FA_send_ipv4_toMN(char *pBuffer, int length)

参数:

pBuffer: 指向要发送的报文的内容的指针

length: 报文的长度

if_no: 发送报文的接口号

说明: 解封装之后的包调用本函数进行发送。

8) 家乡代理转发报文的函数

void HA_forward_ipv4packet(char *pBuffer, int length)

参数:

pBuffer: 指向要发送的报文的内容的指针

length: 报文的长度

说明: 在包传送-家乡代理功能的实现测试例中, 转发不是发往移动节点的数据包调用该函数。

9) 家乡代理丢弃报文的函数

```
void HA_DiscardPkt(char * pBuffer, int length)
```

参数:

pbuffer: 指向要丢弃的报文的内容的指针

length: 报文的长度

说明: 在包传送-家乡代理功能的实现测试例中, 丢弃查不到对应路由表的数据包时调用该函数。

10) 外地代理丢弃报文的函数

```
extern void FA_DiscardPkt(char * pBuffer, int length)
```

参数:

pbuffer: 指向要丢弃的报文的内容的指针

length: 报文的长度

说明: 在包传送-外地代理功能的实现测试例中, 丢弃查不到对应路由表的数据包时调用该函数。

11 IPv4 协议交互实验

11.1 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。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。

11.2 实验要求

根据计算机网络实验系统所提供的上下文,对给出的 IP 报文进行字段检查,如果出错指出其出错字段或组装一个正确的 IP 报头。

11.3 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。在选项中选择对应的项。

2) 实现 IPv4 分组的封装发送

根据题干中给出的上下文环境,封装 IPv4 分组,使用系统提供的发送 IP 报文界面将分组发送出去。

11.4 实验帮助

在使用系统提供的发送 IP 报文界面(图 9.1 发送 IP 报文界面)时,校验和字段的初始值为零。如要计算正确的校验和则需使用 Valid Checksum 选项(图中用红色椭圆标注)。在打开该选项后,系统会自动计算 IP 头部的校验和(即时反应 IP 头部各个字段值的变化)。如需观察当前填充值下 IP 头部对应的十六进制和字符串表示,可点击 Decode 按钮,结果显示在图中蓝色方框标示的区域中。

Version	4	<input checked="" type="checkbox"/> Length Override	0
Header Length	5	Identifier	0
Precedence (TOS Bits 0-2)	000-Routine	Fragment	May Fragment
Delay (TOS Bits 3)	0 - Normal		Last Fragment
Throughput (TOS Bits 4)	0 - Normal	Fragment Offset(x8)	0
Reliability (TOS Bits 5)	0 - Normal	Time To Live	64
Cost (TOS Bits 6)	0 - Normal	Protocol	000 - IP
Reserved (TOS Bits 7)	0	<input type="checkbox"/> Valid Checksum	0x0000

Destination Address	0 . 0 . 0 . 0	Source Address	0 . 0 . 0 . 0
---------------------	---------------	----------------	---------------

IP Head Encoding

```

0000 45 00 00 00 00 00 00 00 40 00 00 00 00 00 00    E
0010 00 00 00 00
          
```

Decode

图 9.1 发送 IP 报文界面

当在检查 IP 报文时，需通过使用 Valid Checksum 选项观察校验和显示是否发生变化。如发生变化则说明该 IP 报文的校验和有错误。校验和的原始值可在蓝色框中的十六进制显示中的相应字段找到。

12 RIP 协议交互实验

12.1 实验目的

通过简单实现路由协议 RIP，深入理解计算机网络中的核心技术——路由技术，并了解计算机网络的路由转发原理。

12.2 实验要求

充分理解 RIP 协议，根据 RIP 协议的流程设计 RIP 协议的报文处理和超时处理函数。能够实现如下功能：

- 1) RIP 报文有效性检查
- 2) 处理 Request 报文
- 3) 处理 Response 报文
- 4) 路由表项超时删除
- 5) 路由表项定时发送

12.3 实验内容

- 1) 对客户端接收到的 RIP 报文进行有效性检查

对客户端接收到的 RIP 协议报文进行合法性检查，如果出错，指出错误原因；

- 2) 处理 Request 报文

正确解析并处理 RIP 协议的 Request 报文，并能够根据报文的内容以及本地路由表组成相应的 Response 报文，回复给 Request 报文的发送者，并实现水平分割；

- 3) 处理 Response 报文

正确解析并处理 RIP 协议的 Response 报文，并根据报文中携带的路由信息更新本地路由表；

- 4) 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；

- 5) 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。

12.4 实验帮助

12.4.1 RIP 协议介绍

12.4.1.1 协议简述

RIP 协议采用的是距离-向量路由算法，该算法早在 Internet 的前身 ARPANET 网络中就已经被广泛采用。在 70 年代中期，Xerox 公司根据它们对互联网的研究成果提出了一套被称为 XNS（Xerox Network System）的网络协议软件。这套协议软件包含了 XNS RIP 协议，该协议就是现在所使用的 RIP 协议的最早原型。80 年代加州大学伯克利分校在开发 Unix 系统的同时在 routed 程序中设计实现了 RIP 协议软件。routed 程序被绑定在 BSD Unix 系统中一起推出，被广泛的应用于早期网络中的机器之间交换路由信息。尽管 RIP/routed 没有非常突出的优点，但是由于 Unix 操作系统的普及，RIP/routed 也逐渐被推广出来，为许多人所接受，成为了中小型网络中最基本的路由协议/程序。

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 的报文结构进行了扩展，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，它在协议报文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。

12. 4. 1. 2RIPv2 协议的报文结构



图 10.1 RIPv2 的报文结构

RIPv2 的报文结构如图 10.1 所示。每个报文都包括一个报文命令字段、一个报文版本字段、一个路由域字段、一个地址类字段、一个路由标记字段以及一些路由信息项（一个 RIP 报文中最多允许 25 个路由信息项），其中每个字段后括号中的数字表示该字段所占的字节数。RIP 报文的最大长度为 $4+20*25=504$ 字节，加上 UDP 报头的 8 字节，一共是 512 字节。如果路由表的路由表项数目大于 25 时，那么就需要多个 RIP 报文来完成路由信息的传播过程。下面对报文字段进行逐一介绍：

- ◆ **命令字段：**表示 RIP 报文的类型，目前 RIP 只支持两种报文类型，分别是请求报文（request 1）和响应（response 2）报文。
- ◆ **版本字段：**表示 RIP 报文的版本信息，RIPv2 报文中此字段为 2。
- ◆ **路由域字段：**是一个选路守护程序的标识符，它指出了这个数据报的所有者。在一个 Unix 实现中，它可以是选路守护程序的进程号。该域允许管理者在单个路由器上运行多个 RIP 实例，每个实例在一个选路域内运行。
- ◆ **地址类字段：**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 地址字段：**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。

- ◆ **子网掩码字段：**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。
- ◆ **下一站 IP 地址字段：**下一驿站，可以对使用多路由协议的网络环境下的路由进行优化。
- ◆ **度量值字段：**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 发送的请求报文和响应报文都符合图 7.1 的报文结构格式，但是当需要发送请求对方路由器全部路由表信息的请求报文时，RIP 使用另一种报文结构，此报文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。

12.4.1. 3RIP 协议的基本特点

协议规定，RIP 协议使用 UDP 的 520 端口进行路由信息的交互，交互的 RIP 信息报文主要是两种类型：请求（request）报文和响应（response）报文。请求报文用来向相邻运行 RIP 的路由器请求路由信息，响应报文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，在实际报文中，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 报文，然后 RIP 协议就进入了循环等待状态，等待外部 RIP 协议报文（包括请求报文和响应报文）的到来；而接收到 request 报文的路由器则应当发出包含它们路由表信息的 response 报文。

当发出请求的路由器接收到一个 response 报文后，它会逐一处理收到的路由表项内容。如果报文中的表项为新的路由表项，那么就会向路由表加入该表项。如果该报文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。

此外，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。

12.4.1.4 水平分割

水平分割是一种避免路由环的出现和加快路由汇聚的技术。由于路由器可能收到它自己发送的路由信息，而这种信息是无用的，水平分割技术不反向通告任何从终端收到的路由更新信息，而只通告那些不会由于计数到无穷而清除的路由。

12.4.2 实验拓扑

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。

12.4.3 实验界面

界面上最多会有 4 个标签页：点击"Rip 报文展示"标签可以查看接收到的 RIP 报文；点击"路由表"标签可以查看/编辑本地路由表；点击"发送 RIP(接口 1)"标签可以封装 RIP 报文并从接口 1 发送；点击"发送 RIP(接口 2)"标签可以封装 RIP 报文并从接口 2 发送。其具体界面如下所示：

The screenshot shows the 'Rip 报文展示' (RIP Message Display) tab selected. The interface is divided into several sections:

- Header:** Contains input fields for 'Command' (value: 1) and 'Version' (value: 2).
- Route Entries With Data:** A table with the following structure:

Route Entries
Route Entry 1
- Configuration Fields:** On the right side, there are input fields for:
 - 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:** A section at the bottom showing hexadecimal data:

```
0000 01 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010 00 00 00 00 00 00 00 10
```

图 10.2 Rip 报文展示

Rip报文展示

路由表

目的地址	子网掩码	下一跳地址	跳跃计数	接口
50.0.0.0	255.255.255.0	20.0.0.2	2	2
40.0.0.0	255.255.255.0	10.0.0.2	3	1
30.0.0.0	255.255.255.0	20.0.0.2	5	2
20.0.0.0	255.255.255.0	0.0.0.0	1	2
10.0.0.0	255.255.255.0	0.0.0.0	1	1

删除

路由表项

目的地址

子网掩码

下一跳地址

50 , 0 , 0 , 0

255 , 255 , 255 , 0

20 , 0 , 0 , 2

跳跃计数

接口

2

2

添加

图 10.3 路由表

如果想删除一条路由信息，可以在表格中选中该条路由，然后点击删除按钮，即可删除该条路由；如果想添加一条路由信息，可以先点击添加按钮，然后在表格下面的信息编辑区域填入相关的路有信息；如果想修改一条路由信息，可以先在表格中选中该条路由，然后在信息编辑区域编辑相关的信息。

可以使用发送 RIP(接口 1)/ 发送 RIP(接口 2)封装相应的 RIP 报文，并将其以相应的接口发送出去。如图 10.4 和图 10.5 所示：

Rip报文展示 | 路由表 | 发送RIP(接口1) | 发送RIP(接口2)

Header
Command
Request
Version
2

Route Entries With Data
Route Entries
Add Remove

Address Family
0
Route Tag
0
IP
0 . 0 . 0 . 0
Subnet
0 . 0 . 0 . 0
Next Hop
0 . 0 . 0 . 0
Metric
0

Rip Header Encoding

0000 01 02 00 00

图 10.4 发送 RIP(接口 1)

Rip报文展示 | 路由表 | 发送RIP(接口1) | 发送RIP(接口2)

Header
Command
Request
Version
2

Route Entries With Data
Route Entries
Add Remove

Address Family
0
Route Tag
0
IP
0 . 0 . 0 . 0
Subnet
0 . 0 . 0 . 0
Next Hop
0 . 0 . 0 . 0
Metric
0

Rip Header Encoding

0000 01 02 00 00

图 10.5 发送 RIP(接口 2)

13 IPv6 协议交互实验

13.1 实验目的

现有的互联网是在 IPv4 协议的基础上运行。IPv6 是下一版本的互联网协议，它的提出最初是因为随着互联网的迅速发展，IPv4 定义的有限地址空间将被耗尽，地址空间的不足必将影响互联网的进一步发展。为了扩大地址空间，拟通过 IPv6 重新定义地址空间。IPv4 采用 32 位地址长度，只有大约 43 亿个地址，估计在 2005~2010 年间将被分配完毕，而 IPv6 采用 128 位地址长度，几乎可以不受限制地提供地址。

本实验通过设计实现主机协议栈中的 IPv6 协议，让学生深入了解网络层协议的基本原理，学习 IPv6 协议基本的分组接收和发送流程。

13.2 实验要求

根据计算机网络实验系统所提供的上下文，对给出的 IP v6 报文进行字段检查，如果出错指出其出错字段或组装一个正确的 IP v6 报头。

13.3 实验内容

1) 实现 IPv4 v6 分组的基本接收处理功能

对于接收到的 IP v6 分组，检查目的地址是否为本地地址，并检查 IP v6 分组头部中其它字段的合法性。在选项中选择对应的项。

2) 实现 IP v6 分组的封装发送

根据题干中给出的上下文环境，封装 IP v6 分组，使用系统提供的发送 IP v6 报文界面将分组发送出去。

13.4 实验帮助

界面上会显示接收到的 IPv6 报头(接收部分)或要填充的 IPv6 报头(发送部分),其内容展示分为字段意义显示和 16 进制显示,在字段意义显示部分可以看到接收到 IP v6 报头或要发送 IPv6 报头各个字段的数据值,在 16 进制显示部分会展示相应的 IPv6 报头的 16 进制显示。

Version	<input type="text" value="0"/>	Payload Length	<input type="text" value="0"/>
Traffic Class	<input type="text" value="0"/>	Next Header	<input type="text" value="0"/>
Flow Label	<input type="text" value="0"/>	Hop Limit	<input type="text" value="19"/>

Address	
Dest	<input type="text" value="00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00"/> <input type="button" value="Encode"/>
Address	
Source	<input type="text" value="00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00"/> <input type="button" value="Encode"/>

Header Encoding	
0010	00 00 00 00 00 00 00 00 13 00 00 00 00 00 00 00 0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0030 00 00 00 00 00 00 00 00

<input type="button" value="Decode"/>	<input type="button" value="Help"/>	<input type="button" value="Next"/>	<input type="button" value="Exit"/>
---------------------------------------	-------------------------------------	-------------------------------------	-------------------------------------

图 11.1 IPv6 报文发送界面

Version	<input type="text" value="5"/>	Payload Length	<input type="text" value="0"/>
Traffic Class	<input type="text" value="0"/>	Next Header	<input type="text" value="0"/>
Flow Label	<input type="text" value="0"/>	Hop Limit	<input type="text" value="16"/>

Address	
Dest	<input type="text" value="07:D1:0D:A8:00:BF:00:00:00:00:00:0A:00:00:03"/>
Address	
Source	<input type="text" value="07:D1:0D:A8:00:BF:00:00:00:00:00:0A:00:00:01"/>

Header Encoding	
0010	50 00 00 00 00 00 00 00 10 07 D1 0D A8 00 BF 00 00 P 0020 00 00 00 00 0A 00 00 01 07 D1 0D A8 00 BF 00 00 0030 00 00 00 00 0A 00 00 03

图 11.2 IPv6 报文接收界面

14 TCP 协议交互实验

14.1 实验目的

传输层是互联网协议栈的核心层次之一，它的任务是在源节点和目的节点间提供端到端的、高效的数据传输功能。TCP 协议是主要的传输层协议，它为两个任意处理速率的、使用不可靠 IP 连接的节点之间，提供了可靠的、具有流量控制和拥塞控制的、端到端的数据传输服务。TCP 协议不同于 IP 协议，它是有状态的，这也使其成为互联网协议栈中最复杂的协议之一。网络上多数的应用程序都是基于 TCP 协议的，如 HTTP、FTP 等。本实验的主要目的是学习和了解 TCP 协议的原理和设计实现的机制。

TCP 协议中的状态控制机制和拥塞控制算法是协议的核心部分。TCP 协议的复杂性主要源于它是一个有状态的协议，需要进行状态的维护和变迁。有限状态机可以很好的从逻辑上表示 TCP 协议的处理过程，理解和实现 TCP 协议状态机是本实验的重点内容。另外，由于在网络层不能保证分组顺序到达，因而在传输层要处理分组的乱序问题。只有在某个序号之前的所有分组都收到了，才能够将它们一起提交给应用层协议做进一步的处理。

本实验要求学生能够运用 TCP 报头正确的建立和拆除连接。

14.2 实验要求

根据计算机网络实验系统所提供的上下文，实现 TCP 建立连接和主动释放连接的过程。

14.3 实验内容

实验内容主要包括：

1) 实现 TCP 三次握手建立连接

根据题目中给出上下文环境，封装 TCP 报文，与目标主机建立起 TCP 连接。

2) 实现 TCP 主动释放连接

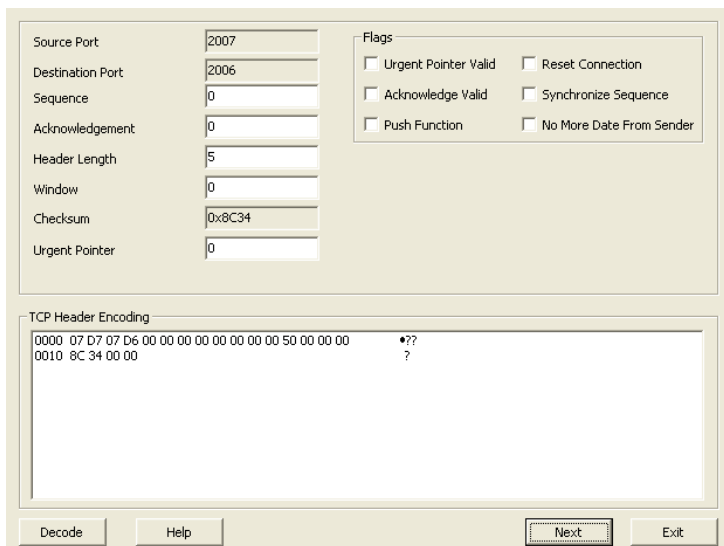
根据题目中给出的上下文环境，释放在上一步中建立起来的 TCP 连接。

14.4 实验帮助

界面上会显示接收到的 TCP 报头(接收部分)或要填充的 TCP 报头(发送部分),其内容展示分为字段意义显示和 16 进制显示,在字段意义显示部分可以看到接收到 TCP 报头或要发送 TCP 报头各个字段的数据值,在 16 进制显

示部分会展示相应的 TCP 报头的 16 进制显示。其中校验和会自动计算，源端口和目的端口已经确定不能更改。

学生需要注意的是，由于建立连接和释放连接具有时序关系，如果在实验过程中有一步出错就会直接显示最终实验结果——错误，而不会在错误的基础上继续下一步。所以学生需要在出错后重新开始本实验。



The interface is a window titled "TCP Header Encoding" with a light beige background. It contains several input fields and a section for flags.

Source Port	2007	Flags <input type="checkbox"/> Urgent Pointer Valid <input type="checkbox"/> Reset Connection <input type="checkbox"/> Acknowledge Valid <input type="checkbox"/> Synchronize Sequence <input type="checkbox"/> Push Function <input type="checkbox"/> No More Data From Sender
Destination Port	2006	
Sequence	0	
Acknowledgement	0	
Header Length	5	
Window	0	
Checksum	0x8C34	
Urgent Pointer	0	

Below the input fields is a section titled "TCP Header Encoding" containing a text area with the following hexadecimal data:

```

0000 07 D7 07 D6 00 00 00 00 00 00 50 00 00 00    •??
0010 8C 34 00 00    ?
  
```

At the bottom of the window are four buttons: "Decode", "Help", "Next", and "Exit". The "Next" button is highlighted with a dashed border.

图 12.1 TCP 报文发送界面

TCP报文展示
发送TCP报文

Source Port	<input type="text" value="2006"/>
Destination Port	<input type="text" value="2007"/>
Sequence	<input type="text" value="1"/>
Acknowledgement	<input type="text" value="1"/>
Header Length	<input type="text" value="5"/>
Window	<input type="text" value="1000"/>
Checksum	<input type="text" value="0x8838"/>
Urgent Pointer	<input type="text" value="0"/>

Flags

☐ Urgent Pointer Valid

☒ Acknowledge Valid

☐ Push Function

☐ Reset Connection

☒ Synchronize Sequence

☐ No More Data From Sender

TCP Header Encoding

```

0000 07 D6 07 D7 00 00 00 01 00 00 00 01 50 12 03 E8      •??
0010 88 38 00 00                                           ?
                    
```

图 12.2 TCP 报文接收界面

15 参考文献

- [1]. A. Tanenbaum (潘爱民译, 徐明伟审). 计算机网络 (第四版). 清华大学出版社, 2004 年. 第 5 章, 网络层.
- [2]. J. Postel. RFC791, Internet Protocol. 1981.
- [3]. R. Braden. RFC1122, Requirements for Internet Hosts – Communication Layers. 1989.
- [4]. D. Comer, D. Stevens. Internetworking with TCP/IP, Vol II: Design, Implementation, and Internals. Third Edition. Prentice-Hall Inc, 1999. (中文译本: 用 TCP/IP 进行网际互联, 第二卷: 设计、实现与内核, 第三版。电子工业出版社, 2001 年)
- [5]. S. Deering RFC2460, Internet Protocol, Version 6 (IPv6) Specification. 1998.
- [6]. C. Hedrick. RFC1058, Routing Information Protocol. 1988.
- [7]. G. Malkin. RFC1723, RIP Version 2. 1994.
- [8]. Jon Postel .RFC793, Transmission Control Protocol. 1981.
- [9]. J. Postel. RFC959, FILE TRANSFER PROTOCOL (FTP). 1985.
- [10]. Y. Rekhter RFC1771, A Border Gateway Protocol 4 (BGP-4) .