# 第十七章 丢包测量实验

丢包率是指一段时间内丢失的数据包数量占所发送数据组的比率。丢包率与网络环境、数据包长度以及包发送的频率等很多因素存在关联。较低的丢包率往往代表着较好的网络性能。

在实验一我们通过主动向目的主机发送ICMP回送请求报文的方法测量丢包率。在实验二我们从TCP流中解析目标主机反馈的ACK报文的关键字段来被动测量发送端的丢包率。

## 实验1：主动测量

### 一、实验目的

主动向目的主机发送指定数量的ICMP回送请求报文，统计接收到的回送回答报文数量从而计算丢包率。

### 二、实验基本原理

#### 1、 ICMP报文封装

ICMP回送请求报文是主机或路由器向一个特定的目的主机发送的询问。收到此报文的主机必须给源主机或路由器发送ICMP回送回答报文。利用这个协议，当我们发出响应的请求报文后，如果没有收到回答报文，就可以统计丢失的分组数量从而计算丢包率。ICMP不是高层协议，它只是封装在IP数据报中，作为其中的数据部分，是IP层的协议。ICMP报文格式如图17.1所示。

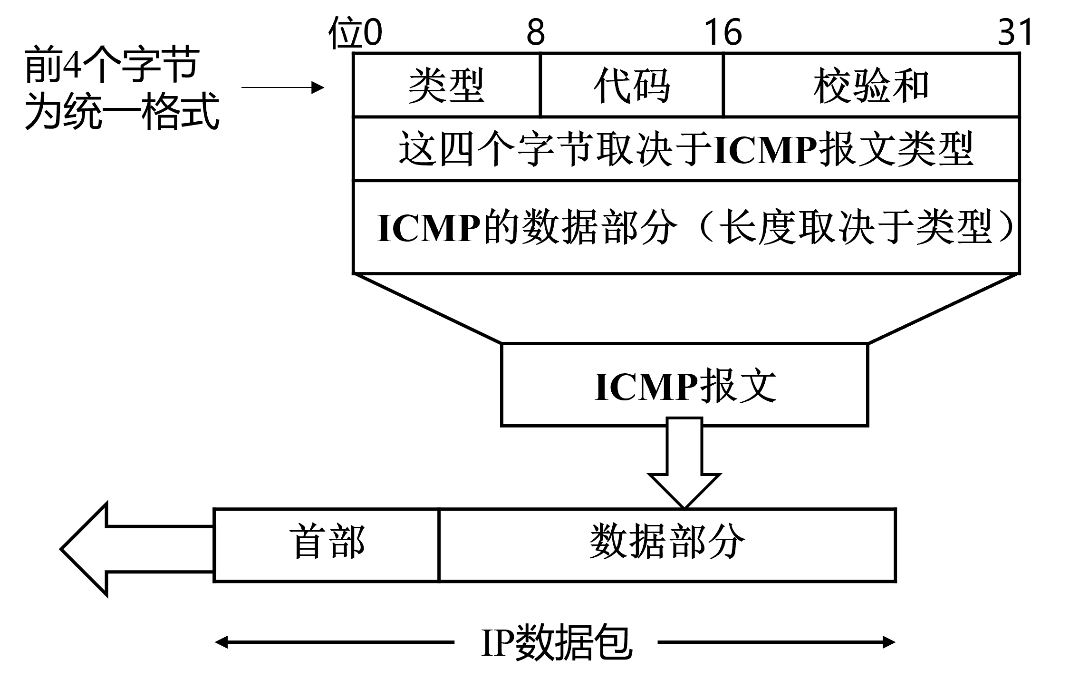


图17.1 ICMP报文格式

ICMP头部只有三个固定字段，其余部分因消息类型而异。固定字段有类型（type）、代码（code）、校验和（checksum）。ICMP报文的类型主要依靠type，code字段来区分。几种常见的ICMP报文类型如表17.1所示。在本实验中我们要用到的报文类型是回送请求与回送回答报文，它们对应的type以及code字段值如表17.2所示。回送报文除了固定字段，其余部分组织成3个字段：标识符（identifier），一般填写进程PID以区分其他进程；报文序列（sequence number）,用于为报文编号；数据（data），可以是任意数据。按ICMP协议规定，回显应答报文会原封不动地回传这些字段。

表17.1 ICMP报文类型

|  |  |  |
| --- | --- | --- |
| **ICMP报文种类** | **类型值** | **ICMP报文的类型** |
| 差错控制报文 | 3 | 终点不可达 |
| 11 | 时间超时 |
| 12 | 参数问题 |
| 5 | 改变路由 |
| 询问报文 | 8或0 | 回送请求或回答 |
| 13或14 | 时间戳请求或回答 |

表17.2 回显报文type code字段值

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **代码** |
| 回显请求 | 8 | 0 |
| 回显应答 | 0 | 0 |

#### 2、校验和的计算

ICMP报文校验和字段需要自行计算，当接收方接收到我们发送的ICMP报文后会按照同样的方法对校验和字段进行核对，如果核对错误则将该数据包丢弃。校验和字段计算步骤如下：

1. 将报文分成两个字节一组，如果总字节数为奇数，则在末尾追加一个零字节；
2. 对所有双字节进行按位求和；
3. 将高于16位的进位取出相加，直到没有进位；
4. 将校验和按位取反；

另外值得注意的是在开始计算前需将原报文校验和字段设为0，再按步骤计算。

### 三、实验步骤

下面，我们开始用C语言编写一个主动发送ICMP回显请求数据包并计算丢包率的程序。这里按照顺序列出了其中的关键步骤，源码为附录中的loss\_czhu.c文件。

#### 1、整体流程

实验一主要流程如图17.2所示。我们用单进程来实现报文的请求与接受。每隔一秒发送一个请求报文，此次请求结束后直到下一次发送请求报文，程序都一直处于接受状态。如果在超时时间内没有接收到回显应答报文，那么丢包个数自增一。程序尝试在超时时间内接受所有应答报文，之后会退出循环同时计算丢包率。

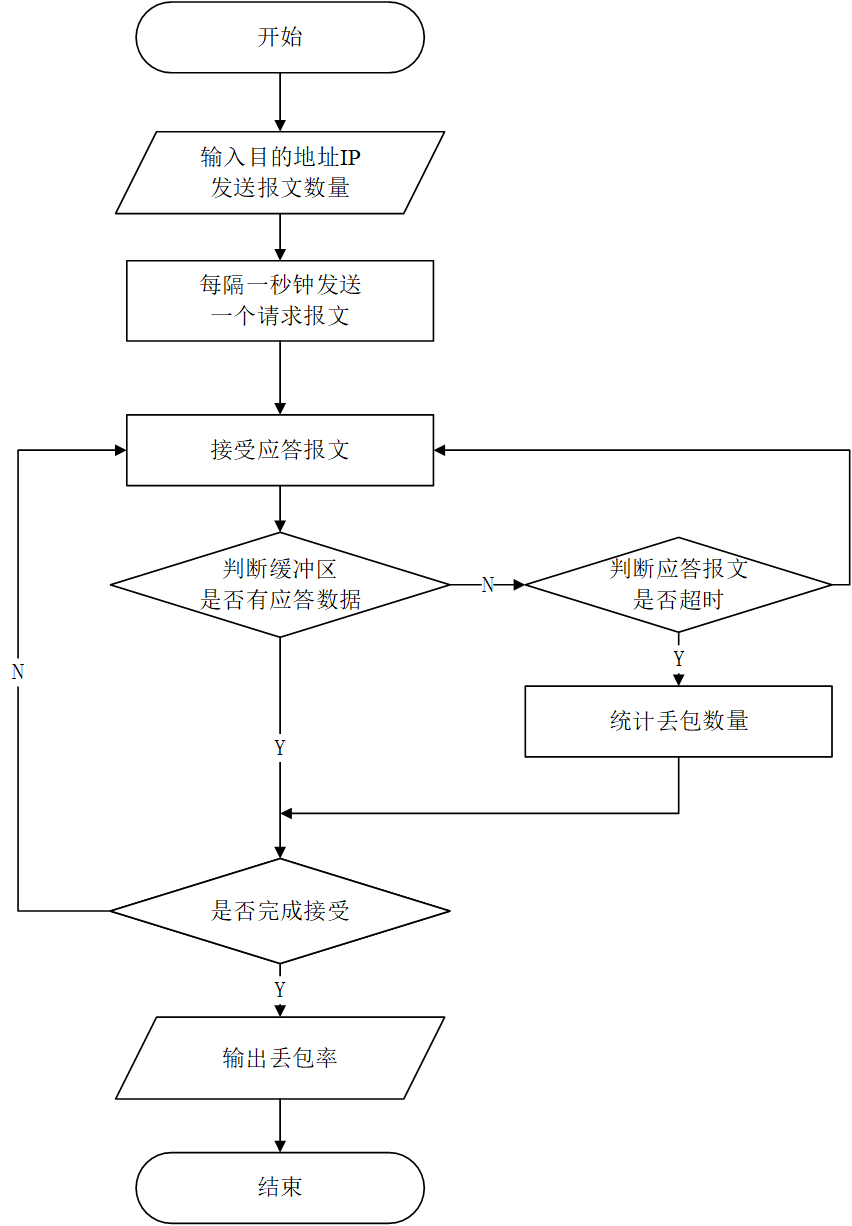


图17.2 实验一流程图

#### 2、定义ICMP回送请求报文结构体

将前面原理部分介绍的字段以代码形式进行定义，其中sending\_ts代表发送时间戳。

|  |
| --- |
| struct \_\_attribute\_\_((\_\_packed\_\_)) icmp\_echo  {  // header  uint8\_t type;  uint8\_t code;  uint16\_t checksum;  uint16\_t ident;  uint16\_t seq;  // data  double sending\_ts;  }; |

#### 3、时间戳与校验和的计算

其中unsigned char \*buffer为指向icmp\_echo结构体的unsigned char \*指针， bytes表示icmp\_echo结构体所占字节数。值得注意的是在定义icmp\_echo结构体时我们用\_\_attribute\_\_((\_\_packed\_\_))作为修饰，目的是阻止编译器对结构体进行对齐优化。

|  |
| --- |
| double get\_timestamp()  uint16\_t calculate\_checksum(unsigned char \*buffer, int bytes) |

#### 4、发送回送请求报文

函数send\_echo\_request用来实现发送回送请求报文的功能，其中的ident,seq分别代表进程号和序列号，我们用这两个参数来填充ICMP的相应字段，然后通过sendto系统调用来实现报文的发送。

|  |
| --- |
| int send\_echo\_request(int sock, struct sockaddr\_in \*addr, int ident, int seq)  {  ……      int bytes = sendto(sock, &icmp, sizeof(icmp), 0,                         (struct sockaddr \*)addr, sizeof(\*addr));  ……  } |

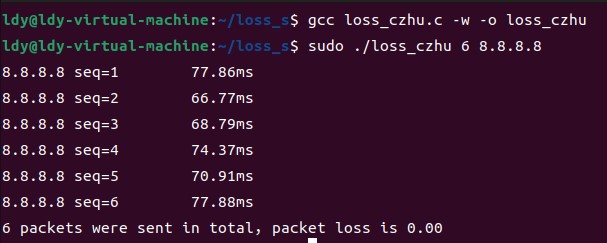
#### 5、接受回送应答报文

函数recv\_echo\_reply用来实现接受回送应答报文的功能，函数参数sock表示套接字标识符，ident表示进程号。函数返回值0表示接收缓冲器无数据可读或者未查找回送应答报文；函数返回值1表示正确接受应答报文；函数返回值-1表示在超时时间内没有接收到应答，应答报文丢失。

|  |
| --- |
| int recv\_echo\_reply(int sock, int ident)  {  ……  int bytes = recvfrom(sock, buffer, sizeof(buffer), 0,                           (struct sockaddr \*)&peer\_addr, &addr\_len);  if(缓冲区没有数据可读or未查找回送应答报文)  {  return 0；  }  eles if(超时时间内未收到应答)  {  return -1;  }  else{  printf();  return 1;  }  } |

### 四、实验案例

我们向IP为8.8.8.8的服务器发送了6个ICMP数据包，通过打印出的信息可以发现这六个数据包的响应数据包被按序接受，不存在丢失，丢包率为0。



## 实验2：被动测量

### 一、实验目的

从PCAP文件中提取TCP会话相应字段的信息，通过被动测量方法得到丢包率。

### 二、实验基本原理

TCP协议中的序列号用来标识源端向目的端发送的字节流，在没有丢包的情况下发送端会按照序列号递增的顺序发送数据报文。当发送端发送的序列号小于当前已经发送的最大序列号时，就证明之前发送的数据包或许因为丢失导致发送端重传。我们选择记录重传的数量来近似丢包的数量。当然这种近似是不准确的，原因在于数据报文的延迟到达和ACK报文的丢失都会导致冗余重传的发生。所以我们利用TCP协议中的SACK可选字段进一步统计冗余重传的数量，对丢包结果做更准确的估计。

#### 1、TCP固定头部格式如图17.3所示。

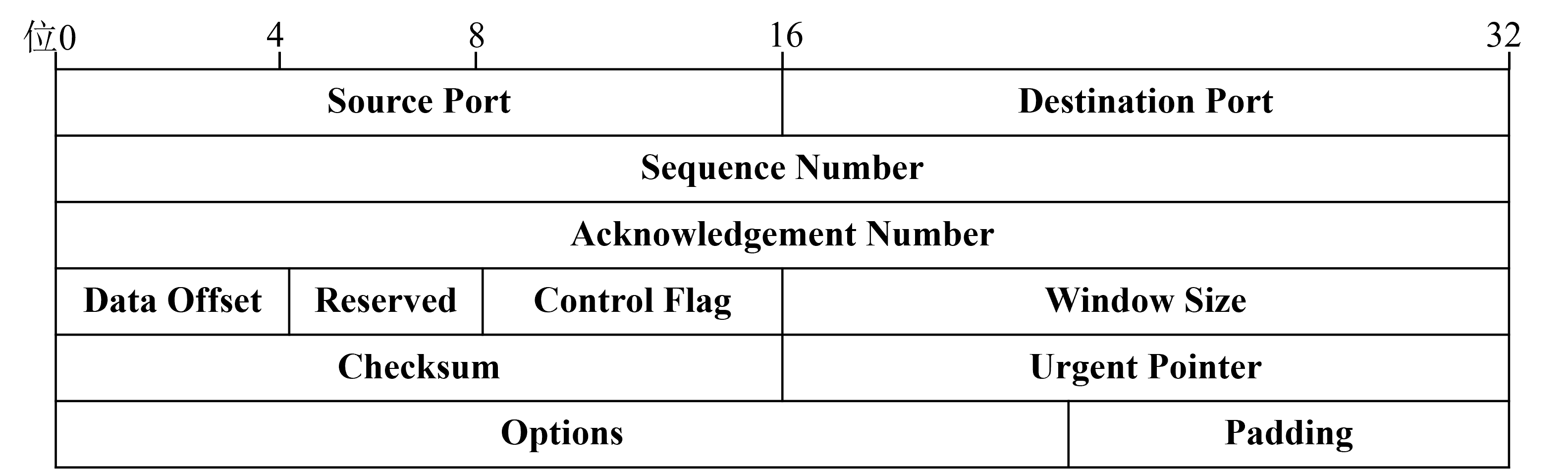


图 17.3 TCP固定头部格式

* Source Port(16bit): 发送方使用的端口号；
* Destination Port(16bit): 接收方使用的端口号；
* Sequence Number(32bit): 序列号，发送数据的位置。每发送一次 数据，就累加一次该数据字节数的大小；
* Acknowledgement Number(32bit): 确认应答号，是指下一次应该收到的数据的序列号；
* Data Offset(4bit): 该字段表示TCP所传输的数据部分应该从TCP包的哪个位开始计算；
* Control Flag(8bit): 控制标志也叫做控制位，每一位从左至右分别为CWR、ECE、URG、ACK、PSH、 RST、SYN、FIN;
* Options: 选项字段用于提高TCP的传输性能。

#### 2、Options可选项

由于前文已经详细讲解TCP header中固定字段的含义，所以本实验重点讲解TCP可选字段options。TCP头部的最后一个选项字段（options）是可变长的可选信息。因为TCP头部最长是60字节，所以除去TCP的20字节固定头部长度options字段最多包含40字节。典型的TCP头部选项结构如图17.4所示。

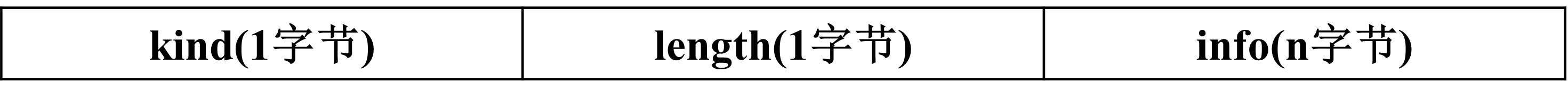


图 17.4 TCP头部选项

选项的第一个字段kind说明选项的类型，有的TCP选项没有后面两个字段，仅包含1字节的kind字段。第二个字段length指定该选项的总长度，该长度包括kind字段和length字段占据的2字节以及info的总字节数。第三个字段info是选项的具体信息，常见的TCP选项有7种，如图17.5所示。

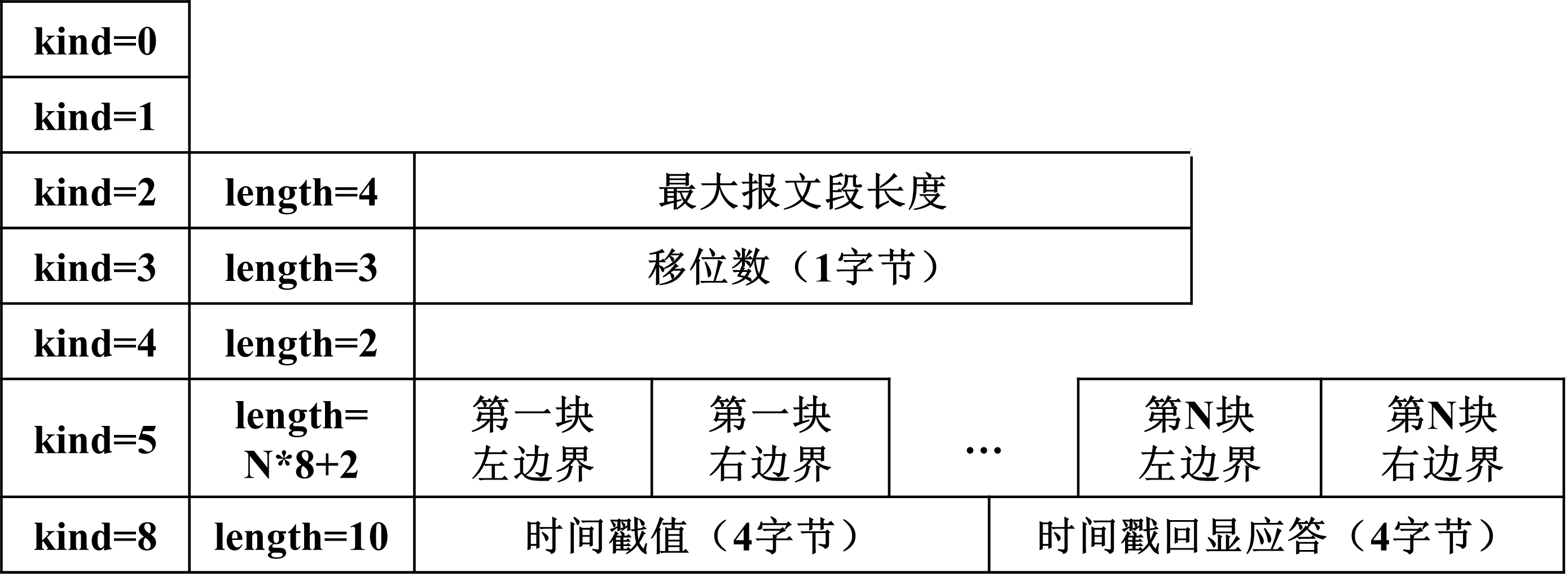


图17.5 TCP选项

在本实验中我们主要用到的是kind=4,kind=5的可选项。kind=4表示选择性确认（Selective Acknowledgment，SACK）选项。TCP通信时如果某个TCP报文段丢失，则TCP会重传最后被确认的TCP报文段后续的所有报文段，这样原先已经正确传输的TCP报文段也可能被重复发送，从而降低了TCP性能。SACK技术正是为改善这种情况而产生的，它使TCP只重新发送丢失的TCP报文段，而不用发送所有未被确认的TCP报文段。选择性确认选项用在连接初始化时，表示是否支持SACK技术。目前大部分TCP链接都默认支持SACK技术。

kind=5表示SACK实际工作的选项，该选项的参数告诉发送方本端已经收到并缓存的不连续的数据块，从而让发送端可以据此检查并重发丢失的数据块。每个块边沿（edge of block）参数包含一个4字节的序号。其中块左边沿表示已经收到的不连续块的第一个序号，而块右边沿则表示已经收到的不连续块的最后一个数据的序号的下一个序号。根据这些块信息，发送方就可以确定接收方具体没有收到的数据就是从ACK到最大SACK信息之间的那些空洞的序号。因为一个块信息占用8字节，所以TCP头部选项中实际上最多可以包含4个这样的不连续数据块。值得注意的是接收端总是会把最近一次接收到的数据块储存在靠近length字段的位置。

#### 3、DSACK

在收到重复报文的时候，SACK选项的第一个块(这个块也叫做DSACK块)可以用来传递这个重复报文的序列号，这个就是DSACK(duplicate-SACK)功能。这样允许TCP发送端根据SACK选项来推测不必要的重传。进而利用这些信息在乱序传输的环境中执行更健壮的操作。这个DSACK扩展是与原有的SACK选项的实现相互兼容的。DSACK的使用也不需要TCP连接的双方额外协商只要之前协商了SACK选项即可。

对于DSACK值得注意的有以下四点：

* 一个DSACK块只用来传递一个接收端最近接收到的重复报文的序列号，每个SACK选项中最多有一个DSACK块。
* 接收端每个重复包最多在一个DSACK块中上报一次。如果接收端依次发送了两个带有相同DSACK块信息的ACK报文，则表示接收端接收了两次重复包。
* 和普通的SACK块一样，DSACK块左边指定重复包的第一个字节的序列号，右边指定重复包最后一个字节的下一个序列号。
* 如果收到重复报文，第一个SACK块应该指定重复报文的序列号 (这个SACK块也叫做DSACK块)。如果这个重复报文是一个大的不连续块的一部分，那么接下来的这个SACK块应该指定这个大的不连续块，额外的SACK块应该按照RFC2018指定的顺序排列。

当发送端接收到SACK报文的时候，要将第一个SACK块与这个ACK报文的ack number比较，如果小于等于ack number则说明是DSACK块，如果大于ack number则应该与第二个SACK块比较，如果第二个SACK块包含第一个SACK块，则说明第一个SACK块为DSACK块，如果上面两个条件都不满足说明第一个SACK块是普通的SACK块。

#### 4、计算丢包率

每当发送端发送一个数据报文，我们都去维护一个代表当前最大序列号的变量。发送端在正常情况下会按照TCP序列号的从小到大的顺序发送数据报文，当发送端发生重传时当前发送的报文序列号一定是小于当前最大的序列号，我们用retransmits变量去记录发送端重传的数量。然而重传并不表示前一个发送的数据包一定丢失，所以单单用retransmits去代表丢失的数据包的数量是不那么准确的。原理部分我们讲到了DSACK，它用来传递一个接收端最近接收到的重复报文的序列号。如果发送端接收到了反馈回来的DSACK，那么我们就会知道DSACK所指示的那个数据包一定发生了重传。我们用dup\_xmits变量来记录重传的冗余个数，由此我们估计的丢包个数=retransmits-dup\_xmits。

### 三、实验步骤

下面，我们用 C 语言编写一个从pcap文件读取关键参数并且统计发送端丢包率的程序。这里按照顺序列出了其中的关键步骤，源码为附录中的 loss\_cbei.c 文件。

#### 1、整体流程

实验二主要流程如图17.6所示。首先程序读入pcap文件并且从每一个packet中提取关键字段信息，主要包括五元组、序列号、确认号、SACK块等关键信息。当程序读到的packet是发送端发送的数据报文，则去维护一个代表当前最大序列号的变量以及发送端发生重传的数量信息。如果程序读到的packet是发送端接收到的ACK报文，则根据是否存在DSACK字段去统计冗余重传的数量。程序循环结束后，通过公式丢包个数=重传数量-冗余重传数量，来估计丢包个数并且计算丢包率。

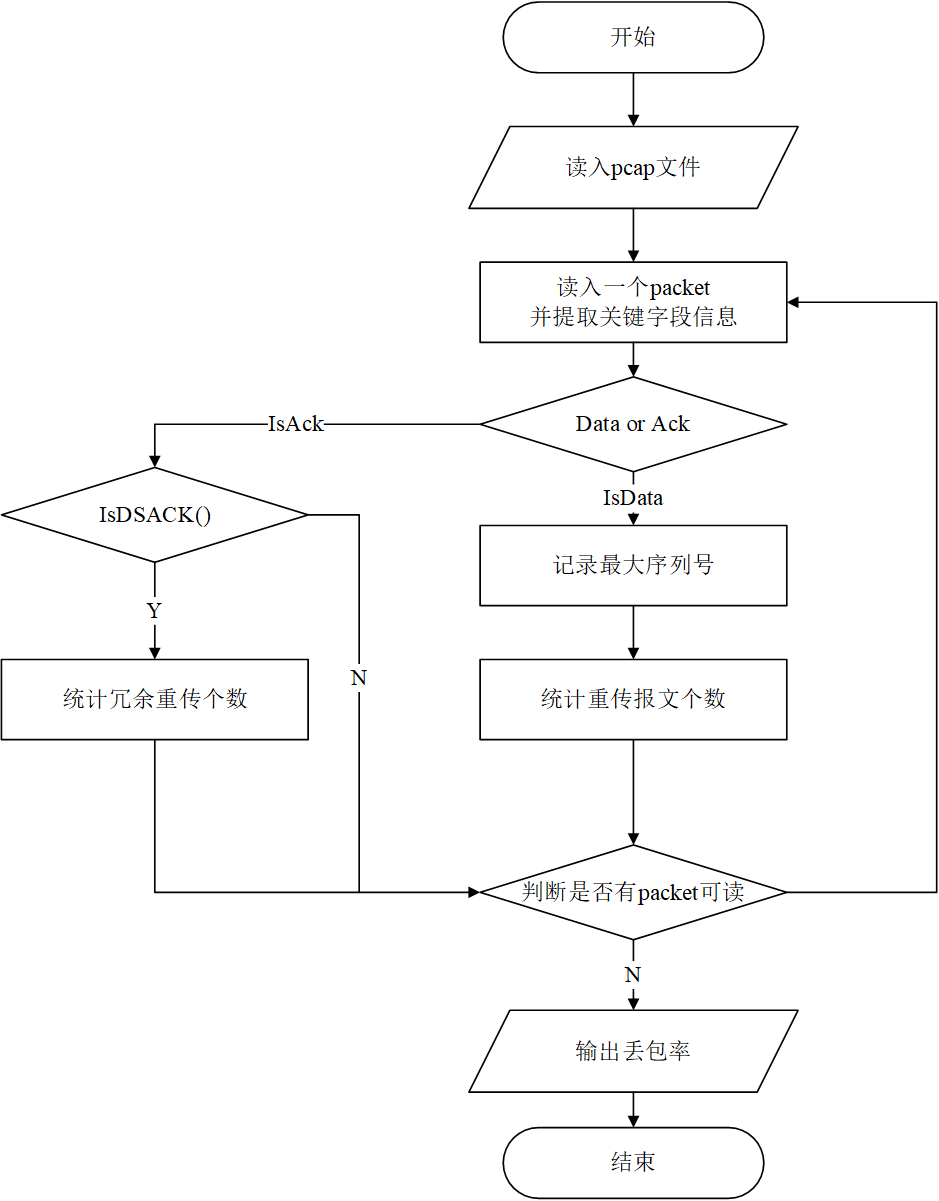


图17.6实验二流程图

#### 2、定义Quintet结构体

Quintet结构体主要储存了报文的关键信息。我们利用TCP流五元组来判断当前报文是否属于我们所要提取的TCP流。序号、确认号、SACK blocks等字段用于后续丢包率的计算。

|  |
| --- |
| typedef struct Sack\_Edges  {      u\_int32 ledge; // SACK block左边界      u\_int32 redge; // SACK block右边界  } Sack\_Edges;  typedef struct \_\_attribute\_\_((\_\_packed\_\_)) tcp\_Tuple  {  u\_int32 SrcIP;   //源 IP 地址      u\_int32 DstIP;   //目的 IP 地址      u\_short SrcPort; //源端口号 16bit      u\_short DstPort; //目的端口号 16bit      u\_int8 Protocol; //协议类型  } tcp\_Tuple;  typedef struct Quintet  {      tcp\_Tuple TcpTuple; // TCP流五元组      u\_int32 SeqNum;     //序号      u\_int32 AckNum;     //确认号      u\_int16 payload;    //负载数据大小      u\_int8 Edges\_Num;   // SACK blocks个数      Sack\_Edges \*Edges;  // SACK blocks  } Quintet; |

#### 3、确定报文类别

通过前面提到的TCP五元组来判断当前报文是属于TCP流中的发送报文还是接收报文。

|  |
| --- |
| u\_int8 IsData(tcp\_Tuple s1, tcp\_Tuple s2)  u\_int8 IsAck(tcp\_Tuple s1, tcp\_Tuple s2) |

#### 4、IsDSACK

当发送端接收到SACK报文的时候，我们需要判断SACK字段的第一个block是否为DSACK块。当第一个block的左边界小于等于ack number则说明它是DSACK块，如果大于ack number则应该与第二个SACK块比较，如果第二个SACK块包含第一个SACK块，则说明第一个SACK块为DSACK块。

|  |
| --- |
| if (quintet->Edges != NULL)  {      Sack\_Edges \*p = quintet->Edges;      if (ntohl(p->ledge) < quintet->AckNum || (quintet->Edges\_Num >= 2 && ntohl((p + 1)->ledge) <= ntohl(p->ledge) && ntohl((p + 1)->redge) >= ntohl(p->redge)))      {          dup\_xmits += 1;      }  } |

#### 5、丢包率的计算

其中highdata代表发送端发送的最大的序列号，retransmits表示发送端的重传个数，dup\_xmits代表发送端的冗余重传个数。我们遍历pcap文件中每一个报文，如果该报文为TCP流中发送端发送的数据报文，则维护当前发送的最大的序列号，当发送的序列号小于highdata， retransmits 自增一。如果该报文为TCP流中发送端接受到的ACK报文，则判断当前报文是否携带DSACK字段信息，如果携带，那么dup\_xmits自增一。循环结束，丢包个数就等于重传个数减去冗余重传个数。

|  |
| --- |
| highdata = retransmits = dup\_xmits = 0  for pkt in snd\_trace:      if pkt.IsData ():          if pkt.SeqNo () > highdata:              highdata = pkt.SeqNo ()          else:              retransmits += 1      if pkt.IsACK ():          if pkt.IsDSACK ():              dup\_xmits += 1  loss=(retransmits - dup\_xmits) / total\_data |

### 四、实验案例

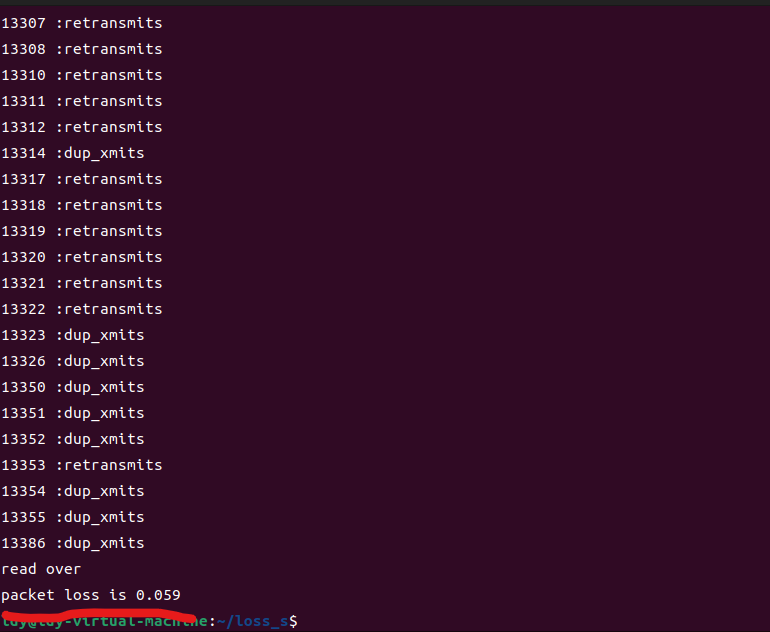
#### 运行过程：

我们要计算的是某一条TCP流的丢包率，运行程序时需要传入的参数有五个，分别为pcap文件路径 、发送端IP 、发送端端口号、接收端IP、接收端端口号 。



#### 运行结果：

我们使用的实验数据是用户上传视频的TCP流信息，在终端上打印出了重传以及冗余重传的报文编号以及最后估计出来的丢包率5.59%。感兴趣的同学可以对比pcap文件找到相应的packet作进一步分析。



## 附录

### 源码

#### 实验一：loss\_czhu.c

|  |
| --- |
| **/\* loss\_czhu.c**  **Experiment 1: actively measure packet loss rate**  **To compile:**  **>gcc loss\_czhu.c -o loss\_czhu**  **To run:**  **>sudo ./loss\_czhu 10 8.8.8.8**  **\*/**  **#include <arpa/inet.h>**  **#include <errno.h>**  **#include <stdint.h>**  **#include <stdio.h>**  **#include <sys/time.h>**  **#include <unistd.h>**  **#include <stdlib.h>**  **#define IP\_BUFFER\_SIZE 65536**  **#define RECV\_TIMEOUT\_USEC 100000**  **//封装的报文格式**  **//其中前三个字段为ICMP公共头部**  **//中间两个字段为回显请求、回显应答惯用头部**  **//剩余部分为数据负载，包括一个双精度发送时间戳**  **struct \_\_attribute\_\_((\_\_packed\_\_)) icmp\_echo**  **{**  **// header**  **uint8\_t type;**  **uint8\_t code;**  **uint16\_t checksum;**  **uint16\_t ident;**  **uint16\_t seq;**  **// data**  **double sending\_ts;**  **};**  **//获得当前系统时间**  **double get\_timestamp()**  **{**  **struct timeval tv;**  **gettimeofday(&tv, NULL);**  **return tv.tv\_sec + ((double)tv.tv\_usec) / 1000000;**  **}**  **//计算ICMP报文校验和**  **uint16\_t calculate\_checksum(unsigned char \*buffer, int bytes)**  **{**  **uint32\_t checksum = 0;**  **unsigned char \*end = buffer + bytes;**  **// 如果总字节数为奇数，则在末尾追加一个零字节**  **if (bytes % 2 == 1)**  **{**  **end = buffer + bytes - 1;**  **checksum += (\*end) << 8;**  **}**  **// 对所有双字节进行按位求和**  **while (buffer < end)**  **{**  **checksum += (buffer[0] << 8) + buffer[1];**  **// 将高于16位的进位取出相加，直到没有进位**  **uint32\_t carray = checksum >> 16;**  **if (carray != 0)**  **{**  **checksum = (checksum & 0xffff) + carray;**  **}**  **buffer += 2;**  **}**  **// 将校验和按位取反**  **checksum = ~checksum;**  **return checksum & 0xffff;**  **}**  **int send\_echo\_request(int sock, struct sockaddr\_in \*addr, int ident, int seq)**  **{**  **//填充相应字段信息**  **struct icmp\_echo icmp;**  **bzero(&icmp, sizeof(icmp));**  **icmp.type = 8;**  **icmp.code = 0;**  **icmp.ident = htons(ident);**  **icmp.seq = htons(seq);**  **icmp.sending\_ts = get\_timestamp();**  **icmp.checksum = htons(**  **calculate\_checksum((unsigned char \*)&icmp, sizeof(icmp)));**  **// 发送数据报**  **int bytes = sendto(sock, &icmp, sizeof(icmp), 0,**  **(struct sockaddr \*)addr, sizeof(\*addr));**  **if (bytes == -1)**  **{**  **return -1;**  **}**  **return 0;**  **}**  **int recv\_echo\_reply(int sock, int ident)**  **{**  **// 分配缓冲区，按 IP 包的最大长度来准备**  **unsigned char buffer[IP\_BUFFER\_SIZE];**  **struct sockaddr\_in peer\_addr;**  **// 接受回显回答报文**  **int addr\_len = sizeof(peer\_addr);**  **int bytes = recvfrom(sock, buffer, sizeof(buffer), 0,**  **(struct sockaddr \*)&peer\_addr, &addr\_len);**  **if (bytes == -1)**  **{**  **//缓冲区没有数据可读返回0**  **if (errno == EAGAIN || errno == EWOULDBLOCK)**  **{**  **return 0;**  **}**  **return -1;**  **}**  **int ip\_header\_len = (buffer[0] & 0xf) << 2;**  **// 在接收到的数据包中查找ICMP数据包**  **struct icmp\_echo \*icmp = (struct icmp\_echo \*)(buffer + ip\_header\_len);**  **if (icmp->type != 0 || icmp->code != 0)**  **{**  **return 0;**  **}**  **if (ntohs(icmp->ident) != ident)**  **{**  **return 0;**  **}**  **printf("%s seq=%-5d %8.2fms\n",**  **inet\_ntoa(peer\_addr.sin\_addr),**  **ntohs(icmp->seq),**  **(get\_timestamp() - icmp->sending\_ts) \* 1000);**  **return 1;**  **}**  **int ping(const char \*num, const char \*ip)**  **{**  **// 储存目的ip地址**  **struct sockaddr\_in addr;**  **bzero(&addr, sizeof(addr));**  **addr.sin\_family = AF\_INET;**  **addr.sin\_port = 0;**  **if (inet\_aton(ip, (struct in\_addr \*)&addr.sin\_addr.s\_addr) == 0)**  **{**  **fprintf(stderr, "bad ip address: %s\n", ip);**  **return -1;**  **};**  **// 创建原始套接字**  **int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP);**  **if (sock == -1)**  **{**  **perror("create raw socket");**  **return -1;**  **}**  **// 设置套接字超时时长**  **struct timeval tv;**  **tv.tv\_sec = 0;**  **tv.tv\_usec = RECV\_TIMEOUT\_USEC;**  **int ret = setsockopt(sock, SOL\_SOCKET, SO\_RCVTIMEO, &tv, sizeof(tv));**  **if (ret == -1)**  **{**  **perror("set socket option");**  **close(sock);**  **return -1;**  **}**  **double next\_ts = get\_timestamp();**  **int ident = getpid();**  **int seq = 1;**  **int loss\_num = 0;**  **int num\_ = atoi(num);**  **int i = 0;**  **for (;;)**  **{**  **if (i >= num\_)**  **break;**  **// 每隔一秒钟发送一个ICMP数据包**  **double current\_ts = get\_timestamp();**  **if (current\_ts >= next\_ts)**  **{**  **ret = send\_echo\_request(sock, &addr, ident, seq);**  **if (ret == -1)**  **{**  **perror("Send failed");**  **}**  **next\_ts = current\_ts + 1;**  **seq += 1;**  **}**  **//尝试去接受回显回答请求**  **ret = recv\_echo\_reply(sock, ident);**  **if (ret == -1)**  **{**  **i++;**  **perror("Receive failed");**  **loss\_num++;**  **}**  **if (ret == 1)**  **{**  **i++;**  **}**  **}**  **printf("%d packets were sent in total，packet loss is %0.2f\n", num\_, loss\_num / (double)num\_);**  **close(sock);**  **return 0;**  **}**  **int main(int argc, const char \*argv[])**  **{**  **if (argc < 2)**  **{**  **fprintf(stderr, "no host specified");**  **return -1;**  **}**  **return ping(argv[1], argv[2]);**  **}** |

#### 实验二：loss\_cbei.c

|  |
| --- |
| **/\* loss\_cbei.c**  **Experiment 1: passively measure packet loss rate**  **To compile:**  **>gcc loss\_cbei.c -o loss\_cbei**  **To run:**  **>sudo ./loss\_czhu pcap文件路径 发送端IP 发送端Port 接收端IP 接收端Port**  **\*/**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <netinet/in.h>**  **#include <arpa/inet.h>**  **typedef u\_int32\_t bpf\_u\_int32;**  **typedef u\_int16\_t u\_short;**  **typedef u\_int32\_t u\_int32;**  **typedef u\_int16\_t u\_int16;**  **typedef u\_int8\_t u\_int8;**  **char tempSrcIp[256]; //存储转化后的字符地址。**  **char tempDstIp[256];**  **//时间戳\*/**  **struct time\_val**  **{**  **int tv\_sec;**  **int tv\_usec;**  **};**  **// pcap 数据包头结构体**  **typedef struct Packet\_header**  **{**  **struct time\_val ts; //时间戳**  **bpf\_u\_int32 caplen; //数据长度**  **bpf\_u\_int32 len;    //离线数据长度**  **} Packet\_header;**  **// IP 数据报头 固定20 字节**  **typedef struct IP\_header\_stable**  **{**  **u\_int8 Ver\_HLen;      //版本+报头长度**  **u\_int8 TOS;           //服务类型**  **u\_int16 TotalLen;     //总长度**  **u\_int16 ID;           //标识**  **u\_int16 Flag\_Segment; //标志+片偏移**  **u\_int8 TTL;           //生存周期**  **u\_int8 Protocol;      //协议类型**  **u\_int16 Checksum;     //头部校验和**  **u\_int32 SrcIP;        //源 IP 地址**  **u\_int32 DstIP;        //目的 IP 地址**  **} IP\_header\_stable;**  **// TCP/UDP 数据**  **typedef struct TCP\_header\_stable**  **{**  **u\_short SrcPort;        //源端口号 16bit**  **u\_short DstPort;        //目的端口号 16bit**  **u\_int32 SeqNum;         //序列号 32bit**  **u\_int32 AckNum;         //确认号 32bit**  **u\_int16 Offset\_re\_Sign; //数据偏移,保留位，标志位 4+6+6 bit**  **u\_int16 windows;        //窗口大小 16bit**  **u\_int16 checksum;       //校验和 16bit**  **u\_int16 urgent\_point;   //紧急指针 16bit**  **} TCP\_header\_stable;**  **typedef struct Sack\_Edges**  **{**  **u\_int32 ledge; // SACK block左边界**  **u\_int32 redge; // SACK block右边界**  **} Sack\_Edges;**  **typedef struct \_\_attribute\_\_((\_\_packed\_\_)) tcp\_Tuple**  **{**  **u\_int32 SrcIP;   //源 IP 地址**  **u\_int32 DstIP;   //目的 IP 地址**  **u\_short SrcPort; //源端口号 16bit**  **u\_short DstPort; //目的端口号 16bit**  **u\_int8 Protocol; //协议类型**  **} tcp\_Tuple;**  **typedef struct Quintet**  **{**  **tcp\_Tuple TcpTuple; // TCP流五元组**  **u\_int32 SeqNum;     //序号**  **u\_int32 AckNum;     //确认号**  **u\_int16 payload;    //负载数据大小**  **u\_int8 Edges\_Num;   // SACK blocks个数**  **Sack\_Edges \*Edges;  // SACK blocks**  **} Quintet;**  **//判断packet是否为发送方发送的数据包**  **u\_int8 IsData(tcp\_Tuple s1, tcp\_Tuple s2)**  **{**  **if (s1.SrcIP == s2.SrcIP && s1.DstIP == s2.DstIP && s1.DstPort == s2.DstPort && s1.SrcPort == s2.SrcPort && s1.Protocol == s2.Protocol)**  **{**  **return 1;**  **}**  **return 0;**  **}**  **//判断packet是否为发送方接收的数据包**  **u\_int8 IsAck(tcp\_Tuple s1, tcp\_Tuple s2)**  **{**  **if (s1.SrcIP == s2.DstIP && s1.DstIP == s2.SrcIP && s1.DstPort == s2.SrcPort && s1.SrcPort == s2.DstPort && s1.Protocol == s2.Protocol)**  **{**  **return 1;**  **}**  **return 0;**  **}**  **/\*argv[1]:pcap文件路径；**  **argv[2]:发送端IP;**  **argv[3]:发送端端口号；**  **argv[4]:接收端IP；**  **argv[5]:接收端端口号\*/**  **int main(int argc, char \*\*argv)**  **{**  **Packet\_header \*packet\_header = NULL;**  **IP\_header\_stable \*ip\_header = NULL;**  **TCP\_header\_stable \*tcp\_header\_stable = NULL;**  **Quintet \*quintet = NULL;**  **Sack\_Edges \*edges = NULL;**  **tcp\_Tuple stream;**  **u\_int32 highdata = 0;    //记录当前最高的序列号**  **u\_int32 retransmits = 0; //记录发送端重传数量**  **u\_int32 dup\_xmits = 0;   //记录发送端重传的冗余数量**  **//初始化，分配内存**  **stream.SrcIP = inet\_addr(argv[2]);**  **stream.SrcPort = atoi(argv[3]);**  **stream.DstIP = inet\_addr(argv[4]);**  **stream.DstPort = atoi(argv[5]);**  **stream.Protocol = 6;**  **packet\_header = (Packet\_header \*)malloc(sizeof(Packet\_header));**  **ip\_header = (IP\_header\_stable \*)malloc(sizeof(IP\_header\_stable));**  **tcp\_header\_stable = (TCP\_header\_stable \*)malloc(sizeof(TCP\_header\_stable));**  **quintet = (Quintet \*)malloc(sizeof(Quintet));**  **//用来提取字段的 PCAP 文件**  **FILE \*pcap\_file = fopen(argv[1], "r");**  **if (pcap\_file == 0)**  **{**  **printf("打开文件失败！");**  **return 0;**  **}**  **long int pkt\_offset; // PCAP 文件指针**  **pkt\_offset = 24;     // PCAP 文件头结构占 24 字节**  **int total\_data = 0;  //记录发送端发送的携带负载数据的数据包个数**  **int i = 0;**  **while (1)**  **{**  **i++;**  **if (fseek(pcap\_file, pkt\_offset, SEEK\_SET) < 0)**  **break; //移动 PCAP 文件指针位置，跳过文件头.**  **if (fread(packet\_header, 16, 1, pcap\_file) != 1)**  **{**  **printf("read over\n");**  **break;**  **}**  **int len = packet\_header->caplen; //每个数据包数据的长度**  **pkt\_offset += 16 + len;**  **fseek(pcap\_file, 14, SEEK\_CUR); //数据包帧头占 14 字节**  **if (fread(ip\_header, sizeof(IP\_header\_stable), 1, pcap\_file) != 1)**  **{**  **printf("%d: can not read ip\_header\n", i);**  **break;**  **}**  **int ip\_varlen = (ip\_header->Ver\_HLen & 0x0f) \* 4 - 20; //计算IP数据包头部可变长度**  **if (ip\_header->Protocol != 6)**  **continue;**  **fseek(pcap\_file, ip\_varlen, SEEK\_CUR);**  **if (fread(tcp\_header\_stable, sizeof(TCP\_header\_stable), 1, pcap\_file) != 1)**  **{**  **printf("%d: can not read tcp\_header\_stable\n", i);**  **break;**  **}**  **u\_int8 tcp\_len = (ntohs(tcp\_header\_stable->Offset\_re\_Sign) >> 12) \* 4; // tcp header长度**  **quintet->TcpTuple.SrcIP = ip\_header->SrcIP;**  **quintet->TcpTuple.DstIP = ip\_header->DstIP;**  **quintet->TcpTuple.Protocol = ip\_header->Protocol;**  **quintet->TcpTuple.SrcPort = ntohs(tcp\_header\_stable->SrcPort);**  **quintet->TcpTuple.DstPort = ntohs(tcp\_header\_stable->DstPort);**  **quintet->AckNum = ntohl(tcp\_header\_stable->AckNum);**  **quintet->SeqNum = ntohl(tcp\_header\_stable->SeqNum);**  **quintet->payload = ntohs(ip\_header->TotalLen) - (ip\_header->Ver\_HLen & 0x0f) \* 4 - tcp\_len; // tcp 数据负载大小**  **quintet->Edges = NULL;**  **//判断当前报文是否为发送端发送的携带数据的报文**  **if (IsData(quintet->TcpTuple, stream) && quintet->payload != 0)**  **{**  **total\_data++;**  **if (quintet->SeqNum > highdata)**  **{**  **highdata = quintet->SeqNum;**  **}**  **else**  **{**  **retransmits += 1;**  **printf("%d :retransmits\n", i);**  **}**  **continue;**  **}**  **if (IsAck(quintet->TcpTuple, stream))**  **{**  **int fpoint = 20;**  **//提取tcp option字段中的SACK块**  **while (fpoint < tcp\_len)**  **{**  **u\_int8 kind = fgetc(pcap\_file);**  **u\_int8 length;**  **if (kind == 0 || kind == 1)**  **{**  **fpoint += 1;**  **continue;**  **}**  **else if (kind != 5)**  **{**  **length = fgetc(pcap\_file);**  **fseek(pcap\_file, length - 2, SEEK\_CUR);**  **fpoint += length;**  **continue;**  **}**  **else**  **{**  **length = fgetc(pcap\_file);**  **quintet->Edges\_Num = (length - 2) / 8;**  **quintet->Edges = (Sack\_Edges \*)malloc(quintet->Edges\_Num \* sizeof(Sack\_Edges));**  **fread(quintet->Edges, sizeof(Sack\_Edges) \* quintet->Edges\_Num, 1, pcap\_file);**  **break;**  **}**  **}**  **//如果存在SACK则判断发送端重传的数据是否存在冗余**  **if (quintet->Edges != NULL)**  **{**  **Sack\_Edges \*p = quintet->Edges;**  **if (ntohl(p->ledge) < quintet->AckNum || (quintet->Edges\_Num>=2 && ntohl((p + 1)->ledge) <= ntohl(p->ledge) && ntohl((p + 1)->redge) >= ntohl(p->redge)))**  **{**  **dup\_xmits += 1;**  **printf("%d :dup\_xmits\n", i);**  **}**  **free(quintet->Edges);**  **}**  **}**  **}**  **//计算丢包率**  **printf("packet loss is %.3f\n", (retransmits - dup\_xmits) / (double)total\_data);**  **fclose(pcap\_file);**  **return 0;**  **}** |