第十七章 丢包测量实验

丢包率是指测试中丢失的数据包数量占所发送数据组的比率。丢包率与数据包长度、包发送的频率以及网络环境等很多因素存在关联。较低的丢包率往往代表着不错的网络性能。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。

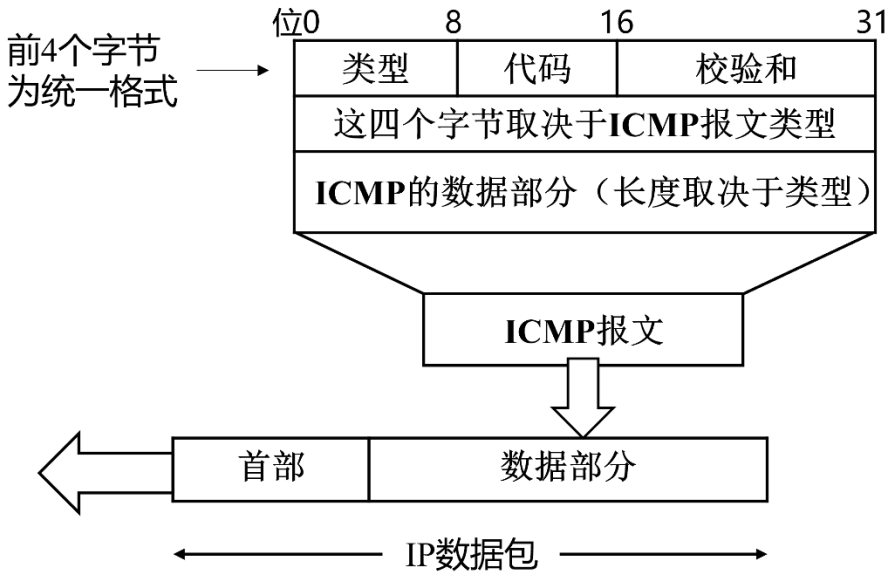


图 17.1 ICMP 报文格式

注意到，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**错误!未找到引用源。**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；

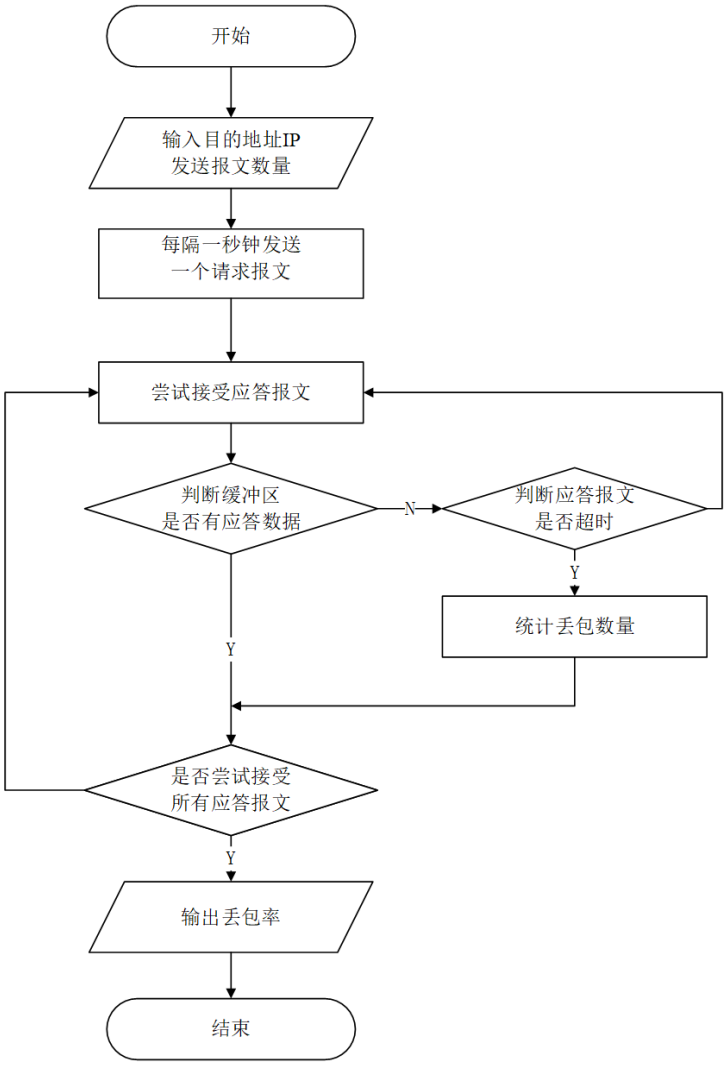
另外值得注意的是在开始计算前需将原报文校验和字段设为 0，再按步骤计算。

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

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，然后会退出循环同时计算丢包率。



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

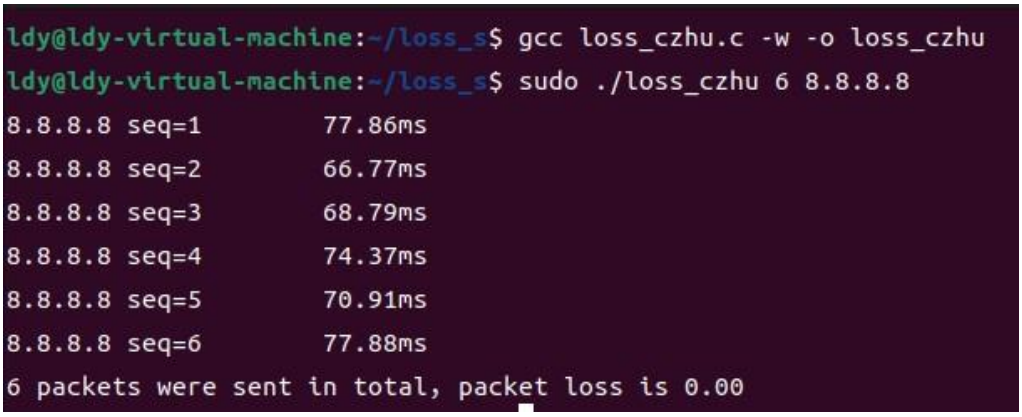
函数 recv_echo_reply 用来实现接受回送应答报文的功能，函数参数 sock 表示套接字标识符，ident 表示进程号。函数返回值 0 表示接收缓冲器无数据可读或者未查找回送应答报文；函数返回值 1 表示正确接受应答报文；函数返回值-1 表示在超时时间内没有接收到应答，应答报文丢失。

```
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，被动计算发送端的丢包率。

二、实验基本原理

1、TCP 固定头部格式如图 17.2 所示。

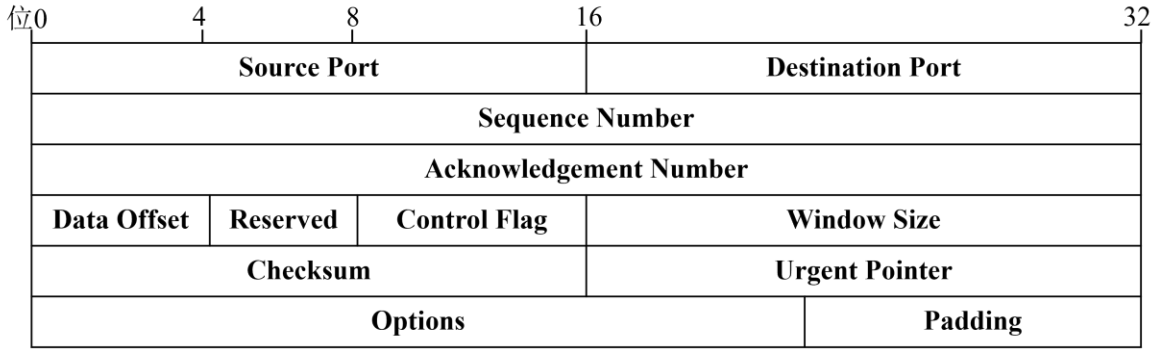


图 17.2 TCP 固定头部格式

- 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；
- Reserved(4bit): 保留字段，该字段主要是为了以后扩展时使用;
- 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;
- Window Size(8bit): 用于通知从相同 TCP 首部的确认应答号所指位置开始能够接收的数据大小;
- Checksum(16bit): 对 TCP 的伪首部、首部和数据部分进行错误校验;
- Urgent Pointer(16 bit): 该字段的数值表示本报文段中紧急数据的指针。从数据部分的首位到紧急指针所指示的位置为止为紧急数据;
- 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 17.3 所示。

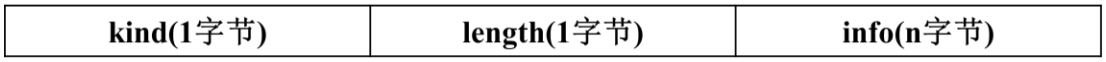


图 17.3 TCP 头部选项

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 17.4 所示。

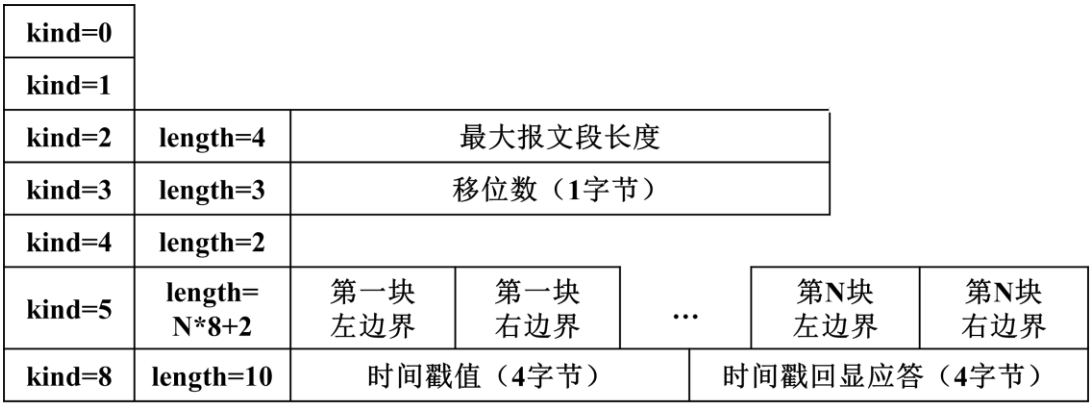


图 17.4 TCP 选项

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

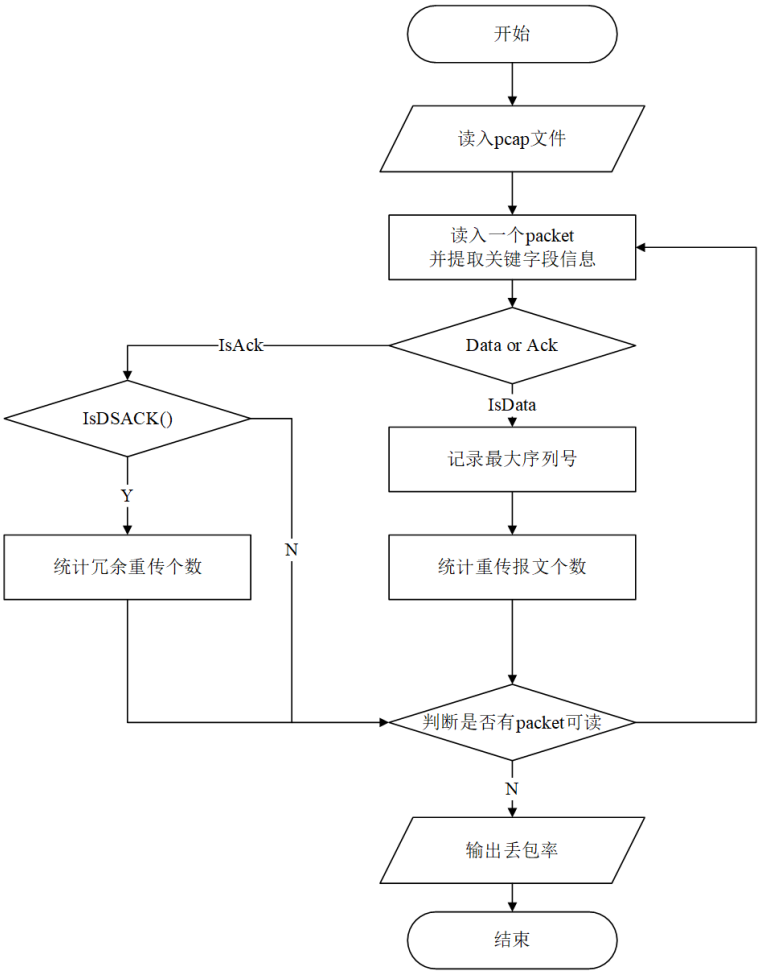
每当发送端发送一个数据报文，我们都去维护一个代表当前最大序列号的变量。发送端在正常情况下会按照 TCP 序列号的从小到大的顺序发送数据报文，当发送端发生重传时当前发送的报文序列号一定是小于当前最大的序列号，我们用 retransmits 变量去记录发送端重传的数量。然而重传并不表示前一个发送的数据包一定丢失，所以单单用 retransmits 去代表丢失的数据包的数量是不那么准确的。原理部分我们讲到了 DSACK，它用来传递一个接收端最近接收到的重复报文的序列号。如果发送端接收到了反馈回来的 DSACK，那么我们就会知道 DSACK 所指示的那个数据包一定发生了重传。我们用 dup_xmits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

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。



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 。

```
~/loss_s$ gcc -w loss_cbei.c -o loss_cbei
~/loss_s$ sudo ./loss_cbei packet_loss.pcap 10.11.44.137 52537 153.35.88.41 443
```

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。

```
13307 :retransmits
13308 :retransmits
13310 :retransmits
13311 :retransmits
13312 :retransmits
13314 :dup_xmits
13317 :retransmits
13318 :retransmits
13319 :retransmits
13320 :retransmits
13321 :retransmits
13322 :retransmits
13323 :dup_xmits
13326 :dup_xmits
13350 :dup_xmits
13351 :dup_xmits
13352 :dup_xmits
13353 :retransmits
13354 :dup_xmits
13355 :dup_xmits
13386 :dup_xmits
read over
packet loss is 0.059
```

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;
}
```