南开大学

Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络实验报告

UDP 可靠传输 Part1

2014074 费泽锟

年级：2020 级

专业：信息安全

指导教师：徐敬东、张建忠

2022 年 11 月 18 日

摘要

在 UDP 实现可靠传输的 Part1 的设计中，主要是为后续的设计内容搭建好了整体的框架，完成了对自己设计的可靠 UDP 的头部 HEADER 结构体的封装，也完成了对 my_udp 类的封装以便后续的扩展设计。

在 Part1 的设计之中，为了实现基于 UDP 的可靠传输，实现了三次握手和四次挥手的动作。在传输文件的过程之中，完成了 rdt3.0 的超时重传机制和停等机制，并且在源代码中添加了对于 Error 差错检验和 Drop 丢包重传的测试。

关键字：UDP 可靠传输、rdt3.0、三次握手、四次挥手

目录

一、 Part1 设计

(一) 可靠 UDP 协议设计

通过课程讲授的知识，我们都知道 UDP 是无连接的传输层协议，提供面向事务的简单不可靠信息传输服务。为了使 UDP 协议变得可靠，我们首先就要考虑关于差错检验的问题，我们需要在自己的协议的设计过程之中，在头部添加上 16bit 的校验和信息。

下图展示的为设计的 HEADER 结构体的字段分布：

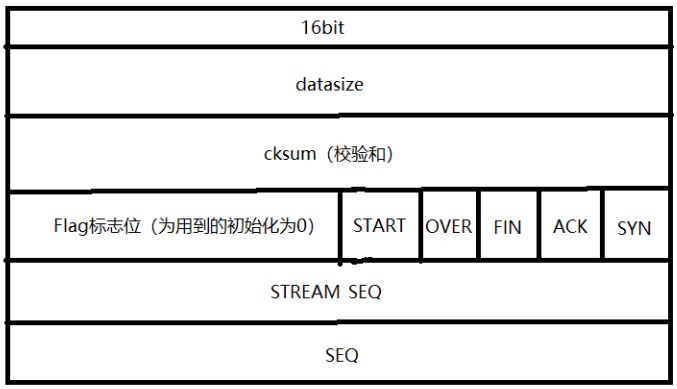

图 1: HEADER 结构

接着我们就需要考虑为了实现 rdt3.0 的可靠传输协议，我们在头部信息中还需要些什么信息，在 Part1 中只需要实现停等机制，理论上只需要 0 和 1 两个序列号就可以了，但是为了后续的 GBN 等实现能够顺利实现，在头部设计中我仍然使用了占据 16bit 的序列号信息，16 位看起来会有点大具体的原因会在后文介绍。

考虑 UDP 是无连接的传输协议，我们的设计之中还需要实现类似于 TCP 三次握手和四次挥手的过程，所以我们在头部信息之中还需要不同的 bit 位来标识这些特殊的响应数据包，我们需要 SYN、ACK 和 FIN 等标志位，所以在设计中我还给头部分配了长为 16bit 的 Flag 字段，为了标识一个文件传输的开始与结束，我们其实应当在应用层协议之中进行设计，但是这里为了简化用户的应用部分，在 Flag 字段之中分配了两个 bit 来标识一个文件的 start 和 over。

最后，在我的设计之中还有其余的两部分，分别为 datasize 和 stream seq 两个字段，这两个字段之中 datasize 是参考 TCP 和 UDP 等协议设计而设置的字段，用于记录一个数据包之中携带的数据字节数，而 stream seq 字段则是考虑如果出现数据包乱序发送的情形，需要这个字段对每个数据包的内容进行标识，所以设计了这个字段，但是这个字段在 Part1 中是没有实际作用的（除非使用并行的方式发送）。

接下来展示的是 HEADER 结构体的源代码以及一些内置的函数：

HEADER 结构体设计

```
1  struct HEADER {
2  uint16_t datasize;
3  uint16_t cksum;
4  // 这里需要注意的是，顺序为STREAM SYN ACK FIN
5  uint16_t Flag; // 使用8位的话，就很麻烦，uint16_t去指向0x7+0x0结果还是0x7
6  uint16_t STREAM_SEQ;
7  uint16_t SEQ;
```

```
8
9       // 初始化函数，STREAM标记的是文件的开始与结束（应当在应用层设计，这里就放
            到udp里标识吧）
10      // 开始的时候需要把文件名传输过去，所以给两位标志位分别为START OVER
11      // 000 START OVER FIN ACK SYN
12      HEADER() {
13          this->datasize = 0;
14          this->cksum = 0;
15          this->Flag = 0;
16          this->STREAM_SEQ = 0;
17          this->SEQ = 0;
18      }
19
20      HEADER(uint16_t datasize, uint16_t cksum, uint16_t Flag, uint16_t
            STREAM_SEQ, uint16_t SEQ) {
21          this->datasize = datasize;
22          this->cksum = cksum;
23          this->Flag = Flag;
24          this->STREAM_SEQ = STREAM_SEQ;
25          this->SEQ = SEQ;
26      }
27
28      void set_value(uint16_t datasize, uint16_t cksum, uint16_t Flag, uint16_t
            STREAM_SEQ, uint16_t SEQ) {
29          this->datasize = datasize;
30          this->cksum = cksum;
31          this->Flag = Flag;
32          this->STREAM_SEQ = STREAM_SEQ;
33          this->SEQ = SEQ;
34      }
35  };
```

在源代码之中除了初始化函数外，仅多了一个 set_value 函数，该函数只是为了赋值简便而设计的。

接下来就会介绍为什么均使用 16bit 的数据类型，因为在实验的过程之中，如果设计 Flag 字段和 STREAM SEQ 字段为 8bit，由于结构体中的数据存储遵循对齐的原则，也就是说会选择结构体中占据 bit 最多的变量类型来进行对齐，所以尽管是 8bit 类型的变量也会按照 16bit 进行存储，那么在使用 check_sum 函数的过程之中想要将两个 8bit 数据按照 16bit 的大小来计算的话，计算出来的校验和就是错误的，所以这里直接手动对齐，均设置为 16bit 位。

在设计好了 HEADER 结构后，我们就可以继续设计 my udp 的类了，对于 UDP 类，这里只是仅仅在 HEADER 结构体之后加入了长度为 4096 的 char 型数组，char 数组不能过大否则会超出 socket 的缓冲区，使用 4096 的原因只是因为页的大小为 4096 字节。

源代码展示如下：

my udp class 类设计

```
1   class my_udp {
2   public:
3       HEADER udp_header;
```

```
    char buffer[DEFAULT_BUFLEN] = ""; // 这里可以+1设置\0, 也可以不+1
public:
    my_udp() {};
    my_udp(HEADER& header);
    my_udp(HEADER& header, string data_segment);
    void set_value(HEADER header, char* data_segment, int size); // 这里一定
        要注意
};

// 针对三次握手和四次挥手的初始化函数
my_udp::my_udp(HEADER& header) {
    udp_header = header;
};

my_udp::my_udp(HEADER& header, string data_segment) {
    udp_header = header;
    for (int i = 0; i < data_segment.length(); i++) {
        buffer[i] = data_segment[i];
    }
    buffer[data_segment.length()] = '\0';
};

void my_udp::set_value(HEADER header, char* data_segment , int size) {
    udp_header = header;
    memcpy(buffer, data_segment, size);
}
```

在协议的结构均设计好之后，首先需要完成就是计算校验和的函数也就是 check_sum 函数，实现的逻辑就是通过一个 16bit 大小的指针强制类型转换去遍历一个指向 class 类对象的地址，来计算校验和。（具体内容参考了课堂讲授的检验校验和伪代码）

计算校验和

```
// 计算校验和, 这里的字符数组运算过程, 充分展现了windows小端存储的特点
uint16_t checksum(uint16_t* udp, int size) {
    int count = (size + 1) / 2;
    uint16_t* buf = (uint16_t*)malloc(size); // 可以+1也可以不+1
    memset(buf, 0, size);
    memcpy(buf, udp, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

正是因为直接使用 16bit 指针遍历 udp 类对象的原因,所以头部信息不能简单的使用 uint8_t 这种数据类型,尽管在测试过程之中也能检验通过校验和,但是实际计算出来的结果是错误的。(可能全部使用 class 类数据存储是连续的)

(二) 文件传输设计

在完成了协议的架构设计后，我们就需要对文件传输的过程来进行设计。我们知道文件传输需要对方知道完整的文件名，所以这里第一个数据包中携带的数据内容就是文件名内容，同时在第一个数据包之中，会使用 HEADER 结构中的 Flag 字段中的 START 标志位，将 START 位置为 1，按道理来说应该在应用层对文件传输的开始进行设计，例如在数据段文件名前带上标志信息，但是这里为了简化传输中的切割 char 数组的操作，就在 udp 的 HEADER 中进行了标识，在某一个文件完全传输完成之后，就将 OVER 位置 1，发送就可以知道这个文件传输已经完成了，可以进行二进制拷贝入文件了。

为了实现 jpg 格式这种字节数很多的文件的传输，我们不可避免地需要将文件分为不同的数据包进行传输的操作。

具体的数据包传输代码如下:

数据传输设计

```
void send_packet(my_udp& Packet, SOCKET& SendSocket, sockaddr_in& RecvAddr) {
    int iResult;
    int RecvAddrSize = sizeof(RecvAddr);
    my_udp Recv_udp;
    char* SendBuf = new char[UDP_LEN];
    char* RecvBuf = new char[UDP_LEN];
    memcpy(SendBuf, &Packet, UDP_LEN);
    iResult = sendto(SendSocket, SendBuf, UDP_LEN, 0, (SOCKADDR*)&RecvAddr,
        sizeof(RecvAddr));
    if (iResult == SOCKET_ERROR) {
        cout << "Sendto failed with error: " << WSAGetLastError() << endl;
    }

    // 测试一下文件的各个内容
    cout << "Send Message " << Packet.udp_header.datasize << " bytes!";
    cout << " Flag:" << Packet.udp_header.Flag << " STREAM_SEQ:" << Packet.
        udp_header.STREAM_SEQ << " SEQ:" << Packet.udp_header.SEQ;
    cout << " Check Sum:" << Packet.udp_header.cksum << endl;

    // 记录发送时间, 超时重传
    // 等待接收ACK信息, 验证序列号
    clock_t start = clock();
    u_long mode = 1;
    ioctlsocket(SendSocket, FIONBIO, &mode);

    while (true) {
        while (recvfrom(SendSocket, RecvBuf, UDP_LEN, 0, (sockaddr*)&RecvAddr
            , &RecvAddrSize) <= 0) {
        // iResult = recvfrom(SendSocket, RecvBuf, UDP_LEN, 0, (sockaddr*)&
            RecvAddr, &RecvAddrSize);
```

```
27          // if (iResult == -1) {
28              // cout << "Recvfrom failed with error: " << WSAGetLastError() <<
                    endl;
29          // }
30
31          if (clock() - start > MAX_TIME) {
32              cout << "*** TIME OUT! ReSend Message *** " << endl;
33
34              // 仅作为调试
35              Packet.udp_header.SEQ = seq_order;
36              memcpy(SendBuf, &Packet, UDP_LEN);
37              iResult = sendto(SendSocket, SendBuf, UDP_LEN, 0, (SOCKADDR*)&
                    RecvAddr, sizeof(RecvAddr));
38
39              cout << "ReSend Message " << Packet.udp_header.datasize << "
                    bytes!";
40              cout << " Flag:" << Packet.udp_header.Flag << " STREAM_SEQ:" <<
                    Packet.udp_header.STREAM_SEQ << " SEQ:" << Packet.udp_header.
                    SEQ;
41              cout << " Check Sum:" << Packet.udp_header.cksum << endl;
42
43              start = clock(); // 重设开始时间
44              if (iResult == SOCKET_ERROR) {
45                  cout << "Sendto failed with error: " << WSAGetLastError() <<
                        endl;
46                  // closesocket(SendSocket);
47                  // WSACleanup();
48              }
49          }
50          }
51
52          // 三个条件要同时满足，这里调试时判断用packet的seq
53          memcpy(&Recv_udp, RecvBuf, UDP_LEN);
54          if (Recv_udp.udp_header.SEQ == Packet.udp_header.SEQ && Recv_udp.
                udp_header.Flag == ACK && checksum((uint16_t*)&Recv_udp, UDP_LEN)
                 == 0) {
55              cout << "Send has been confirmed! Flag:" << Recv_udp.udp_header.
                    Flag << " STREAM_SEQ:" << Recv_udp.udp_header.STREAM_SEQ << "
                     SEQ:" << Recv_udp.udp_header.SEQ << endl;
56              seq_order++; // 全局变量的序列号
57              check_seq();
58              break;
59          }
60          else {
61              continue;
62          }
63      }
64      mode = 0;
```

```
    ioctlsocket(SendSocket, FIONBIO, &mode);

    delete[] SendBuf; // ?
    delete[] RecvBuf;
}
```

这里需要注意的就是，我们需要使用二进制的方式打开一个文件，然后读取文件的大小用于计算平均吞吐量等指标，并且在这里一定要使用 memcpy 函数进行深拷贝，避免出现浅拷贝中的指针的问题，使用该函数的过程之中一定要注意的就是按照 size 字节数的大小进行深拷贝，使用 memcpy 就不需要考虑字符数组末尾的问题了。

具体的 send_file 函数就不再继续展示了，这里继续补充的一点就是，在停等机制之中，我们需要在每次发送完数据包之后进行等待的操作，等待 ACK 响应，每次发送后会重置计时器，如果超时会进行重传，发送成功接收响应后 SEQ 会递增 +1。ACK 的响应需要关注三个点，分别是序列号能否匹配，校验和计算是否为 0 以及是否 ACK 标志位是否为 1。

在 SEQ 序列号和 STREAM SEQ 文件流序列号的使用过程之中，对于每一个文件的传输，SEQ 序列号都会从 0 开始计数，每次计数后都会进行 check 检查避免溢出，而对于 STREAM SEQ 文件流序列号而言，则是每次成功发送一个文件之后递增 +1，每次递增后也会进行检查，用于接收端判断数据包具体是哪个文件流的数据包。

(三) 三次握手与四次挥手

在能够成功实现文件传输的停等机制，差错检验以及超时重传的功能之后，就只差三次握手和四次挥手建立连接的过程了，这个过程之中，发送端只需要在第一次握手，以及第一次挥手的过程之中进行等待响应即可，发送完数据包后，需要 while 循环进行等待消息确认。

在三次握手和四次挥手的过程之中，还涉及到 SEQ 序列号以及 ACK NUM 的问题，在 TCP 之中设置 ACK NUM 字段，但是在我的设计之中做了简化，并没有设计该字段，而作为验证的逻辑则是，在第一次握手发送端发送的过程之中，会将序列号设置为 0xFFFF，接收端会验证这个序列号是否为 0xFFFF，验证通过后会设置第二次握手的数据包的 SEQ 为 0xFFFF+1 也就是 0 进行返回，这时发送端也设置第三次握手的序列号为 0 发送即可，对方验证通过则为建立连接成功，为什么设计成 0xFFFF，因为刚好发送这个数据包之后 SEQ 会被重设为 0，正好为文件传输的开始。

具体的源代码如下：

三次握手源代码

```
// 三次握手建立连接，目前就先不发随机序列号，连接序列号先设为-1，16位SEQ也就是FFFF
bool Connect(SOCKET& SendSocket, sockaddr_in& RecvAddr) {
    HEADER udp_header;
    udp_header.set_value(0, 0, SYN, 0, 0xFFFF); // 初始的SEQ给0xFFFF
    my_udp first_connect(udp_header); // 第一次握手

    uint16_t temp = checksum((uint16_t*)&first_connect, UDP_LEN);
    first_connect.udp_header.cksum = temp;

    // cout << endl;
    // cout << first_connect.udp_header.Flag << " " << first_connect.
        udp_header.SEQ << " " << first_connect.udp_header.cksum << endl;
```

```
12      // cout << first_connect.udp_header.datasize << " " << first_connect.
            udp_header.STREAM_SEQ << endl;
13      // cout << first_connect.buffer << endl;
14      // cout << checksum((uint16_t*)&first_connect, UDP_LEN) << endl;
15
16      int iResult = 0;
17      int RecvAddrSize = sizeof(RecvAddr);
18      char* connect_buffer = new char[UDP_LEN];
19
20      memcpy(connect_buffer, &first_connect, UDP_LEN);// 深拷贝准备发送
21      iResult = sendto(SendSocket, connect_buffer, UDP_LEN, 0, (SOCKADDR*)&
            RecvAddr, RecvAddrSize);
22      if (iResult == SOCKET_ERROR) {
23          cout << "-----*** 第一次握手Error, 请重启Sender ***-----:(" << endl;
24          return 0;
25      }
26
27      clock_t start = clock(); //记录发送第一次握手发出时间
28      u_long mode = 1;
29      ioctlsocket(SendSocket, FIONBIO, &mode); // 设置成阻塞模式等待ACK响应
30
31      // 接收第二次握手ACK响应, 其中的ACK应当为0xFFFF+1 = 0
32      while (recvfrom(SendSocket, connect_buffer, UDP_LEN, 0, (SOCKADDR*)&
            RecvAddr, &RecvAddrSize) <= 0) {
33          // rdt3.0: 超时, 重新传输第一次握手
34          if (clock() - start > MAX_TIME) {
35              cout << "-----*** 第一次握手超时, 正在重传 ***-----:(" << endl;
36              // memcpy(connect_buffer, &first_connect, UDP_LEN); // 这一句好像
                    可以不用
37              iResult = sendto(SendSocket, connect_buffer, UDP_LEN, 0, (
                    SOCKADDR*)&RecvAddr, RecvAddrSize);
38              if (iResult == SOCKET_ERROR) {
39                  cout << "-----*** 第一次握手重传Error, 请重启Sender
                        ***-----:(" << endl;
40                  return 0;
41              }
42              start = clock(); // 重设时间
43          }
44      }
45
46      cout << "-----*** 完成第一次握手 ***-----" << endl;
47
48      // 获取到了第二次握手的ACK消息, 检查校验和, 这时接收到的在connect_buffer
            之中
49      memcpy(&first_connect, connect_buffer, UDP_LEN);
50      // 保存SYN_ACK的SEQ信息, 完成k+1的验证
51      uint16_t Recv_connect_Seq = 0x0;
52      if (first_connect.udp_header.Flag == SYN_ACK && checksum((uint16_t*)&
```

```
        first_connect, UDP_LEN) == 0 && first_connect.udp_header.SEQ == 0
        xFFFF){
53          Recv_connect_Seq = 0xFFFF; // first_connect.udp_header.SEQ
54          cout << "-----*** 完成第二次握手 ***-----" << endl;
55      }
56      else{
57          cout << "-----*** 第二次握手Error, 请重启Sender ***-----:(" << endl;
58          return 0;
59      }
60
61      // 第三次握手ACK, 先清空缓冲区
62      memset(&first_connect, 0, UDP_LEN);
63      memset(connect_buffer, 0, UDP_LEN);
64      first_connect.udp_header.Flag = ACK;
65      first_connect.udp_header.SEQ = Recv_connect_Seq + 1; // 0
66      first_connect.udp_header.cksum = checksum((uint16_t*)&first_connect,
        UDP_LEN);
67      memcpy(connect_buffer, &first_connect, UDP_LEN);
68
69      iResult = sendto(SendSocket, connect_buffer, UDP_LEN, 0, (SOCKADDR*)&
        RecvAddr, RecvAddrSize);
70      if (iResult == SOCKET_ERROR) {
71          cout << "-----*** 第三次握手Error, 请重启Sender ***-----:(" << endl;
72          return 0;
73      }
74
75      cout << "-----*** 接收端成功连接! 可以发送文件! ***-----" << endl;
76
77      delete[] connect_buffer;
78      return 1;
79  }
```

四次挥手的设计与三次握手的设计在停等机制和超时重传上基本一致，这里就不再赘述了，需要说明的是，在四次挥手的过程之中，会出现发送端向接收端发送的过程以及接收端向发送端发送的过程，这里在序列号的使用上双方均使用了 0xFFFF 序列号进行发送，等待 SEQ 为 0 的 ACK 响应消息，如果验证通过，就会继续进四次挥手剩余的步骤，这里就不再展示四次挥手的源代码了。

(四) 程序退出指令设计

这一部分只是为了是整体程序的设计流程完整完善才添加的，因为我们不可能设计文件传输时，只发送选定的一个文件，在发送完成之后就进行四次挥手然后退出，这样显然是不合理的，为了能够持续地传输想要传输的文件，我们需要循环接收命令，直到出现退出的命令。

具体源代码如下:

quit 命令设计

```
1       cout << "******** You can use quit command to disconnect!!! ********" <<
        endl;
```

```
    cout << "************************************************************" <<
        endl << endl;
    while (true) {
        string command;
        cout << "-----*** 请输入想要发送的文件名 ***-----" << endl;
        cin >> command;
        cout << endl;
        if (command == "quit") {
            HEADER command_header;
            command_header.set_value(4, 0, START_OVER, 0, 0);
            my_udp command_udp(command_header);
            char* command_buffer = new char[UDP_LEN];
            memcpy(command_buffer, &command_udp, UDP_LEN);

            if (sendto(SendSocket, command_buffer, UDP_LEN, 0, (SOCKADDR*)&
                RecvAddr, sizeof(RecvAddr)) == SOCKET_ERROR) {
                cout << "-------*** quit命令发送失败 ***-----" << endl;
                return 0;
            }
            else {
                cout << "------*** quit命令发送成功 ***-----" << endl;
            }

            break;
        }
        else {
            clock_t start = clock();
            send_file(command, SendSocket, RecvAddr);
            clock_t end = clock();
            cout << "**传输文件时间为: " << (end - start) / CLOCKS_PER_SEC <<
                 "s" << endl;
            cout << "**吞吐率为:" << ((float)file_size) / ((end - start) /
                CLOCKS_PER_SEC) << " bytes/s " << endl << endl;
            continue;
        }
    }
```

在我的设计之中，也就是 quit 命令，接收到 quit 命令之后，发送端会停止发送，接收端也会停止循环接收，双方紧接着进行四次挥手过程断连。这时，如何在文件传输的过程之中嵌入一个命令，就是一个问题了。所以在设计之中，发送端每次可以输入一个想要发送的文件名或者 quit 命令，发送文件名会进行发送文件的操作，然后继续等待命令。

如果命令为 quit，这时我们就需要单独发送一个数据包给接收端来标识自己要四次挥手了，显然如果在数据段进行标识很容易会被文件的内容混淆，如果文件的内容为 quit 那么就会出现大问题，所以我的设计中取巧使用了不可能出现一种状态，也就是 START_OVER 均设为 1 的状态，使用这个状态来标识即将进行结束，单独发送一个数据包即可，如果接收端发现 Flag 为 START_OVER，那么也会结束 while 循环接收过程。

二、 可靠 UDP 传输流程展示

上述部分，叙述了一大堆设计内容，看起来复杂且枯燥，这里直接进行程序流程的完全展示，来清晰展示具体的实现。

首先是三次握手的实现 Log 信息：

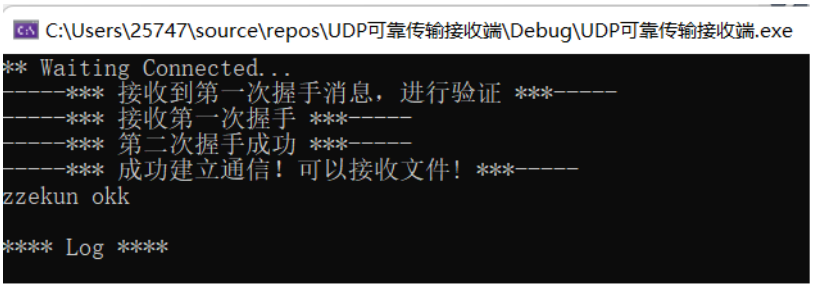

图 2: 接收端三次握手

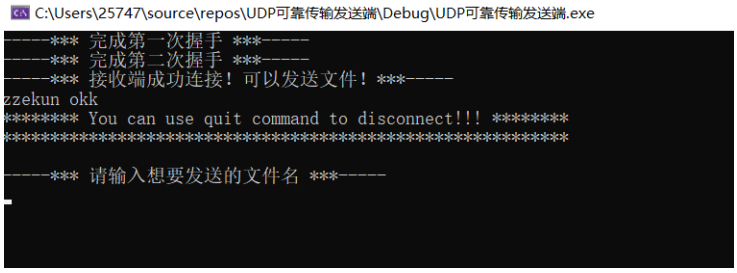

图 3: 发送端三次握手

接下来就是文件传输的过程，文件传输的过程之中会将数据包头部的相应的信息均打印出来，这里的校验和没有加伪首部，因为都将 IP 地址固定为了 127.0.0.1，伪首部的信息是一致的，所以在计算校验和的过程之中没有添加伪首部的内容。

文件传输示例：

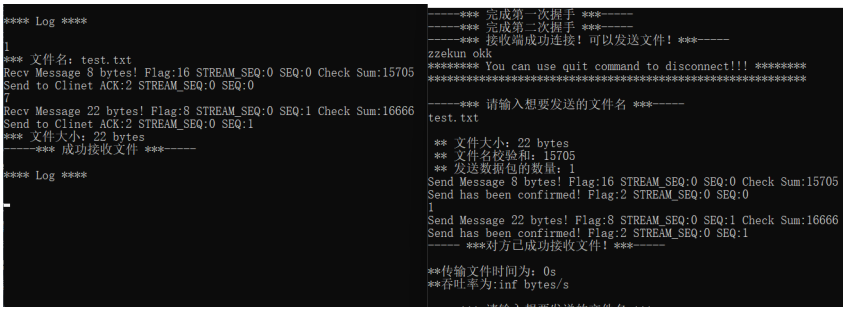

图 4: 文件传输示例

接下来我们就需要去检查如果出现了错误，或者出现了丢包的情况下，该设计能否正确的进行处理这些异常情况了，因为实验所发的路由程序在 Windows 11 操作系统上根据成功同学的指挥，也无法出现丢包情形（猜测是系统兼容问题），所以这里手动设置的错误率和丢包率，在 send packet 前会生成一个 0-9 的随机数，如果随机数小于 1，则人为改变其中的序列号，在接收端，以同样的方式设置 10% 的概率不处理发送端发来的数据包。

具体源代码如下:

接收端丢包率设置

```
// 仅作为测试
int drop_probability = rand() % 10;
cout << drop_probability << endl;
if (drop_probability < 1) {
    continue;
}
```

通过这两处设置，就可以看到我们的程序处理错误和丢包的结果了。

选取的样例数据包如下:

```
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:400 Check Sum:36354
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:400
0
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:402 Check Sum:36353
*** TIME OUT! ReSend Message ***
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:401 Check Sum:36353
Send has been confirmed! Flag:2 STREAM_SEQ:1 SEQ:401
```

图 5: 错误重传样例

可以看到在 401 号数据包发送出现错误之后，SEQ 号别人为设置递增多加了 1 而变成了 402，那么超时后就会进行重传，将正确的 401 号数据包再次发送过去等待 ACK 响应。

接下来展示的就是接收端不处理数据包导致的丢包重传样例:

```
Send Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:400 Check Sum:36354
*** TIME OUT! ReSend Message ***
ReSend Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:400 Check Sum:36354
```

图 6: 丢包重传样例发送端

```
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:399 Check Sum:36355
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:399
0
6
Recv Message 4096 bytes! Flag:0 STREAM_SEQ:1 SEQ:400 Check Sum:36354
Send to Clinet ACK:2 STREAM_SEQ:1 SEQ:400
```

图 7: 丢包重传样例接收端

可以看到在 400 号数据包发送成功后服务端随机数为 0，没有对其进行处理，那么超时后就会进行重传，将正确的 400 号数据包再次发送过去等待 ACK 响应，接下来随机数为 6 进行了处理得到了响应，可以看到接收端的两次操作。

最后就是四次挥手的展示:

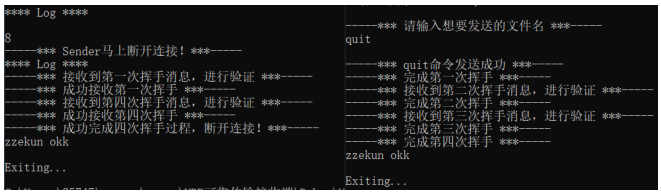

图 8: 四次挥手

我们可以看到在输入了 quit 命令之后双方按照顺序进行了四次挥手断开连接的过程。

三、 总结

在本次实验之中，深入地了解了可靠传输 rdt 从 2.0 到 3.0 的具体的状态机转换内容，熟悉了 C++ 之中关于文件的操作，对于使用 uint_t 类型的变量有了深入的理解，进一步体会了 socket 编程。

这里给出的是个人的 Github 仓库链接：

Github 仓库链接：Github

参考文献

1、https://github.com/AnthonyHaozeZhu
2、03-计算机网络-第三章-2022-s.pdf
3、上机作业 3 讲解-2022.pdf