南開大學
Nankai University

南　　开　　大　　学

网　络　空　间　安　全　学　院

计算机网络实验报告

基于 UDP 服务设计可靠传输协议并编程实现

张洋 2111460

年级：2021 级

专业：信息安全

指导教师：吴英

2023 年 11 月 16 日

目录

1 实验要求

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：**建立连接**、**差错检测**、**接收确认**、**超时重传**等。流量控制采用**停等机制**，完成给定测试文件的传输。

- 数据报套接字：UDP
- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 建立连接、断开连接：类似 TCP 的握手、挥手功能
- 差错检验：校验和
- 接收确认、超时重传：rdt2.0、rdt2.1、rdt2.2、rtd3.0 等，亦可自行设计协议
- 单向传输：发送端、接收端
- 日志输出：收到/发送数据包的序号、ACK、校验和等，传输时间与吞吐率
- 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

2 前期准备

2.1 UDP 协议特点

- 发送方和接收方不需要握手过程
- 每个 UDP 数据单元独立传输
- 提供复用分用功能和可选的差错检测功能
- 支持组播通信
- 不提供可靠性保证：无确认重传、可能有出错、丢失、乱序等现象

2.2 rdt 协议不同版本

- RDT 2.0:

  功能：停等协议。

  实现：发送方发送一个数据包，等待接收方的确认（ACK）消息。如果未收到 ACK，发送方会在超时后重新发送相同的数据包。

- RDT 2.1:

  改进：引入了 NACK（Negative Acknowledgment）。

  实现：除了 ACK 外，接收方还可以发送 NACK，指示发送方某个数据包有误，需要重新发送。

- RDT 2.2:

  改进：引入了校验和和超时重传。

  实现：发送方计算数据包的校验和，接收方验证校验和。如果接收方检测到错误，会发送 NACK。此外，发送方引入了超时重传机制，如果在规定时间内未收到 ACK 或 NACK，发送方会重新发送数据包。

- RDT 3.0:

  改进：引入了选择性重传和可变大小的窗口。

  实现：发送方和接收方都维护一个窗口，窗口内的数据包可以被发送或者接收。接收方可以选择性地接收、确认或拒绝窗口内的数据包，而不仅仅是按顺序处理。此外，引入了 ACK、NACK 和超时重传的机制，以确保数据的可靠传输。

3 实验代码及解释

在本次实验中，我在 rdt3.0 协议的基础上，根据实验具体的实验传输要求设计实现自己的传输协议。

3.1 数据报格式

我自己设计了一个数据报格式，下图为示意图：

0-15	16-31	32-47	48-63
Checksum	seq	ack	flag
length	Source_ip		Des_ip
Des_ip	Source_port	Des_port	data
data	...	...	...

```
struct Header {
    u_short checksum;      //校验和
    u_short seq;           //序列号,停等最低位只有0和1两种状态
    u_short ack;           //ack号, 停等最低位只有0和1两种状态
    u_short flag;          //状态位
    u_short length;        //长度位
    u_int source_ip;       //源ip地址
    u_int des_ip;          //目的ip地址
    u_short source_port;   //源端口号
    u_short des_port;      //目的端口号
    Header() {//构造函数
        checksum = 0;
        source_ip = SOURCEIP;
        des_ip = DESIP;
        source_port = SOURCEPORT;
        des_port = DESPORT;
        seq = 0;
        ack = 0;
        flag = 0;
        length = 0;
    }
};
```

报文头部

flag 段：

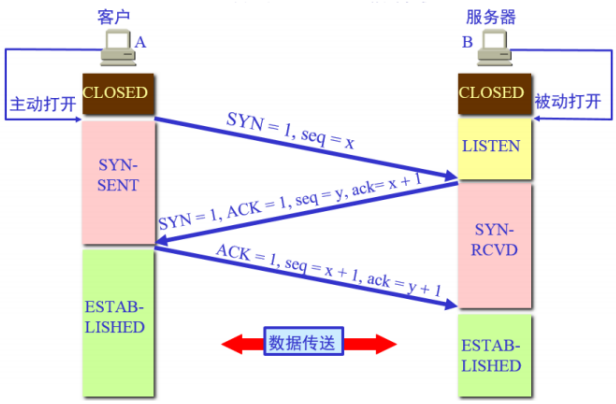
- SYN 作为初始化挥手的请求标志
- ACK 作为确认标志
- SYN_ACK 作为第二次挥手时应该具备的 flag 段值
- OVER 表示传输结束时告知服务端的数据报标识位
- OVER_ACK 表示服务端收到并确认传输结束数据报
- FIN 用作表示尝试断开即挥手时的标识位
- FIN_ACK 表示收到并确认尝试断开请求数据报
- FINAL_CHECK 表示服务端接受到客户端的第四次挥手，避免丢包死锁

```
const unsigned char SYN = 0x01;          //FC=0,OVER=0,FIN=0,ACK=0,SYN=1
const unsigned char ACK = 0x02;          //FC=0,OVER=0,FIN=0,ACK=1,SYN=0
const unsigned char SYN_ACK = 0x03;      //FC=0,OVER=0,FIN=0,ACK=1,SYN=1
const unsigned char FIN = 0x04;          //FC=0,OVER=0,FIN=1,ACK=0,SYN=0
const unsigned char FIN_ACK = 0x06;      //FC=0,OVER=0,FIN=1,ACK=1,SYN=0
const unsigned char OVER = 0x08;         //FC=0,OVER=1,FIN=0,ACK=0,SYN=0
const unsigned char OVER_ACK = 0xA;      //FC=0,OVER=1,FIN=0,ACK=1,SYN=0
const unsigned char FINAL_CHECK=0x10;    //FC=1,OVER=0,FIN=0,ACK=0,SYN=0
```

flag 标志位

3.2 建立连接

建立连接阶段主要是三次握手以及在第三次握手后增加的一个确认机制，第一次由客户端向服务端发送请求，发送 SYN，服务端确认数据报无误后返回 SYN_ACK 表示确认握手请求，最后再由客户端发送 ACK 表示三次握手结束。



3.2.1 接收端

第一次握手:

- 循环等待第一次握手的请求。
- 将套接字设置为非阻塞模式。
- 接收第一次握手请求，如果超时则断开连接。
- 判断接收到的数据是否是期望的连接请求，并且校验和正确。

第二次握手:

- 准备发送第二次握手的信息，设置数据头。
- 将数据头复制到发送缓冲区。
- 发送第二次握手消息，如果失败则断开连接。
- 循环等待接收第二次握手的确认，如果超时则重传第二次握手消息。

第三次握手:

- 接收第二次握手的确认，如果超时则重传第二次握手消息。
- 判断接收到的第二次握手消息是否为期望的确认消息，并且校验和正确。
- 发送第三次握手消息，表示连接建立成功。

```
int Connect() {
    linkClock = clock();
    Header header;//声明一个数据头
    char* recvshbuffer = new char[sizeof(header)];//创建一个和数据头一样大的接收缓冲区
    char* sendshbuffer = new char[sizeof(header)];//创建一个和数据头一样大的发送缓冲区
    cout << "[System]:";
    cout << "等待连接..." << endl;
    //等待第一次握手
    while (true) {
        // 将套接字设置为非阻塞模式
        ioctlsocket(server, FIONBIO, &unblockmode);
        // 接收第一次握手的申请
        while (recvfrom(server, recvshbuffer, sizeof(header), 0,
            (sockaddr*)&router_addr, &rlen)<=0) {
            // 判断连接超时
            if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
                cout  << "[Failed]:";
                cout << "连接超时,服务器自动断开" << endl;
                return -1;
            }
        }
        // 将接收到的数据复制到数据头结构体中
        memcpy(&header, recvshbuffer, sizeof(header));
        //如果是单纯的请求建立连接请求，并且校验和相加取反之后就是0
        if (header.flag == SYN && vericksum((u_short*)(&header), sizeof(header)) ==
            0) {
            cout << "[1]:";
            cout << "第一次握手建立成功。" << endl;
            break;
        }
        else {
            cout << "[Failed]:";
            cout << "第一次握手失败，正在等待重传..." << endl;
        }
    }
SECONDSHAKE:
    //准备发送第二次握手的信息
    header.source_port = SOURCEPORT;
    header.des_port = DESPORT;
```

```
38      header.flag = SYN_ACK;
39      header.source_port = SOURCEPORT;
40      header.des_port = DESPORT;
41      header.ack = (header.seq + 1) % 2;
42      header.seq = 0;
43      header.length = 0;
44      header.checksum = calcksum((u_short*)(&header), sizeof(header));
45      // 将数据头复制到发送缓冲区
46      memcpy(sendshbuffer, &header, sizeof(header));
47      // 发送第二次握手消息
48      if (sendto(server, sendshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
            rlen) == -1) {
49          cout << "[Failed]:第二次握手消息发送失败...." << endl;
50          return -1;
51      }
52      cout << "[2]:第二次握手消息发送成功...." << endl;
53      // 判断连接超时
54      clock_t start = clock();
55      if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
56          cout << "[Failed]:连接超时,服务器自动断开" << endl;
57          return -1;
58      }
59
60      // 第二次握手消息的超时重传, 重传时直接重传sendshbuffer里的内容
61      while (recvfrom(server, recvshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
            &rlen) <= 0) {
62          if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
63              cout << "[Failed]:连接超时,服务器自动断开" << endl;
64              return -1;
65          }
66          if (clock() - start > MAX_TIME) {
67              // 重传第二次握手消息
68              if (sendto(server, sendshbuffer, sizeof(header), 0,
                    (sockaddr*)&router_addr, rlen) == -1) {
69                  cout << "[Failed]:第二次握手消息重新发送失败..." << endl;
70                  return -1;
71              }
72              cout << "[2]:第二次握手消息重新发送成功..." << endl;
73              start = clock();
74          }
75      }
76
77      memcpy(&header, recvshbuffer, sizeof(header));
78      if (header.flag == ACK && vericksum((u_short*)(&header), sizeof(header)) == 0) {
79          cout << "[3]:第三次握手建立成功! 可以开始接收数据..." << endl;
80      return 1;
81  }
```

服务器端建立连接

3.2.2 发送端

第一次握手:
- 设置第一次握手请求的数据头。
- 将数据头复制到发送缓冲区。
- 发送第一次握手消息，如果失败则断开连接。
- 将套接字设置为非阻塞模式。
- 设置计时器，并进行第一次握手消息的超时重传。

第二次握手:
- 循环等待接收第二次握手的确认，如果超时则重传第一次握手消息。
- 检验接收到的第二次握手消息是否为期望的确认消息，并且校验和正确。

第三次握手:
- 设置第三次握手的数据头。
- 将数据头复制到发送缓冲区。
- 发送第三次握手消息，如果失败则断开连接。
- 循环等待接收第三次握手的确认，如果超时则重传第三次握手消息。

```
int Connect() {
    linkClock = clock();
    Header header;
    char* recvshbuffer = new char[sizeof(header)];
    char* sendshbuffer = new char[sizeof(header)];

    //第一次握手请求
    header.source_port = SOURCEPORT;
    header.des_port = DESPORT;
    header.flag = SYN;
    header.seq = 0;
    header.ack = 0;
    header.length = 0;
    header.checksum = calcksum((u_short*)&header, sizeof(header));
    //cout << vericksum((u_short*)&header, sizeof(header)) << endl;
    u_short* test = (u_short*)&header;
    memcpy(sendshbuffer, &header, sizeof(header));
FIRSTSHAKE:
    if (sendto(client, sendshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
        rlen) == -1) {
        cout << "[Failed]:第一次握手请求发送失败..." << endl;
        return -1;
    }
    cout << "[1]:第一次握手消息发送成功...." << endl;
    //设置是否为非阻塞模式
    ioctlsocket(client, FIONBIO, &unlockmode);

    //设置计时器
    clock_t start = clock();

```

```
30      //第一次握手重传
31      while (recvfrom(client, recvshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
            &rlen) <= 0) {
32          if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
33              cout << "[Failed]:连接超时,服务器自动断开" << endl;
34              return -1;
35          }
36          if (clock() - start > MAX_TIME) {
37              if (sendto(client, sendshbuffer, sizeof(header), 0,
                    (sockaddr*)&router_addr, rlen) == -1) {
38                  cout << "[Failed]:第一次握手请求发送失败..." << endl;
39                  return -1;
40              }
41              start = clock();
42              cout << "[1]:第一次握手消息反馈超时....正在重新发送" << endl;
43              goto FIRSTSHAKE;
44          }
45      }
46
47      //检验第二次握手信息是否准确
48      memcpy(&header, recvshbuffer, sizeof(header));
49      if (header.flag == SYN_ACK && vericksum((u_short*)&header, sizeof(header)) == 0) {
50          cout << "[2]:第二次握手建立成功..." << endl;
51      }
52      else {
53          cout << "[1]:不是期待的服务端数据包,即将重传第一次握手数据包...." << endl;
54          if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
55              cout << "[Failed]:连接超时,服务器自动断开" << endl;
56              return -1;
57          }
58          goto FIRSTSHAKE;
59      }
60
61      //发送第三次握手信息
62      header.source_port = SOURCEPORT;
63      header.des_port = DESPORT;
64      header.flag = ACK;
65      header.seq = 1;
66      header.ack = 1;
67      header.checksum = calcksum((u_short*)&header, sizeof(header));
68      memcpy(sendshbuffer, &header, sizeof(header));
69  THIRDSHAKE:
70      if (sendto(client, sendshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
            rlen) == -1) {
71          cout << "[Failed]:第三次握手信息发送失败...." << endl;
72          return -1;
73      }
74      start = clock();
75      while (recvfrom(client, recvshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
```

```
        &rlen) <= 0) {
        if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
            cout << "[Failed]:连接超时,服务器自动断开" << endl;
            return -1;
        }
        if (clock() - start >= 5 * MAX_TIME) {
            cout << "[Failed]:第三次握手信息反馈超时...正在重新发送" << endl;
            if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
                cout << "[Failed]:连接超时,服务器自动断开" << endl;
                return -1;
            }
            goto THIRDSHAKE;
        }
    }
    memcpy(&header, recvshbuffer, sizeof(header));
    if (header.flag == ACK && vericksum((u_short*)&header, sizeof(header)) == 0) {
        cout << "[3]:正确发送第三次握手消息" << endl;
        cout << "[DONE]:与服务器连接成功, 准备发送数据" << endl;
        return 1;
    }
    else {
        cout << "[ERROR]:确认消息接受失败...重发第三次握手" << endl;
        goto THIRDSHAKE;
    }
}
```

客户端建立连接

3.2.3 改进

在第三次握手数据报发送后可能会丢包，这将导致服务端无法正常开始，于是规定当客户端发送第三次握手后需要服务器返回一个再确认数据报，如果客户端在 MAX_TIME 的时限内未收到该数据报则会重传第三次握手消息，在超过 LINKTIME 的情况下将直接断开连接。

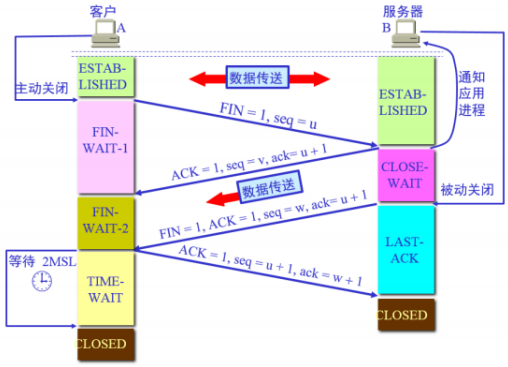
```
// 设置第三次握手消息的信息
        header.source_port = SOURCEPORT;
        header.des_port = DESPORT;
        header.flag = ACK;
        header.ack = (header.seq + 1) % 2;
        header.seq = 0;
        header.length = 0;
        header.checksum = calcksum((u_short*)(&header), sizeof(header));
        memcpy(sendshbuffer, &header, sizeof(header));
        sendto(server, sendshbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
            rlen);
        cout << "[DONE]:确认信息传输成功...." << endl;
    }
    else {
        cout << "[Failed]:不是期待的数据包, 正在重传并等待客户端等待重传" << endl;
```

```
        if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
            cout << "[Failed]:连接超时,服务器自动断开" << endl;
            return -1;
        }
        goto SECONDSHAKE;
    }
    cout << "[WAITING]:正在等待接收数据...." << endl;
    return 1;
```

第三次握手后的改进

3.3 断开连接

断开连接阶段主要是四次挥手以及对四次挥手的改进。先由客户端发起请求，发送 FIN，随后客户端先后发送 ACK 以及 FIN_ACK 来作为确认请求发送给客户端，最后客户端在发送 ACK 后等待两个 MSL 后中断连接。



3.3.1 接收端

接收第一次挥手消息:
- 通过 recvfrom 函数接收来自服务器的第一次挥手消息。
- 将接收到的数据存储到 recvbuffer 中，并解析为 Header 结构。
- 检查消息的标志位是否为 FIN，以及校验和是否通过。

发送第二次挥手消息:
- 构造一个 ACK 标志的 Header 结构。
- 设置序列号和校验和。
- 将数据存储到 sendbuffer 中，并通过 sendto 函数发送给服务器。
- 在发送后等待一段时间（Sleep(80)），允许服务器处理。

发送第三次挥手消息:
- 构造一个 FIN_ACK 标志的 Header 结构。
- 设置序列号和校验和。
- 将数据存储到 sendbuffer 中，并通过 sendto 函数发送给服务器。

等待第四次挥手消息:
- 设置套接字为非阻塞模式。
- 使用循环和 recvfrom 函数等待服务器的第四次挥手消息。

- 如果超时，准备重新发送第二和第三次挥手消息。

```
int Disconnect() {
    Header header;
    char* sendbuffer = new char[sizeof(header)];
    char* recvbuffer = new char[sizeof(header)];
RECVWAVE1:
    while(recvfrom(server,recvbuffer,sizeof(header),0,(sockaddr*)&router_addr,&rlen)<=0){}
    memcpy(&header, recvbuffer, sizeof(header));
    if (header.flag == FIN && checksum((u_short*)&header, sizeof(header)) == 0) {
        cout << "[1]:第一次挥手消息接收成功" << endl;
    }
    else {
        cout << "[Failed]:第一次挥手消息接收失败" << endl;
        goto RECVWAVE1;
    }
SENDWAVE2:
    header.seq = 0;
    header.flag = ACK;
    header.checksum = calcksum((u_short*)&header, sizeof(header));
    memcpy(sendbuffer, &header, sizeof(header));
    if (sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen)
        == -1) {
        cout << "[Failed]:第二次挥手消息发送失败..." << endl;
        return -1;
    }
    cout << "[2]:第二次挥手消息发送成功..." << endl;
    Sleep(80);
    //第三次挥手
    header.seq = 1;
    header.flag = FIN_ACK;
    header.checksum = calcksum((u_short*)&header, sizeof(header));
    memcpy(sendbuffer, &header, sizeof(header));
    if (sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen)
        == -1) {
        cout << "[Failed]:第三次挥手消息发送失败..." << endl;
        return -1;
    }
    cout << "[3]:第三次挥手消息发送成功..." << endl;
    clock_t start = clock();
    ioctlsocket(server, FIONBIO, &unblockmode);
    while (recvfrom(server, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
        &rlen) <= 0) {
        if (clock() - start > MAX_TIME) {
            cout << "[Failed]:第四次挥手消息接收超时...准备重发二三次挥手" << endl;
            ioctlsocket(server, FIONBIO, &blockmode);
            goto SENDWAVE2;
        }
    }
```

```
45      cout << "[4]:第四次挥手消息接收成功" << endl;
```

服务器端断开连接

3.3.2 发送端

发送第一次挥手消息:
- 构造一个 FIN 标志的 Header 结构。
- 设置序列号和校验和。
- 将数据存储到 sendbuffer 中，并通过 sendto 函数发送给服务器。
- 等待第二次挥手消息的反馈，如果超时则重新发送第一次挥手消息。

等待第二次挥手消息:
- 使用循环和 recvfrom 函数等待服务器的第二次挥手消息。
- 如果超时，重新发送第一次挥手消息。
- 接收到消息后，检查标志位是否为 ACK，以及校验和是否通过。

等待第三次挥手消息:
- 使用循环和 recvfrom 函数等待服务器的第三次挥手消息。
- 接收到消息后，检查标志位是否为 FIN_ACK，以及校验和是否通过。

发送第四次挥手消息:
- 构造一个 ACK 标志的 Header 结构。
- 设置序列号和校验和。
- 将数据存储到 sendbuffer 中，并通过 sendto 函数发送给服务器。
- 等待第四次挥手消息的反馈，如果超时则重新发送第四次挥手消息。

```
1   int Disconnect() {
2       Header header;
3       char* sendbuffer = new char[sizeof(header)];
4       char* recvbuffer = new char[sizeof(header)];
5       header.seq = 0;
6       header.flag = FIN;
7       header.checksum = calcksum((u_short*)&header, sizeof(header));
8       memcpy(sendbuffer, &header, sizeof(header));
9       if (sendto(client, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen)
            == -1) {
10          cout << "[Failed]:第一次挥手发送失败" << endl;
11          return -1;
12      }
13  WAITWAVE2:
14      clock_t start = clock();
15      while (recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
            &rlen) <= 0) {
16          if (clock() - start > MAX_TIME) {
17              if (sendto(client, sendbuffer, sizeof(header), 0,
                    (sockaddr*)&router_addr, rlen) == -1) {
18                  cout << "[Failed]:第一次挥手发送失败" << endl;
19                  return -1;
```

```
            }
            start = clock();
            cout << "[Failed]:第一次挥手消息反馈超时" << endl;
        }
    }
    cout << "[1]:第一次挥手消息发送成功" << endl;
    memcpy(&header, recvbuffer, sizeof(header));
    if (header.flag == ACK && checksum((u_short*)&header, sizeof(header)) == 0) {
        cout << "[2]:第二次挥手消息接收成功" << endl;
    }
    else {
        cout << "[Failed]:第二次挥手消息接受失败" << endl;
        goto WAITWAVE2;
    }
WAITWAVE3:
    while(recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
        &rlen) <= 0){}
    memcpy(&header, recvbuffer, sizeof(header));
    if (header.flag == FIN_ACK && checksum((u_short*)&header, sizeof(header)) == 0) {
        cout << "[3]:第三次挥手消息接收成功" << endl;
    }
    else {
        cout << "[Failed]:第三次挥手消息接受失败" << endl;
        goto WAITWAVE3;
    }
    header.seq = 1;
    header.flag = ACK;
    header.checksum = calcksum((u_short*)&header, sizeof(header));
    memcpy(sendbuffer, &header, sizeof(header));
SENDWAVE4:
    if (sendto(client, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen)
        == -1) {
        cout << "[Failed]:第四次挥手发送失败" << endl;
        return -1;
    }
    start = clock();
    ioctlsocket(client, FIONBIO, &unlockmode);
    while (recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
        &rlen) <= 0) {
        if (clock() - start > 2 * MAX_TIME) {
            cout << "[Failed]:接受反馈超时, 重发第四次挥手" << endl;
            goto SENDWAVE4;
        }
    }
    cout << "[4]:第四次挥手发送成功" << endl;
```

客户端断开连接

3.3.3 改进

如果第四次挥手的包丢失，服务端将无法正常关闭，改进后要求服务端在收到第四次挥手消息后发送一个确认数据报，并等待 10 个 MAX_TIME 后再关闭连接，因为如果服务端的包丢失了，那么最多两个 MAX_TIME，客户端就会重发第四次挥手的消息，此时服务端就可以发现自己的再确认消息丢包，于是就会再次发送，这样双方都可以完成连接的正常关闭。

服务器端发送最终确认的数据报

- 构造一个 FINAL_CHECK 标志的 Header 结构。
- 设置序列号和校验和。
- 将数据存储到 sendbuffer 中，并通过 sendto 函数发送给服务器。
- 在等待确认的阶段如果超时，通过 goto FINALCHECK 回到代码的开头，重新发送 FINAL_CHECK 消息并等待确认。

```
FINALCHECK:
    memcpy(&header, recvbuffer, sizeof(header));
    if (header.flag == ACK && checksum((u_short*)&header, sizeof(header)) == 0) {
        header.seq = 0;
        header.flag = FINAL_CHECK;
        header.checksum = calcksum((u_short*)&header, sizeof(header));
        memcpy(sendbuffer, &header, sizeof(header));
        sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen);
        cout << "[FINAL]:成功发送确认报文" << endl;
    }
    start = clock();
    while (recvfrom(server, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
        &rlen) <= 0) {
        if (clock() - start > 10 * MAX_TIME) {
            cout << "[System]:四次挥手结束，即将断开连接" << endl;
            return 1;
        }
    }
    goto FINALCHECK;
```

服务器端第四次挥手后的改进

客户端接收最终确认的数据报

- 接收第四次挥手消息的反馈。
- 检查标志位是否为 FINAL_CHECK，以及校验和是否通过。
- 如果确认成功，表示四次挥手完成，即将断开连接。

```
    memcpy(&header, recvbuffer, sizeof(header));
    if (header.flag == FINAL_CHECK && checksum((u_short*)&header, sizeof(header)) ==
        0) {
        cout << "[System]:四次挥手完成，即将断开连接" << endl;
        return 1;
    }
    else {
        cout << "[Failed]:数据包错误,准备重发第四次握手" << endl;
```

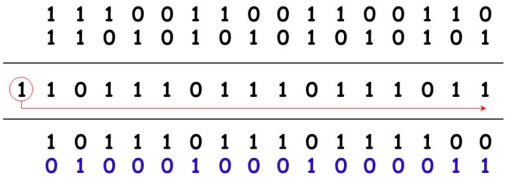
```
        goto SENDWAVE4;
    }
```

客户端第四次挥手后的改进

3.4 差错检验

校验和的计算是差错检验的核心途径，具体的校验和计算方法就是传入一个 char* 指针指向需要计算校验和的起始地址以及他的字节长度，根据字节长度计算需要计算多少次校验和。

校验和的具体计算流程为对每一个 16 位数进行二进制相加，如果产生进位就把最高位加到最后一位，等到所有的计算结束后再进行取反。下图是校验和计算的示例。




```
u_short checksum(u_short* mes, int size) {
    int count = (size + 1) / 2;
    u_short* buf = (u_short*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, mes, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

校验和

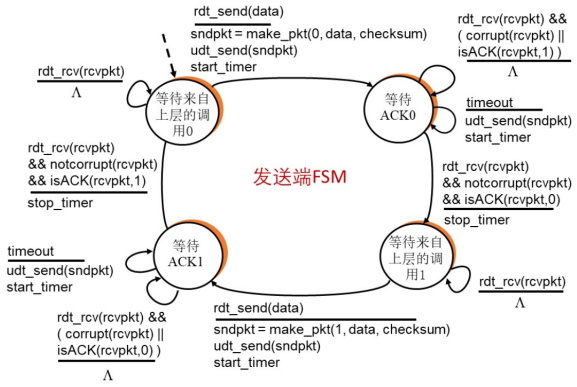
3.5 确认重传

本次实验采用停等机制，所以只需要有两个序列号 0、1 就可以满足实验要求，根据停等机制的特性，不支持流水线作业，所以当未收到准确无误的对 0 号序列号确认的 ACK 前不能发送 1 号数据报。

3.5.1 发送端

在这里客户端是发送端，根据停等机制的特性，我们可以使用 while 循环来完成对停等机制的实现，因为实现的是 rdt3.0，所以在传输过程中还要对可能发生的错误进行检测并完成差错重传。网络中可能存在的传输错误大体有两种：丢包和数据报损毁。我们可以使用超时重传机制来判定网络中的

丢包现象并重新发包，使用计算校验和的方式来判断包的数据完整性。并且使用有限状态机来描述传输过程中具体可能出现的情况。



当发包完成后即进入等待确认状态，并且设置时钟，如果时钟超时还未收到服务端的确认则触发超时重传机制，将重新发送对应的数据报，如果收到了服务端的确认信息，还要检测该确认信息的 ack 是否是发送的（seq+1）%2 并检查数据报是否损毁，如果没有损毁则认为成功接受到确认，并准备发送下一个数据报。

消息分段发送：

- 通过循环发送消息，每次发送一个数据包。
- 每个数据包包含一个 Header 和实际数据部分。
- 数据包的序列号通过 header.seq 区分，可以为 0 或 1。
- 每个数据包的发送前会检查是否还有未发送的消息，如果所有消息已发送完毕，通过 endsend() 完成最终的消息发送。

数据包发送流程：

- 计算本次数据包的长度 ml。
- 设置 header.seq 和 header.length。
- 将 header 复制到 sendbuffer 中。
- 复制实际数据到 sendbuffer 中。
- 计算并填充校验和 header.checksum。
- 发送数据包，并输出相关信息，包括序列号、数据长度、校验和等。

数据包接收和确认：

- 通过 recvfrom 接收服务器的确认消息，使用非阻塞模式。
- 检查接收到的 header.ack 和校验和是否正确。
- 如果正确，表示数据包接收成功，输出相关信息。
- 如果不正确，进行错误处理，并重新发送对应序列号的数据包。

超时重传机制：

- 使用 clock 函数记录开始时间，并在一定时间内等待接收确认消息。
- 如果超时，重新发送数据包，并输出相关信息。
- 这里使用了 goto 语句实现了一个简单的重传机制，即回到发送数据包的步骤。

```
1 int sendmessage() {
2     ALLSTART = clock();
3     //设置是否为非阻塞模式
```

```
    ioctlsocket(client, FIONBIO, &unlockmode);

    Header header;
    char* recvbuffer = new char[sizeof(header)];
    char* sendbuffer = new char[sizeof(header) + MAX_DATA_LENGTH];

    while (true) {

        int ml;//本次数据传输长度
        if (messagepointer > messagelength) {//不需要再发了，都发完了
            if (endsend() == 1) { return 1; }
            return -1;
        }
        if (messagelength - messagepointer >= MAX_DATA_LENGTH) {//可以按照最大限度发送
            ml = MAX_DATA_LENGTH;
        }
        else {
            ml = messagelength - messagepointer + 1;//需要计算发送的长度
        }

        header.seq = 0;//这次发的是零号序列号
        header.length = ml;//实际数据长度
        memset(sendbuffer, 0, sizeof(header) + MAX_DATA_LENGTH);//sendbuffer全部置零
        memcpy(sendbuffer, &header, sizeof(header));//拷贝header内容
        memcpy(sendbuffer+sizeof(header), message + messagepointer, ml);//拷贝数据内容
        messagepointer += ml;//更新数据指针
        header.checksum = calcksum((u_short*)sendbuffer,
            sizeof(header)+MAX_DATA_LENGTH);//计算校验和
        memcpy(sendbuffer, &header, sizeof(header));//填充校验和
    SEQ0SEND:
        //发送seq=0的数据包
        cout << "[send]:" << endl << "seq:0" << " length:" << ml<<" ";
        cout << "checksum:" << checksum((u_short*)sendbuffer, sizeof(header) +
            MAX_DATA_LENGTH) << endl;
        if (sendto(client, sendbuffer, (sizeof(header) + MAX_DATA_LENGTH), 0,
            (sockaddr*)&router_addr, rlen) == -1) {
            cout << "[Failed]:seq0数据包发送失败" << endl;
            return -1;
        }
        clock_t start = clock();
    SEQ0RECV:
        //如果收到数据了就不发了，否则延时重传
        //设置了非阻塞，所以不会卡住
        while (recvfrom(client, recvbuffer, sizeof(header), 0,
            (sockaddr*)&router_addr, &rlen) <= 0) {
            if (clock() - start > MAX_TIME) {
                if (sendto(client, sendbuffer, (sizeof(header)+MAX_DATA_LENGTH), 0,
                    (sockaddr*)&router_addr, rlen) == -1) {
                    cout << "[Failed]:seq0数据包发送失败" << endl;
```

```
48                      return -1;
49                  }
50                  start = clock();
51                  cout << "[ERROR]:seq0数据包反馈超时" << endl;
52              }
53          }
54          //检查ack位是否正确，如果正确则准备发下一个数据包
55          memcpy(&header, recvbuffer, sizeof(header));
56          cout << "[RECV]:" << endl << "ack:" << header.ack << "checksum:" <<
                checksum((u_short*)&header, sizeof(header)) << endl;
57          if (header.ack == 1 && checksum((u_short*)&header, sizeof(header) == 0)) {
58              cout << "[CHECKED]:seq0接收成功" << endl;
59          }
60          else {
61              cout << "[ERROR]:服务端未反馈正确的数据包" << endl;
62              goto SEQ0SEND;
63          }
64
65          //准备开始发SEQ=1的数据包
66          if (messagepointer > messagelength) {
67              if (endsend() == 1) { return 1; }
68              return -1;
69          }
70          if (messagelength - messagepointer >= MAX_DATA_LENGTH) {
71              ml = MAX_DATA_LENGTH;
72          }
73          else {
74              ml = messagelength - messagepointer + 1;
75          }
76
77          header.seq = 1;//零号序列号
78          header.length = ml;//实际数据长度
79          memset(sendbuffer, 0, sizeof(header) + MAX_DATA_LENGTH);//sendbuffer全部置零
80          memcpy(sendbuffer, &header, sizeof(header));//拷贝header内容
81          memcpy(sendbuffer + sizeof(header), message + messagepointer,
                ml);//拷贝数据内容
82          messagepointer += ml;//更新数据指针
83          header.checksum = calcksum((u_short*)sendbuffer, sizeof(header) +
                MAX_DATA_LENGTH);//计算校验和
84          memcpy(sendbuffer, &header, sizeof(header));//填充校验和
85          //cout << checksum((u_short*)sendbuffer, sizeof(header) + MAX_DATA_LENGTH) <<
                endl;
86      SEQ1SEND:
87          //发送seq=1的数据包
88          cout << "[send]:" << endl << "seq:1" << " length:" << ml <<" ";
89          cout << "checksum:" << checksum((u_short*)sendbuffer, sizeof(header) +
                MAX_DATA_LENGTH) << endl;
90          if (sendto(client, sendbuffer, (sizeof(header) + MAX_DATA_LENGTH), 0,
                (sockaddr*)&router_addr, rlen) == -1) {
```

```
91              cout << "[Failed]:seq1数据包发送失败" << endl;
92              return -1;
93          }
94          start = clock();
95      SEQ1RECV:
96          //如果收到数据了就不发了，否则延时重传
97          while (recvfrom(client, recvbuffer, sizeof(header), 0,
                (sockaddr*)&router_addr, &rlen) <= 0) {
98              if (clock() - start > MAX_TIME) {
99                  if (sendto(client, sendbuffer, (sizeof(header)+MAX_DATA_LENGTH), 0,
                        (sockaddr*)&router_addr, rlen) == -1) {
100                     cout << "[Failed]:seq1数据包发送失败" << endl;
101                     return -1;
102                 }
103                 start = clock();
104                 cout << "[ERROR]:seq1数据包反馈超时" << endl;
105             }
106         }
107         //检查ack位是否正确，如果正确则准备发下一个数据包
108         memcpy(&header, recvbuffer, sizeof(header));
109         cout << "[RECV]:" << endl << "ack:" << header.ack << "checksum:" <<
                checksum((u_short*)&header, sizeof(header)) << endl;
110         if (header.ack == 0 && checksum((u_short*)&header, sizeof(header)) == 0) {
111             cout << "[CHECKED]seq1接收成功" << endl;
112         }
113         else {
114             cout << "[ERROR]:服务端未反馈正确的数据包" << endl;
115             goto SEQ1SEND;
116         }
117     }
118 }
```

发送端确认重传

判断所有消息是否已发送完毕，通过 endsend() 完成最终的消息发送。

传输结束信号发送：

- 构造一个 Header 结构，设置 flag 为 OVER 表示传输结束。
- 计算并填充校验和 header.checksum。
- 将 header 复制到 sendbuffer 中。
- 通过 sendto 发送传输结束信号，输出相关信息。
- 使用 clock 记录发送开始的时间。

传输结束信号接收和确认：

- 通过 recvfrom 接收服务器的确认消息，使用非阻塞模式。
- 如果超时，重新发送传输结束信号，并输出相关信息。
- 检查接收到的 header.flag 和校验和是否正确。
- 如果正确，表示传输结束消息发送成功，输出相关信息。
- 如果不正确，进行错误处理，并重新发送传输结束信号。

超时重传机制:

- 使用 clock 函数记录开始时间，并在一定时间内等待接收确认消息。
- 如果超时，重新发送传输结束信号，并输出相关信息。
- 这里使用了 goto 语句实现了一个简单的重传机制，即回到发送传输结束信号的步骤。

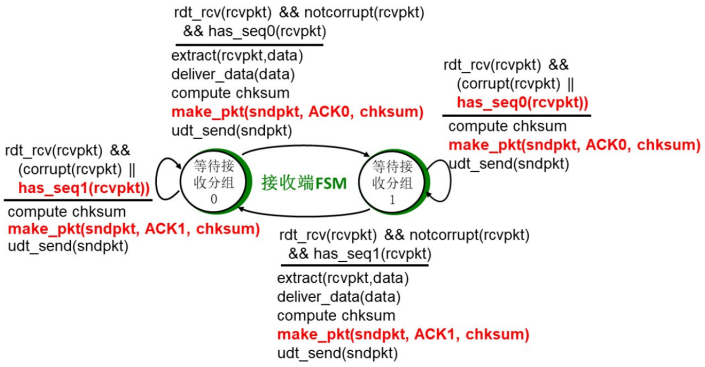
```
int endsend() {
    ALLEND = clock();
    Header header;
    char* sendbuffer = new char[sizeof(header)];
    char* recvbuffer = new char[sizeof(header)];

    header.flag = OVER;
    header.checksum = calcksum((u_short*)&header, sizeof(header));
    memcpy(sendbuffer, &header, sizeof(header));
SEND:
    if (sendto(client, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen)
        == -1) {
        cout << "[Failed]:传输结束信号发送失败" << endl;
        return -1;
    }
    cout << "[SEND]:传输结束信号发送成功" << endl;
    clock_t start = clock();
RECV:
    while (recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr,
        &rlen) <= 0) {
        if (clock() - start > MAX_TIME) {
            if (sendto(client, sendbuffer, sizeof(header), 0,
                (sockaddr*)&router_addr, rlen) == -1) {
                cout << "[Failed]:传输结束信号发送失败" << endl;
                return -1;
            }
            start = clock();
            cout << "[ERROR]:传输结束信号反馈超时" << endl;
            goto SEND;
        }
    }
    memcpy(&header, recvbuffer, sizeof(header));
    if (header.flag == OVER_ACK && checksum((u_short*)&header, sizeof(header)) == 0) {
        cout << "[FINFISH]:传输结束消息发送成功" << endl;
        return 1;
    }
    else {
        cout << "[ERROR]:数据包错误" << endl;
        goto RECV;
    }
}
```

endsend

3.5.2 接收端

在这里服务器端是接收端，接收发送端发送的数据报并检测数据报的正确性，如果数据报正确则保存进缓冲区，反之则直接丢弃。如果收到正确的数据报应给回 ACK 信号，ack 的值应为（seq+1）%2，同样在接收端也设置超时重传和差错检验，如果超过固定的时间没有收到发送端的消息则重传 ACK 数据报，同理如果数据报出错则不给回 ACK，等待发送端重发。



接收 seq=0 的数据包：

- 使用 recvfrom 接收来自客户端的数据包，该数据包包含 Header 和实际数据。
- 检查是否接收到传输结束的信号（OVER 标志），如果是，进行结束消息的处理。
- 检查数据包的校验和和序列号，如果正确，将数据包中的数据复制到消息缓冲区中，并输出相关信息。
- 如果数据包错误，输出错误信息，并等待对方重新发送。

发送 ACK1 消息：

- 构造一个 Header 结构，设置 ack 为 1，表示接收到 seq=0 的数据包。
- 计算并填充校验和 header.checksum。
- 将 header 复制到 sendbuffer 中。
- 通过 sendto 发送 ACK1 消息，输出相关信息。
- 设置一个计时器，并在一定时间内等待 ACK1 的反馈，如果超时，重新发送 ACK1 消息。

接收 seq=1 的数据包：

- 使用 recvfrom 接收来自客户端的 seq=1 的数据包。
- 检查是否接收到传输结束的信号（OVER 标志），如果是，进行结束消息的处理。
- 检查数据包的校验和和序列号，如果正确，将数据包中的数据复制到消息缓冲区中，并输出相关信息。
- 如果数据包错误，输出错误信息，并等待对方重新发送。

发送 ACK0 消息：

- 构造一个 Header 结构，设置 ack 为 0，表示接收到 seq=1 的数据包。
- 计算并填充校验和 header.checksum。
- 将 header 复制到 sendbuffer 中。
- 通过 sendto 发送 ACK0 消息，输出相关信息。

重复以上步骤：

- 使用 goto 语句回到接收 seq=0 的数据包的步骤，以循环接收和处理数据包。

```
1 int receivemessage() {
```

```
    Header header;
    char* recvbuffer = new char[sizeof(header) + MAX_DATA_LENGTH];
    char* sendbuffer = new char[sizeof(header)];

RECVSEQ0:
    //接受seq=0的数据
    while (true) {
        //此时可以设置为阻塞模式
        ioctlsocket(server, FIONBIO, &unblockmode);
        while (recvfrom(server, recvbuffer, sizeof(header) + MAX_DATA_LENGTH, 0,
            (sockaddr*)&router_addr, &rlen) <= 0) {
        }
        memcpy(&header, recvbuffer, sizeof(header));
        //传输结束
        if (header.flag == OVER) {
            //发送OVER_ACK
            if (checksum((u_short*)&header, sizeof(header)) == 0) { if (endreceive())
                { return 1; }return 0; }
            else { cout << "[ERROR]:数据包出错, 正在等待重传" << endl; goto RECVSEQ0;
                }
        }
        cout << "[RECV]:" << endl << "seq:" << header.seq << " checksum:" <<
            checksum((u_short*)recvbuffer, sizeof(header) + MAX_DATA_LENGTH) << endl;
        if (header.seq == 0 && checksum((u_short*)recvbuffer,
            sizeof(header)+MAX_DATA_LENGTH) == 0) {
            cout << "[CHECKED]:成功接收seq0数据包" << endl;
            memcpy(message + messagepointer, recvbuffer + sizeof(header),
                header.length);
            messagepointer += header.length;
            break;
        }
        else {
            cout << "[ERROR]:数据包错误, 正在等待对方重新发送" << endl;
        }
    }
    header.ack = 1;
    header.seq = 0;
    header.checksum = calcksum((u_short*)&header, sizeof(header));
    memcpy(sendbuffer, &header, sizeof(header));
SENDACK1:
    if (sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen)
        == -1) {
        cout << "[Failed]:ack1发送失败" << endl;
        return -1;
    }
    clock_t start = clock();
RECVSEQ1:
    //设置为非阻塞模式
    ioctlsocket(server, FIONBIO, &unblockmode);
```

```
44      while (recvfrom(server, recvbuffer, sizeof(header) + MAX_DATA_LENGTH, 0,
            (sockaddr*)&router_addr, &rlen) <= 0) {
45          if (clock() - start > MAX_TIME) {
46              if (sendto(server, sendbuffer, sizeof(header), 0,
                    (sockaddr*)&router_addr, rlen) == -1) {
47                  cout << "[Failed]:ack1发送失败" << endl;
48                  return -1;
49              }
50              start = clock();
51              cout << "[ERROR]:ack1消息反馈超时" << endl;
52              goto SENDACK1;
53          }
54      }
55      memcpy(&header, recvbuffer, sizeof(header));
56      if (header.flag == OVER) {
57          //传输结束
58          if (checksum((u_short*)&header, sizeof(header) == 0)) { if (endreceive()) {
                return 1; }return 0; }
59          else { cout << "[ERROR]:数据包出错, 正在等待重传" << endl; goto RECVSEQ0; }
60      }
61      cout << "[RECV]:" << endl << "seq:" << header.seq << " checksum:" <<
            checksum((u_short*)recvbuffer, sizeof(header) + MAX_DATA_LENGTH) << endl;
62
63      if (header.seq == 1 && checksum((u_short*)recvbuffer,
            sizeof(header)+MAX_DATA_LENGTH)==0) {
64          cout << "[CHECKED]:成功接受seq1数据包" << endl;
65          memcpy(message + messagepointer, recvbuffer + sizeof(header), header.length);
66          messagepointer += header.length;
67      }
68      else {
69          cout << "[ERROR]:数据包损坏, 正在等待重新传输" << endl;
70          goto RECVSEQ1;
71      }
72      header.ack = 0;
73      header.seq = 1;
74      header.checksum = calcksum((u_short*)&header, sizeof(header));
75      memcpy(sendbuffer, &header, sizeof(header));
76      sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen);
77      goto RECVSEQ0;
78  }
```

接收端确认重传

3.6 日志输出

在客户端会输出一个传输日志，记录了报文总长度，分成多少个报文段转发，传输总时长和吞吐率。

```
1   void printlog() {
```

```
    cout << "--------------传输日志--------------" << endl;
    cout << "本次报文总长度为 " << messagepointer << "字节，共分为 " <<
        (messagepointer / 256) + 1 << "个报文段分别转发" << endl;
    double t = (ALLEND - ALLSTART) / CLOCKS_PER_SEC;
    cout << "本次传输的总时长为" <<t <<"秒"<< endl;
    t = messagepointer / t;
    cout << "本次传输吞吐率为" << t <<"字节每秒"<< endl;
}
```

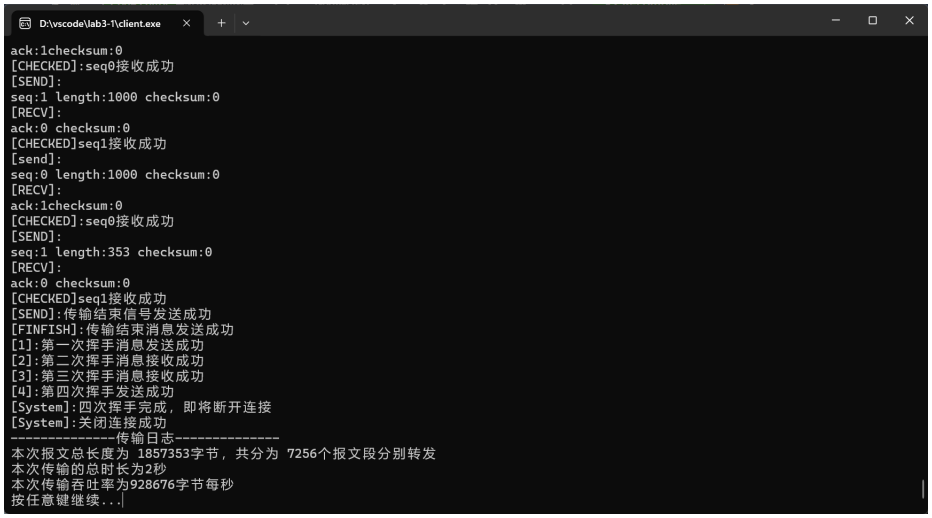
传输日志

3.7 性能测试

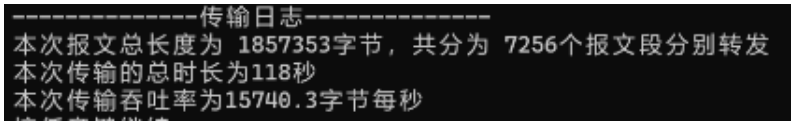
3.7.1 更改丢包率

设置窗口中传输的最大字节数为 1000，在传输过程中对传输文件的大小、传输时间以及吞吐率进行计算并展示。通过调节路由器的丢包率展示网络中丢包对数据传输的性能影响。

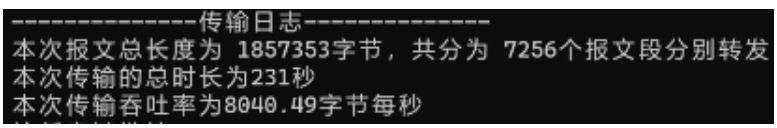
当网络中不发生丢包时，传输一个 185 万字节的数据需要大概 2 秒



当丢包率为 20% 时，传输一个 185 万字节的数据需要大概 118 秒，虽然丢包率只增加了 20%，但是传输速率却低了将近 50 倍，这是因为在停等机制中只有超时重传，所以一切错误的处理都必须等待固定的秒数（在本实验中设置的是 200ms）

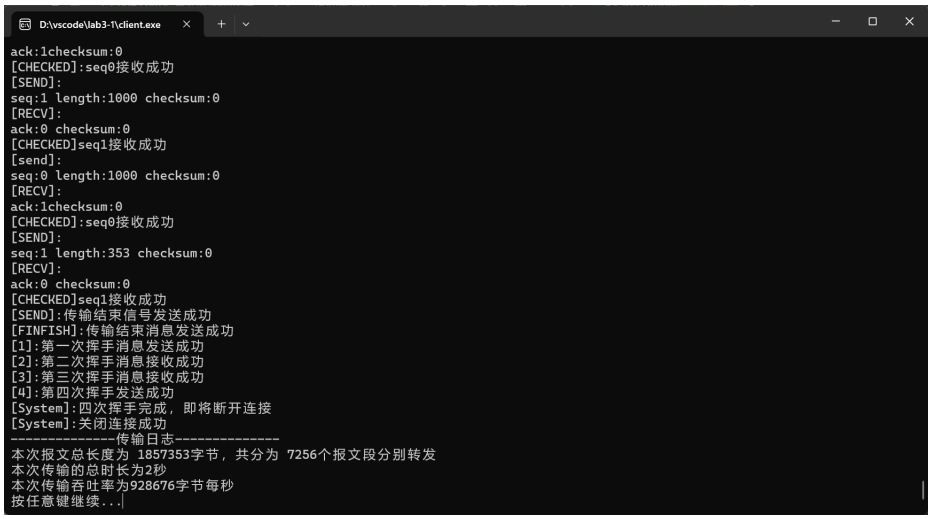


当丢包率为 50% 时，传输一个 185 万字节的数据需要大概 231 秒，相较于丢包率为 20% 的情况，传输时间稳步增长主要是因为其开销主要花在了等待时间上，传输时间相较于等待时间已经很小了，所以等待时间的翻倍最终也使传输时间翻倍。

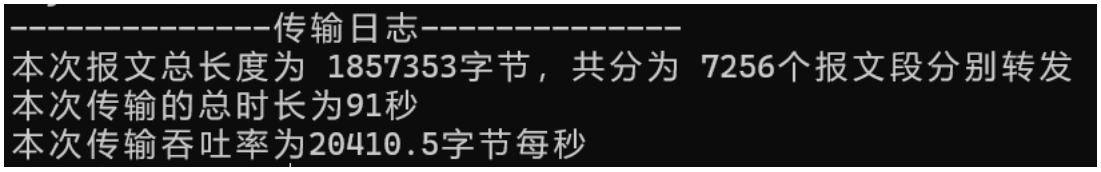


3.7.2 更改延时

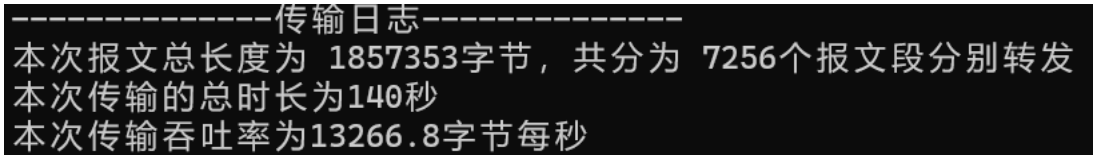
当延时设置为 0 时，传输一个 185 万字节的数据需要大概 2 秒



当延时设置为 20ms 时，传输一个 185 万字节的数据需要大概 91 秒。每个数据包在传输前需要等待 20 毫秒，这样的等待时间会叠加，导致传输时间明显增加。在这种情况下，数据包之间的等待时间占据了大部分时间，而实际的传输时间可能相对较短。传输时间的增加可能主要是由于等待而不是实际的数据传输引起的。



当延时设置为 50ms 时，传输一个 185 万字节的数据需要大概 140 秒。延时设置为 50ms 会导致更长的等待时间，从而增加了总的传输时间。在这种情况下，更多的时间花费在等待而不是实际的数据传输上。



4 问题与总结

- 三次握手的不可靠性：在网络延迟或丢包的情况下，三次握手可能不会成功完成。

  解决办法

  增加超时重传：如果在设定的时间内没有收到预期的响应，客户端或服务器应该重发握手消息。

  状态检查：在每一步握手过程中，检查收到的消息是否符合当前的状态和预期。

- 数据包损坏：在网络传输过程中，数据包可能会损坏。

  解决方法：

校验和增强：改进校验和算法以更可靠地检测数据损坏。

确认和重传机制：对损坏的数据包进行重传。

- 超时设置不当：超时设置不当可能导致过早重传或不必要的延迟。

  解决方法：

  动态调整超时时间：基于网络的当前状况动态调整超时时间。

  反馈循环：利用前几次消息传输的往返时间来估计合理的超时时间。

- 四次挥手的复杂性：在断开连接时，四次挥手过程可能因为消息丢失或延迟而变得复杂。

  解决方法：

  状态机逻辑：实现详细的状态机逻辑，处理不同的挥手阶段和异常情况。

  额外的确认消息：在四次挥手的最后增加额外的确认消息，确保双方都明确连接已断开。

- 网络条件变化：不同的网络条件（如高丢包率和高延迟）对协议的性能有显著影响。

  解决方法：

  适应性调整：根据网络状况调整消息大小、重传策略等。

  负载测试：在不同网络条件下进行负载测试，优化协议参数。