南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

Lab 3.3 基于 UDP 服务设计可靠传输协议并编程实现

徐俊智

年级：2022 级

专业：计算机科学与技术

指导教师：吴英

2024 年 12 月 15 日

目录

一、 实验目的

实验 3-3：在实验 3-2 的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

二、 实验要求

1. 实现单向传输。
2. 对于每个任务要求给出详细的协议设计。
3. 给出实现的拥塞控制算法的原理说明。
4. 完成给定测试文件的传输，显示传输时间和平均吞吐率。
5. 性能测试指标：吞吐率、文件传输时延，给出图形结果并进行分析。
6. 完成详细的实验报告。
7. 编写的程序应该结构清晰，具有较好的可读性。
8. 现场演示。
9. 提交程序源码、可执行文件和实验报告。

三、 UDP 报文段格式

```
class Message {
public:
    u_long flag;          // 伪首部
    u_short seq;          // 序列号
    u_short ack;          // 确认号
    u_long len;           // 数据部分长度
    u_long num;           // 数据包个数
    u_short checksum;     // 校验和
    char data[1024];      // 数据

    Message() { memset(this, 0, sizeof(Message)); }

    bool isSYN() { return this->flag & 1; }

    bool isACK() { return this->flag & 2; }

    bool isFIN() { return this->flag & 4; }

    bool isSTART() { return this->flag & 8; }

    bool isEND() { return this->flag & 16; }
```

Message 的成员有伪首部 flag，发送号 seq 和确认号 ack，数据部分长度 len，校验和 checksum，数据部分 data[1024]，其长度设置为 1024。

其中，flag 包含的属性有：SYN，ACK，FIN，START，END。SYN 和 FIN 分别用于建立和关闭连接，ACK 表示响应，START 和 END 表示文件传输的开始和结束。

其次，定义了查询 flag 是否包含某个属性的函数。

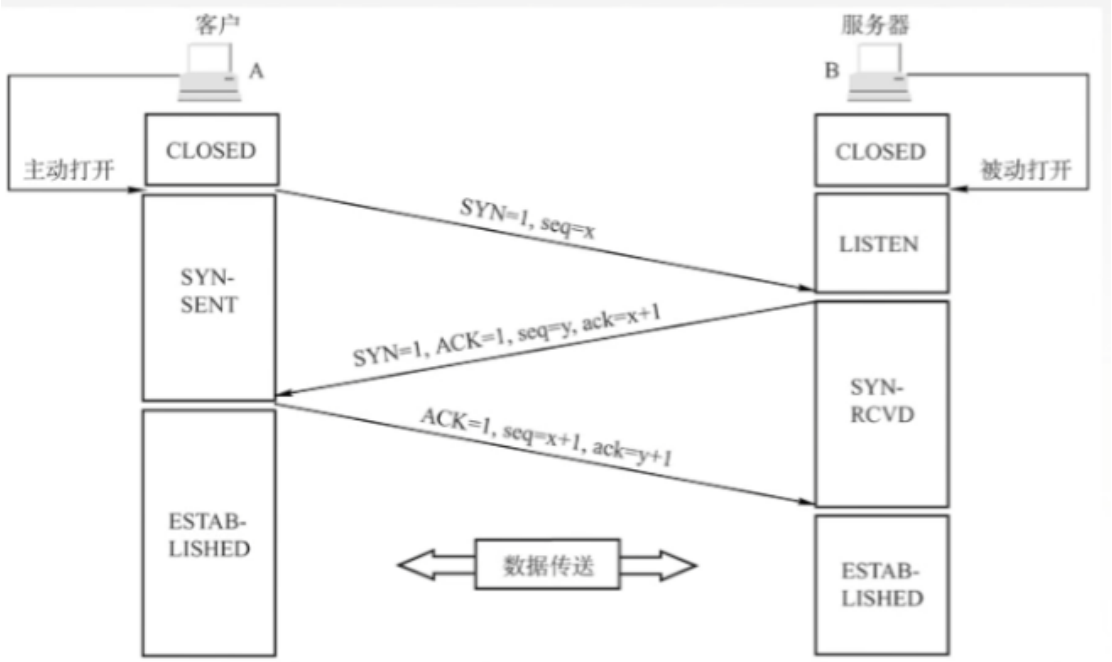
```
void setSYN() { this->flag |= 1; }
void setACK() { this->flag |= 2; }
void setFIN() { this->flag |= 4; }
void setSTART() { this->flag |= 8; }
void setEND() { this->flag |= 16; }
```

最后，定义了计算校验和的函数 `check_sum` 和检查数据包是否损坏的函数 packetIncorruption。

```
    void setChecksum() {
        int sum = 0;
        u_char* temp = (u_char*)this;
        for (int i = 0; i < 8; i++) {
            sum += (temp[i << 1] << 8) + temp[i << 1 | 1];
            while (sum >= 0x10000) {
                // 溢出
                int t = sum >> 16; // 将最高位回滚添加至最低位
                sum += t;
            }
        }
        this->checksum = ~(u_short)sum; // 按位取反，方便校验计算
    }

    bool packetIncorruption() {
        int sum = 0;
        u_char* temp = (u_char*)this;
        for (int i = 0; i < 8; i++) {
            sum += (temp[i << 1] << 8) + temp[i << 1 | 1];
            while (sum >= 0x10000) {
                // 溢出
                int t = sum >> 16; // 计算方法与设置校验和相同
                sum += t;
            }
        }
        // 把计算出来的校验和和报文中该字段的值相加，如果等于0xffff，则校验成功
        if (checksum + (u_short)sum == 65535)
            return false;
        return true;
    }
```

四、 建立连接 & 关闭连接

通过三次握手建立连接



- 第一次握手：客户端发出连接请求报文，SYN=1，seq=x=2000。
- 第二次握手：服务器端收到来自客户端的连接请求报文后，通过标志位 SYN=1 知道了客户端请求建立连接。然后服务器端向客户端发出确认报文，SYN=1，ACK=1，seq=y=1000，ack=x+1=2001。
- 第三次握手：客户端收到来自服务器端的确认报文后，检查 ACK 是否为 1、ack 是否为 x+1。如果正确，客户端向服务器端发出确认报文,ACK=1，seq=x+1=2001,ack=y+1=1001。

代码如下:

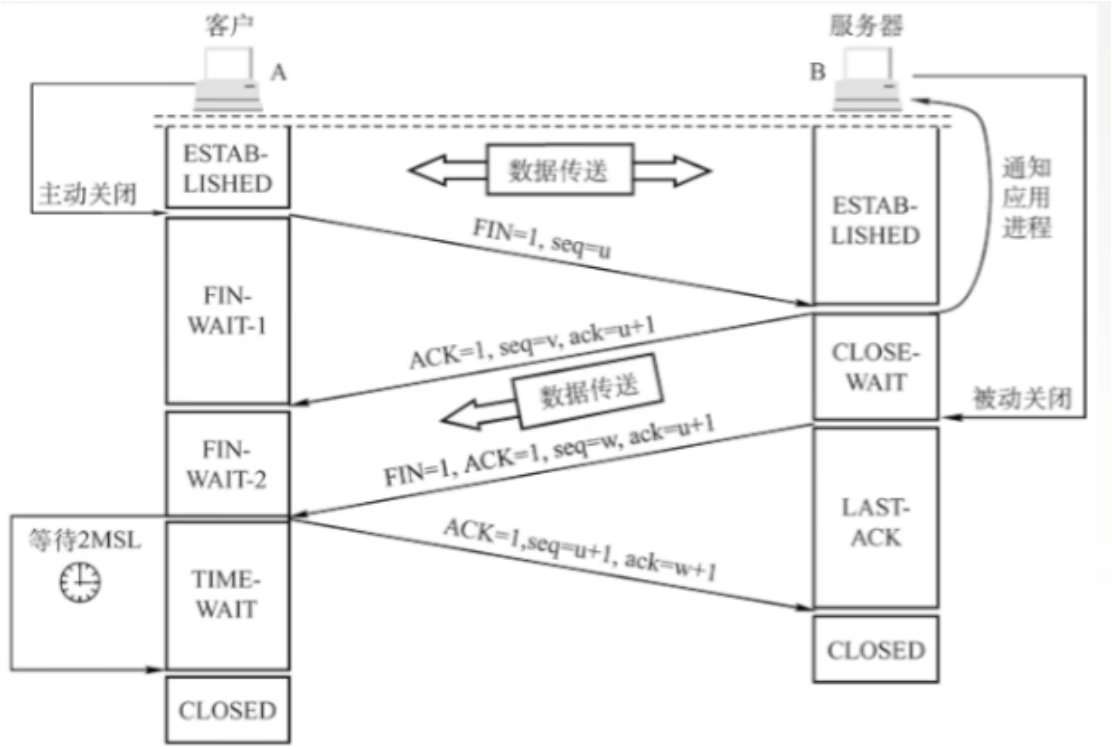
```
//Server.cpp
bool waitConnect() {
    // 设置套接字为非阻塞模式
    int mode = 1;
    ioctlsocket(socketServer, FIONBIO, (u_long FAR*) & mode);

    Message sendMsg, recvMsg;
    clock_t start;

    // 接收第一次握手消息
    while (1) {
        if (recvfrom(socketServer, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
            clientAddr, &len) != SOCKET_ERROR) {
            if (recvMsg.isSYN() && !recvMsg.packetIncorruption()) {
                cout << "服务器端接收到第一次握手消息! 第一次握手成功!" <<
                    endl;
```

```
                break;
            }
        }
    }

    // 发送第二次握手消息
    cout << "服务器端发送第二次握手消息! " << endl;
    sendMsg.setSYN();
    sendMsg.setACK();
    sendMsg.seq = 1000;
    sendMsg.ack = recvMsg.seq + 1;
    sendMsg.setChecksum();
    if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
        clientAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
        cout << "服务器端发送第二次握手消息失败!" << endl;
        cout << "当前网络状态不佳, 请稍后再试" << endl;
        return 0;
    }

    // 接收第三次握手消息, 超时重传
    start = clock();
    while (1) {
        if (recvfrom(socketServer, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
            clientAddr, &len) != SOCKET_ERROR) {
            if (recvMsg.isACK() && recvMsg.ack == sendMsg.seq + 1 && !recvMsg
                .packetIncorruption()) {
                cout << "服务器端接收到第三次握手消息! 第三次握手成功! " <<
                    endl;
                break;
            }
        }
        if (clock() - start > RTO) {
            cout << "第二次握手超时,服务器端重新发送第二次握手消息" << endl;
            if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
                clientAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
                cout << "服务器端发送第二次握手消息失败!" << endl;
                cout << "当前网络状态不佳, 请稍后再试" << endl;
                return 0;
            }
            start = clock();
        }
    }

    // 设置套接字为阻塞模式
    mode = 0;
    ioctlsocket(socketServer, FIONBIO, (u_long FAR*) & mode);

    return 1;
```

```
}

//Client.cpp
bool waitConnect() {
    Message sendMsg, recvMsg;
    clock_t start;

    // 设置套接字为非阻塞模式
    int mode = 1;
    ioctlsocket(socketClient, FIONBIO, (u_long FAR*) & mode);

    // 发送第一次握手消息
    cout << "尝试建立连接! 客户端发送第一次握手消息" << endl;
    sendMsg.setSYN();
    sendMsg.seq = 2000;
    sendMsg.setChecksum();
    if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
        serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
        cout << "客户端发送第一次握手消息失败!" << endl;
        cout << "当前网络状态不佳, 请稍后再试" << endl;
        return 0;
    }

    // 接收第二次握手消息, 超时重传
    start = clock();
    while (1) {
        if (recvfrom(socketClient, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
            serverAddr, &len) != SOCKET_ERROR) {
            if (recvMsg.isSYN() && recvMsg.isACK() && recvMsg.ack == sendMsg.
                seq + 1 && !recvMsg.packetIncorruption()) {
                cout << "客户端接收到第二次握手消息! 第二次握手成功!" << endl
                    ;
                break;
            }
        }
        if (clock() - start > RTO) {
            cout << "第一次握手超时,客户端重新发送第一次握手消息" << endl;
            if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
                serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
                cout << "客户端发送第一次握手消息失败!" << endl;
                cout << "当前网络状态不佳, 请稍后再试" << endl;
                return 0;
            }
            start = clock();
        }
    }

    // 发送第三次握手消息
```

```
101     cout << "客户端发送第三次握手消息" << endl;
102     sendMsg.setACK();
103     sendMsg.seq = 2001;
104     sendMsg.ack = recvMsg.seq + 1;
105     sendMsg.setChecksum();
106     if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
            serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
107         cout << "客户端发送第三次握手消息失败!" << endl;
108         cout << "当前网络状态不佳，请稍后再试" << endl;
109         return 0;
110     }
111
112     return 1;
113 }
```

通过四次挥手关闭连接



- 第一次挥手：客户端发出连接释放报文，FIN=1，seq=u=3000。
- 第二次挥手：服务器端收到来自客户端的连接请求报文后，通过标志位 FIN=1 知道了客户端请求释放连接。然后服务器端向客户端发出确认报文,ACK=1,seq=v=4000,ack=u+1=3001。
- 第三次挥手:当服务器端确认数据传输完毕后,向客户端发送连接释放报文,FIN=1,ACK=1,seq=w=5000，ack=u+1=3001。
- 第四次挥手：客户端收到来自服务器端的连接释放报文后，通过标志位 FIN=1 知道了服务器端请求释放连接,然后客户端向服务器发出确认报文,ACK=1,seq=u+1=3001,ack=w+1=5001。

代码如下:

```cpp
//Server.cpp
bool closeConnect(Message recvMsg) {
    Message sendMsg;
    clock_t start;

    /*
        第一次挥手在recv_file函数里面处理
    */

    // 设置套接字为非阻塞模式
    int mode = 1;
    ioctlsocket(socketServer, FIONBIO, (u_long FAR*) & mode);

    // 发送第二次挥手消息
    cout << "服务器端发送第二次挥手消息! " << endl;
    sendMsg.setACK();
    sendMsg.seq = 4000;
    sendMsg.ack = recvMsg.seq + 1;
    sendMsg.setChecksum();
    if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
        clientAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
        cout << "服务器端发送第二次挥手消息失败!" << endl;
        cout << "当前网络状态不佳, 请稍后再试" << endl;
        return 0;
    }

    // 发送第三次挥手消息
    cout << "服务器端发送第三次挥手消息! " << endl;
    sendMsg.setFIN();
    sendMsg.setACK();
    sendMsg.seq = 5000;
    sendMsg.ack = recvMsg.seq + 1;
    sendMsg.setChecksum();
    if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
        clientAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
        cout << "服务器端发送第三次挥手消息失败!" << endl;
        cout << "当前网络状态不佳, 请稍后再试" << endl;
        return 0;
    }

    // 接收第四次挥手消息, 超时重传
    start = clock();
    while (1) {
        if (recvfrom(socketServer, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
            clientAddr, &len) != SOCKET_ERROR) {
            if (recvMsg.isACK() && recvMsg.ack == sendMsg.seq + 1 && !recvMsg
                .packetIncorruption()) {
```

```
44                  cout << "服务器端接收到第四次挥手消息! 第四次挥手成功! " <<
                        endl;
45                  break;
46              }
47          }
48          if (clock() - start > RTO) {
49              cout << "第三次挥手超时,服务器端重新发送第三次挥手消息" << endl;
50              if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
                    clientAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
51                  cout << "服务器端发送第三次挥手消息失败!" << endl;
52                  cout << "当前网络状态不佳, 请稍后再试" << endl;
53                  return 0;
54              }
55              start = clock();
56          }
57      }
58      return 0;
59  }
60
61
62  //Client.cpp
63  bool closeConnect() {
64      Message sendMsg, recvMsg;
65      clock_t start;
66
67      // 发送第一次挥手消息
68      cout << "尝试关闭连接! 客户端发送第一次挥手消息" << endl;
69      sendMsg.setFIN();
70      sendMsg.seq = 3000;
71      sendMsg.setChecksum();
72      if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
            serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
73          cout << "客户端发送第一次挥手消息失败!" << endl;
74          cout << "当前网络状态不佳, 请稍后再试" << endl;
75          return 0;
76      }
77
78      // 接收第二次挥手消息, 超时重传
79      start = clock();
80      while (1) {
81          if (recvfrom(socketClient, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
                serverAddr, &len) != SOCKET_ERROR) {
82              if (recvMsg.isACK() && recvMsg.ack == sendMsg.seq + 1 && !recvMsg
                    .packetIncorruption()) {
83                  cout << "客户端接收到第二次挥手消息! 第二次挥手成功! " <<
                        endl;
84                  break;
85              }
```

```
86          }
87          if (clock() - start > RTO) {
88              cout << "第一次挥手超时,客户端重新发送第一次挥手消息" << endl;
89              if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
                    serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
90                  cout << "客户端发送第一次挥手消息失败!" << endl;
91                  cout << "当前网络状态不佳, 请稍后再试" << endl;
92                  return 0;
93              }
94              start = clock();
95          }
96      }
97
98      // 接收第三次挥手消息, 超时重传
99      start = clock();
100     while (1) {
101         if (recvfrom(socketClient, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
                serverAddr, &len) != SOCKET_ERROR) {
102             if (recvMsg.isACK() && recvMsg.ack == sendMsg.seq + 1 && !recvMsg
                    .packetIncorruption()) {
103                 cout << "客户端接收到第三次挥手消息! 第三次挥手成功! " <<
                        endl;
104                 break;
105             }
106         }
107         if (clock() - start > RTO) {
108             cout << "第二次挥手超时,客户端重新发送第二次挥手消息" << endl;
109             if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
                    serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
110                 cout << "客户端发送第二次挥手消息失败!" << endl;
111                 cout << "当前网络状态不佳, 请稍后再试" << endl;
112                 return 0;
113             }
114             start = clock();
115         }
116     }
117
118     // 发送第四次挥手消息
119     cout << "客户端发送第四次挥手消息! " << endl;
120     sendMsg.setACK();
121     sendMsg.seq = 3001;
122     sendMsg.ack = recvMsg.seq + 1;
123     sendMsg.setChecksum();
124     if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&
            serverAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
125         cout << "客户端发送第四次挥手消息失败!" << endl;
126         cout << "当前网络状态不佳, 请稍后再试" << endl;
127         return 0;
```

```
    }

    return 1;
}
```

因为建立和关闭连接时都是客户先发起，服务器需要先阻塞在一个 while 循环里不断的接收消息，直到接收了客户发来的信息，才能继续往下执行。

五、 差错检测

计算检验和主要有三个步骤:

- 求和：把需要校验的数据看成以 16 位为单位的数字组成，依次进行二进制求和。
- 回卷：求和后超过 16 位的加到低 16 位。
- 取反：最后结果取反码就是检验和。

客户端发送数据包的时候，通过 setChecksum 函数设置校验和。服务器端接收数据包的时候，通过 packetIncorruption 函数验证数据包是否正确。代码如下:

```
void setChecksum() {
    this->checksum = 0;// 清0校验和字段
    int dataLen = this->len;// 数据部分长度
    int paddingLen = (16 - (dataLen % 16)) % 16;// 数据部分需要填0
    char* paddedData = new char[dataLen + paddingLen];// 填充后数据
    memcpy(paddedData, this->data, dataLen);
    memset(paddedData + dataLen, 0, paddingLen);
    // 分段求和，并处理溢出
    u_short* buffer = (u_short*)this;
    int sum = 0;
    for (int i = 0; i < (sizeof(Message) + paddingLen) / 2; i++) {
        sum += buffer[i];
        if (sum > 0xFFFF) {
            sum = (sum & 0xFFFF) + (sum >> 16);
        }
    }
    // 计算结果取反写入校验和字段
    this->checksum = ~sum;
    // 释放动态分配的内存
    delete[] paddedData;
}
bool packetIncorruption() {
    // 计算数据长度并填充
    int dataLen = this->len;
    int paddingLen = (16 - (dataLen % 16)) % 16;
    // 使用动态内存分配
    char* paddedData = new char[dataLen + paddingLen];
    memcpy(paddedData, this->data, dataLen);
    memset(paddedData + dataLen, 0, paddingLen);
```

```
    // 进行16 - bit段反码求和
    u_short* buffer = (u_short*)this;
    int sum = 0;
    for (int i = 0; i < (sizeof(Message) + paddingLen) / 2; i++) {
        sum += buffer[i];
        if (sum > 0xFFFF) {
            sum = (sum & 0xFFFF) + (sum >> 16);
        }
    }
    // 如果计算结果为全为1则无差错; 否则, 有差错
    bool result = sum != 0xFFFF;
    // 释放动态分配的内存
    delete[] paddedData;
    return result;
}
```

六、 发送文件

客户端首先需要检测用户输入，如果用户输入的是“quit”，说明用户要关闭连接，那么进入挥手模式。

如果用户输入的是文件名，我们就以二进制方式打开文件（因为要发送的文件包含图片等非文本文件）将文件拆成若干个 1024Bytes 大小的数据包。

我们要发送的第一个数据包内容是文件名，发送成功后正式开始发送文件内容。

为了实现并行的发送和接收，我们在 `send_file` 函数中创建两个线程，分别是发送线程 `send_thread` 和接收线程 `recv_thread`。

```
// 发送线程
void send_thread(SOCKET socketClient, sockaddr_in& routerAddr, ifstream& in,
    int filePtrLoc, int packetNum) {
    Message sendMsg;

    // 窗口内发送数据
    while (!send_over) {
        {
            unique_lock<mutex> lock(seq_mutex);// 加锁
            if (next_seq <= packetNum && (next_seq - base) < min(cwnd, rwnd))
                {

                if (next_seq == packetNum) {
                    in.read(sendMsg.data, filePtrLoc);
                    sendMsg.len = filePtrLoc;
                    sendMsg.setEND();
                    filePtrLoc = 0;
                }
                else {
                    in.read(sendMsg.data, 1024);
                    sendMsg.len = 1024;
```

```
                filePtrLoc -= 1024;
            }

            // 发送数据包
            sendMsg.seq = next_seq;
            sendMsg.setChecksum();
            if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&routerAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
                cout << "发送数据包失败!" << endl;
            }
            // 记录发送时间
            send_buffer[next_seq] = sendMsg;
            send_times[next_seq] = clock();

            cout << "发送seq:" << next_seq << endl;
            cout << "checksum:" << sendMsg.checksum << endl;
            cout << "ssthresh:" << ssthresh << endl;
            cout << "swnd:" << min(cwnd, rwnd) << endl;
            cout << "base:" << base << endl;

            next_seq++;

            cout << "next_seq:" << next_seq << endl;
            cout << "top:" << base + min(cwnd, rwnd) << endl << endl;
        }
        lock.unlock(); // 解锁
    }
    {
        unique_lock<mutex> lock(seq_mutex);// 加锁
        // 超时重传
        for (auto it = send_buffer.begin(); it != send_buffer.end(); it++) {
            if (clock() - send_times[it->first] > TIMEOUT) {
                cout << "应答超时, 重新发送已发送未确认的数据包" << endl;

                Message sendMsg = it->second;
                if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&routerAddr, sizeof(SOCKADDR)) == SOCKET_ERROR) {
                    cout << "重传数据包失败!" << endl;
                }
                // 重置计时器
                send_times[it->first] = clock();

                cout << "发送seq:" << it->first << endl;
                cout << "checksum: " << sendMsg.checksum << endl;
                cout << "ssthresh:" << ssthresh << endl;
```

```
63                      cout << "swnd:" << min(cwnd, rwnd) << endl;
64                      cout << "base: " << base << endl;
65                      cout << "next_seq: " << next_seq << endl;
66                      cout << "top: " << base + min(cwnd, rwnd) << endl << endl
                            ;
67
68                      cout << "重新发送完毕" << endl;
69                  }
70              }
71              lock.unlock(); // 解锁
72          }
73
74      }
75  }
76
77  // 接收线程
78  void recv_thread(SOCKET socketClient, int packetNum) {
79      Message recvMsg;
80      clock_t start;
81      int expected_ack = 2; // 期望收到的ack
82      int prev_ack = 0; // 上一次收到的确认号
83
84      while (!send_over) {
85          if (recvfrom(socketClient, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
                routerAddr, &len) != SOCKET_ERROR) {
86              {
87                  unique_lock<mutex> lock(seq_mutex);// 加锁
88                  if (recvMsg.isACK() && !recvMsg.packetIncorrection()) {
89                      if (recvMsg.ack > prev_ack) { // 新的 ack
90                          cout << "接收新的ack:" << recvMsg.ack << endl;
91                          dup_ack_count = 0; // 重置重复 ACK 计数
92                          if (recvMsg.ack >= expected_ack) {
93                              // 移除已确认的数据包
94                              auto it = send_buffer.begin();
95                              while (it != send_buffer.end() && it->first <
                                    recvMsg.ack) {
96                                  it = send_buffer.erase(it); // 从发送缓冲区中
                                        删去对应报文
97                                  send_times.erase(recvMsg.ack); // 同时移除计
                                        时器
98                              }
99                              base = recvMsg.ack;
100
101                             if (cwnd < ssthresh) {
102                                 // 慢启动
103                                 cout << "慢启动:" << endl;
104                                 cwnd *= 2;
105                             }
```

```
106                         else {
107                             // 拥塞避免
108                             cout << "拥塞避免:" << endl;
109                             cwnd += 1;
110                         }
111                         expected_ack += min(cwnd, rwnd);
112                     }
113                     prev_ack = recvMsg.ack;
114 
115                     // 展示窗口情况
116                     cout << "ssthresh:" << ssthresh << endl;
117                     cout << "swnd:" << min(cwnd, rwnd) << endl;
118                     cout << "base:" << base << endl;
119                     cout << "next_seq:" << next_seq << endl;
120                     cout << "top:" << base + min(cwnd, rwnd) << endl <<
                            endl;
121 
122                 }
123                 else if (recvMsg.ack == prev_ack) { //重复 ack
124                     if (dup_ack_count < 3) {
125                         dup_ack_count++;
126                         cout << "接收到第" << dup_ack_count << "个重复ack
                                :" << recvMsg.ack << endl;
127                         cwnd += 1;
128                     }
129                     if (dup_ack_count == 3) { // 收到第3个重复 ACK
130                         // 快重传
131                         cout << "快重传:" << endl;
132                         Message sendMsg = send_buffer.find(prev_ack)->
                                second;
133                         if (sendto(socketClient, (char*)&sendMsg, BUFFER,
                                0, (SOCKADDR*)&routerAddr, sizeof(SOCKADDR))
                                == SOCKET_ERROR) {
134                             cout << "重传数据包失败!" << endl;
135                         }
136                         cout << "发送seq:" << sendMsg.seq << endl;
137                         cout << "checksum:" << sendMsg.checksum << endl;
138                         cout << "ssthresh:" << ssthresh << endl << endl;
139 
140                         // 快恢复
141                         cout << "快恢复:" << endl;
142                         ssthresh = cwnd / 2;
143                         cwnd = ssthresh + 3;
144 
145                         // 展示窗口情况
146                         cout << "ssthresh:" << ssthresh << endl;
147                         cout << "swnd:" << min(cwnd, rwnd) << endl;
148                         cout << "base:" << base << endl;
```

```
149                             cout << "next_seq:" << next_seq << endl;
150                             cout << "top:" << base + min(cwnd, rwnd) << endl
                                    << endl;
151
152                             dup_ack_count = INT_MAX;
153                         }
154                     }
155                     // 如果所有文件传输结束
156                     if (recvMsg.ack == packetNum + 1) {
157                         cout << "文件传输完成! " << endl;
158                         send_over = true;
159                     }
160                 }
161                 lock.unlock(); // 解锁
162             }
163         }
164     }
165 }
```

1. 慢启动算法

当 cwnd<ssthresh 时，使用慢启动算法

先发送少量数据试探一下网络的拥塞程度，每个传输轮次拥塞窗口 cwnd 按指数增长。

```
1 if (cwnd < ssthresh) {
2     // 慢启动
3     cout << "慢启动:" << endl;
4     cwnd *= 2;
5 }
```

2. 拥塞避免算法

当 cwnd>ssthresh 时，停止使用慢启动算法而改用拥塞避免算法。

```
1 else {
2     // 拥塞避免
3     cout << "拥塞避免:" << endl;
4     cwnd += 1;
5 }
```

3. 快重传算法

有时个别报文段会在网络中丢失，但实际上网络并未发生拥塞。这将导致发送方超时重传，并误认为网络发生了拥塞，把拥塞窗口 cwnd 又设置为最小值 1，错误地启动慢启动算法，因而降低了传输效率。

快重传算法要求接收方收到数据，立即发送确认。接收方如果收到失序的报文段，发出对已收到的报文段的重复确认。发送方一旦收到 3 个连续的重复确认，就将相应的报文段立即重传

```
else if (recvMsg.ack == prev_ack) { //重复 ack
    if (dup_ack_count < 3) {
        dup_ack_count++;
        cout << "接收到第" << dup_ack_count << "个重复ack:" << re
        cwnd += 1;
    }
    if (dup_ack_count == 3) { // 收到第3个重复 ACK
        // 快重传
        cout << "快重传:" << endl;
        Message sendMsg = send_buffer.find(prev_ack)->second;
        if (sendto(socketClient, (char*)&sendMsg, BUFFER, 0, (SOC
            cout << "重传数据包失败!" << endl;
        }
        cout << "发送seq:" << sendMsg.seq << endl;
        cout << "checksum:" << sendMsg.checksum << endl;
        cout << "ssthresh:" << ssthresh << endl << endl;
```

4. 快恢复算法

发送方一旦收到 3 个重复确认，就知道现在只是丢失了个别的报文段。于是不启动慢启动算法，而执行快恢复算法。

```
// 快恢复
cout << "快恢复:" << endl;
ssthresh = cwnd / 2;
cwnd = ssthresh + 3;
```

加 3 是因为发送方收到 3 个重复的确认，就表明有 3 个数据报文段已经离开了网络，这 3 个报文段不再消耗网络资源而是停留在接收方的接收缓存中，可见现在网络中不是堆积了报文段而是减少了 3 个报文段。因此可以适当把拥塞窗口扩大些。

七、 接收文件

如果服务器端接收到 FIN 报文，说明客户端准备断开连接，进入挥手模式（需要注意的是，接收第一次挥手的消息已经在 `recv_file` 函数中处理了，closeConnect 函数只需处理剩下的三次挥手即可）。

如果服务器端接收到 START 报文，说明客户端准备开始发送文件，服务器端以二进制方式打开文件，并开始写入接收到的数据。

```
void recv_file() {
    cout << "服务器正在等待接收文件中......" << endl;
    Message recvMsg, sendMsg;
    clock_t start, end;
    char filePath[20];
    string outputPath;
    ofstream out;
    int dataAmount = 0;
```

```
9     int packetNum;
10
11    // 接收文件名
12    while (1) {
13        if (recvfrom(socketServer, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
              routerAddr, &len) != SOCKET_ERROR) {
14            // 接收第一次挥手信息
15            if (recvMsg.isFIN() && !recvMsg.packetIncorrection()) {
16                cout << "客户端准备断开连接! 进入挥手模式! " << endl;
17                cout << "服务器端接收到第一次挥手消息! 第一次挥手成功!" <<
                      endl;
18                closeConnect(recvMsg);
19                quit = true;
20                return;
21            }
22            if (recvMsg.isSTART() && !recvMsg.packetIncorrection()) {
23                ZeroMemory(filePath, 20);
24                memcpy(filePath, recvMsg.data, recvMsg.len);
25                outputPath = "./output/" + string(filePath);
26                out.open(outputPath, ios::out | ios::binary);//以写入模式、二
                      进制模式打开文件
27                cout << "文件名为: " << filePath << endl;
28                cout << "接收到的seq:" << recvMsg.seq << endl;
29                cout << "checksum: " << recvMsg.checksum << endl;
30
31                if (!out.is_open()) {
32                    cout << "文件打开失败! ! ! " << endl;
33                    exit(1);
34                }
35
36                packetNum = recvMsg.num;
37                cout << "文件" << filePath << "有" << packetNum << "个数据包"
                      << endl;
38
39                // 发送ack给客户端
40                sendMsg.setACK();
41                sendMsg.ack = recvMsg.seq + 1;
42                sendMsg.setChecksum();
43                if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (
                      SOCKADDR*)&routerAddr, sizeof(SOCKADDR)) == SOCKET_ERROR)
                       {
44                    cout << "服务器端发送ack报文失败!" << endl;
45                    cout << "当前网络状态不佳, 请稍后再试" << endl;
46                    return;
47                }
48                break;
49            }
50        }
```

```
51      }
52
53      // 设置套接字为阻塞模式
54      int mode = 0;
55      ioctlsocket(socketServer, FIONBIO, (u_long FAR*) & mode);
56
57      // 开始接收文件内容
58      cout << "服务器端开始接收文件内容! " << endl << endl;
59      int expected_seq = 1;
60      start = clock();
61      while (1) {
62          if (recvfrom(socketServer, (char*)&recvMsg, BUFFER, 0, (SOCKADDR*)&
                routerAddr, &len) != SOCKET_ERROR) {
63              // 如果校验和错误
64              if (recvMsg.packetIncorrection()) {
65                  cout << "checksum错误! " << endl;
66                  cout << "checksum: " << recvMsg.checksum << endl;
67                  continue;
68              }
69              // 存储接收到的数据包
70              recv_buffer[recvMsg.seq] = recvMsg;
71              cout << "接收seq:" << recvMsg.seq << endl;
72              cout << "checksum:" << recvMsg.checksum << endl;
73
74              if (recvMsg.seq == expected_seq) {
75
76                  auto last_seq = recv_buffer.rbegin()->first;
77
78                  // 查找下一个期望的序列号
79                  bool foundGap = false;
80                  for (int i = expected_seq + 1; i <= last_seq; ++i) {
81                      if (recv_buffer.find(i) == recv_buffer.end()) {
82                          // 找到第一个gap, 即下一个期望的序列号
83                          expected_seq = i;
84                          foundGap = true;
85                          break;
86                      }
87                  }
88                  if (!foundGap) {
89                      // 如果没有找到gap, 那么expected_seq就是最后一个序列号加1
90                      expected_seq = last_seq + 1;
91                  }
92
93                  // 发送ack给客户端
94                  sendMsg.setACK();
95                  sendMsg.ack = expected_seq;
96                  sendMsg.setChecksum();
97                  cout << "发送ack:" << sendMsg.ack << endl << endl;
```

```
98                  if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (
                        SOCKADDR*)&routerAddr, sizeof(SOCKADDR)) == SOCKET_ERROR)
                         {
99                      cout << "服务器端发送ack报文失败!" << endl;
100                     cout << "当前网络状态不佳, 请稍后再试" << endl;
101                     return;
102                 }
103             }
104             // 如果接收到的不是期望接收的seq
105             else {
106                 // 发送累计确认的ack给客户端
107                 cout << "期望接收seq:" << expected_seq << endl;
108
109                 sendMsg.setACK();
110                 sendMsg.ack = expected_seq;
111                 sendMsg.setChecksum();
112                 cout << "发送累计确认ack:" << sendMsg.ack << endl << endl;
113                 if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (
                        SOCKADDR*)&routerAddr, sizeof(SOCKADDR)) == SOCKET_ERROR)
                         {
114                     cout << "服务器端发送ack报文失败!" << endl;
115                     cout << "当前网络状态不佳, 请稍后再试" << endl;
116                     return;
117                 }
118             }
119
120             // 如果接收到所有数据包
121             if (expected_seq == packetNum + 1) {
122                 cout << "接收缓冲区写入文件" << endl << endl;
123                 // 以追加模式打开文件, 并写入文件
124                 ofstream out(outputPath, ios::app | std::ios::binary);
125                 // 遍历接收缓冲区, 按序列号从小到大顺序将数据包写入文件
126                 for (auto it = recv_buffer.begin(); it != recv_buffer.end();
                        ++it) {
127                     const Message& bufferRecvMsg = it->second;
128                     out.write(bufferRecvMsg.data, bufferRecvMsg.len);
129                     dataAmount += bufferRecvMsg.len;
130                 }
131                 out.close();
132                 recv_buffer.clear(); // 清空接收缓冲区
133
134
135                 end = clock();
136                 cout << "接收文件成功! " << endl;
137                 out.close();
138                 out.clear();
139
140                 double TotalTime = (double)(end - start) / CLOCKS_PER_SEC;
```

```
                cout << "传输总时间" << TotalTime << "s" << endl;
                cout << "吞吐率" << (double)dataAmount / TotalTime << "bytes
                    /s" << endl << endl;

                return;
            }

        }
    }
}
```

1. 确认重传

服务器端先将接收到的数据包存到接收缓冲区，然后判断 seq 是否为期望的序列号。

如果是期望的序列号，就寻找下一个期望的序列号（接收缓冲区的第一个 gap 或者接收缓冲区最后一个序列号加 1)。最后将期望的序列号作为 ack 的内容发送给客户端。

如果不是期望的序列号，说明发送丢包，就将期望收到的序列号作为 ack 报文的内容发送给客户端。

如果接收到所有数据包，则尝试写入文件，并计算传输总时间和吞吐率。

```
if (recvMsg.seq == expected_seq) {
    auto last_seq = recv_buffer.rbegin()->first;
    // 查找下一个期望的序列号
    bool foundGap = false;
    for (int i = expected_seq + 1; i <= last_seq; ++i) {
        if (recv_buffer.find(i) == recv_buffer.end()) {
            // 找到第一个gap，即下一个期望的序列号
            expected_seq = i;
            foundGap = true;
            break;
        }
    }
    if (!foundGap) {
        // 如果没有找到gap，那么expected_seq就是最后一个序列
        expected_seq = last_seq + 1;
    }
    // 发送ack给客户端
    sendMsg.setACK();
    sendMsg.ack = expected_seq;
    sendMsg.setChecksum();
    cout << "发送ack:" << sendMsg.ack << endl << endl;
    if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOC
        cout << "服务器端发送ack报文失败!" << endl;
        cout << "当前网络状态不佳，请稍后再试" << endl;
        return;
    }
}
// 如果接收到的不是期望接收的seq
```

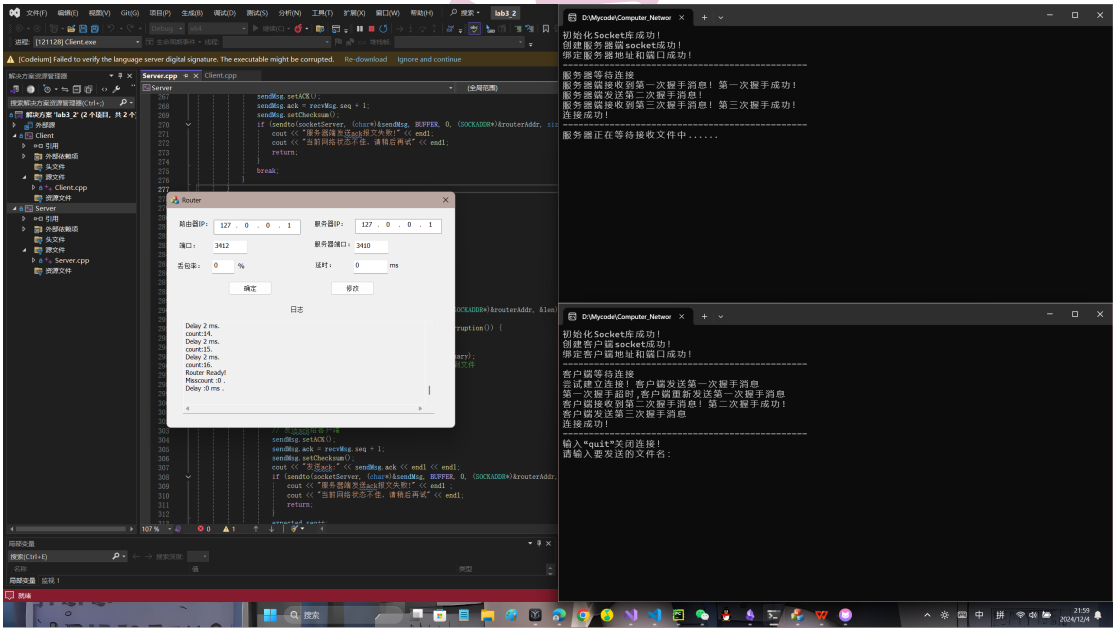
```
else {
    // 发送累计确认的ack给客户端
    cout << "期望接收seq:" << expected_seq << endl;
    sendMsg.setACK();
    sendMsg.ack = expected_seq;
    sendMsg.setChecksum();
    cout << "发送累计确认ack:" << sendMsg.ack << endl << endl
    if (sendto(socketServer, (char*)&sendMsg, BUFFER, 0, (SOC
        cout << "服务器端发送ack报文失败!" << endl;
        cout << "当前网络状态不佳，请稍后再试" << endl;
        return;
    }
}
```

八、 测试

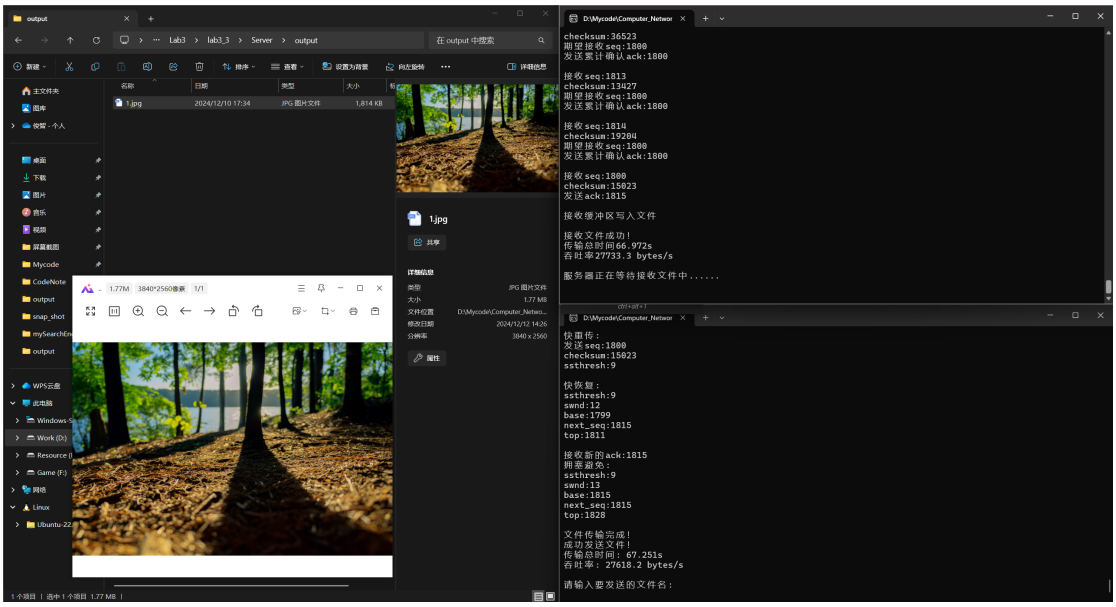
我们分别定义了服务器端、客户端、router 的端口号，IP 统一使用 127.0.0.1。

```
#define SERVER_PORT 3410
#define CLIENT_PORT 3411
#define ROUTER_PORT 3412
```

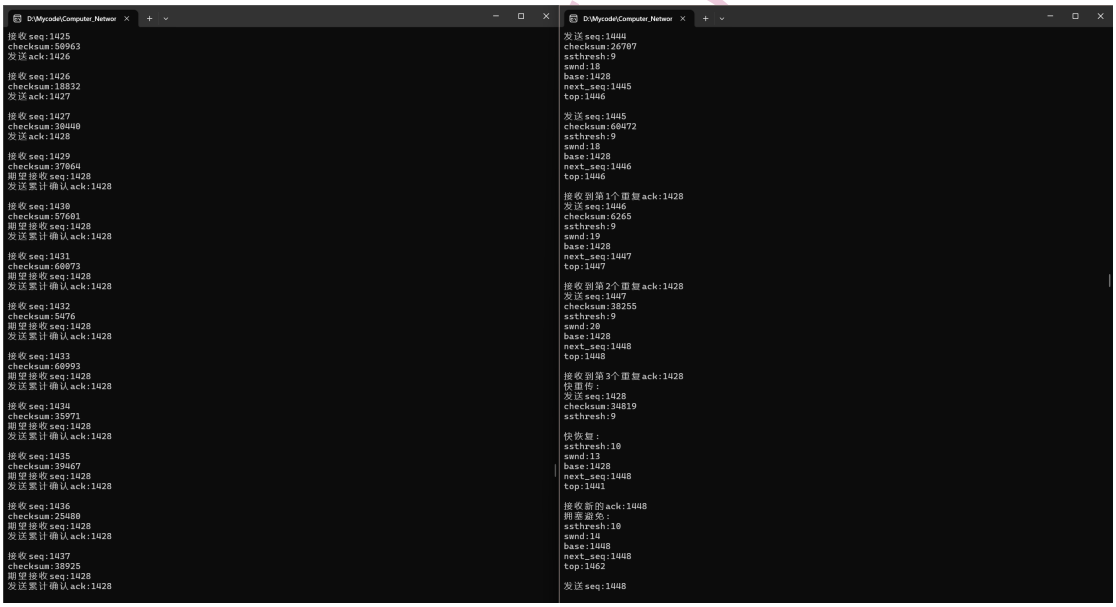
打开路由程序，设置 0% 的丢包率和 0ms 延迟，开始运行，首先建立连接。



接下来发送文件“1.jpg”。检查 Server output 目录，出现文件“1.jpg”，且与客户端发送的文件大小相等。



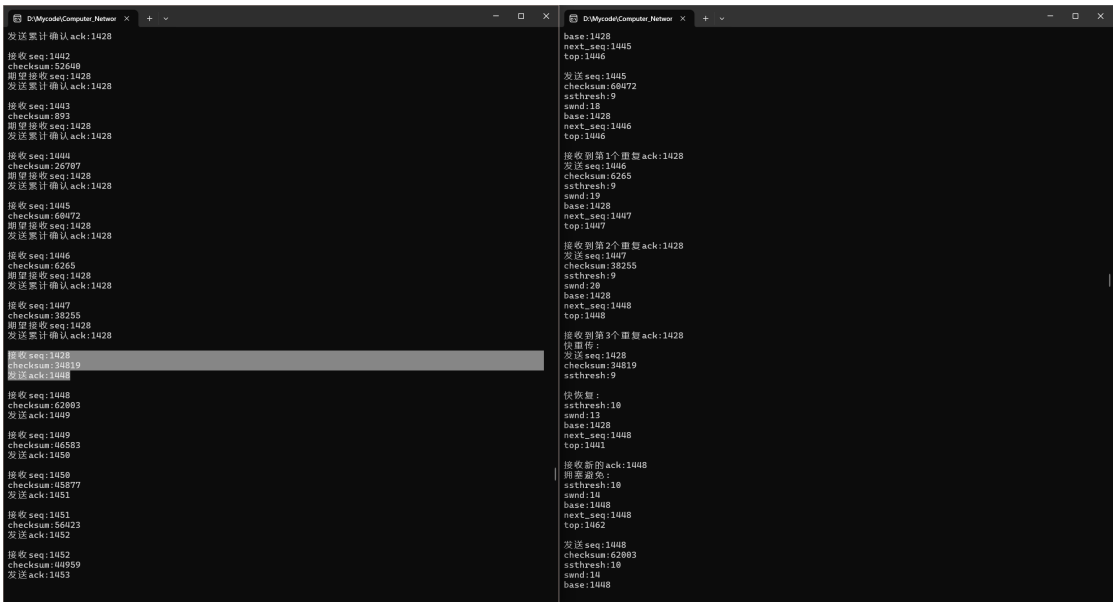
接下来测试快重传，打开路由程序，设置 3% 的丢包率和 2ms 延迟, 开始运行。



客户端发送的过程中发生丢包，seq 为 1428，服务器端再接受数据包，均返回 ack1428。

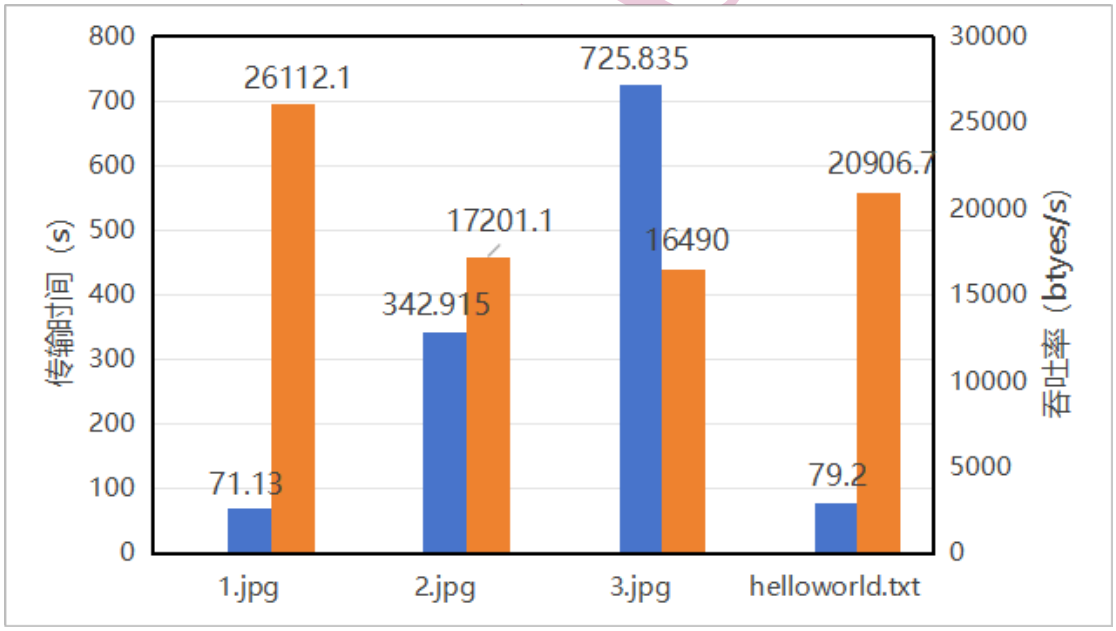
客户端收到 3 个冗余 ack1428 后，立刻执行快重传算法，重传 seq 为 1428 的数据包，而不是等待到超时重传。

执行快重传算法后，不启动慢启动算法，而执行快恢复算法。



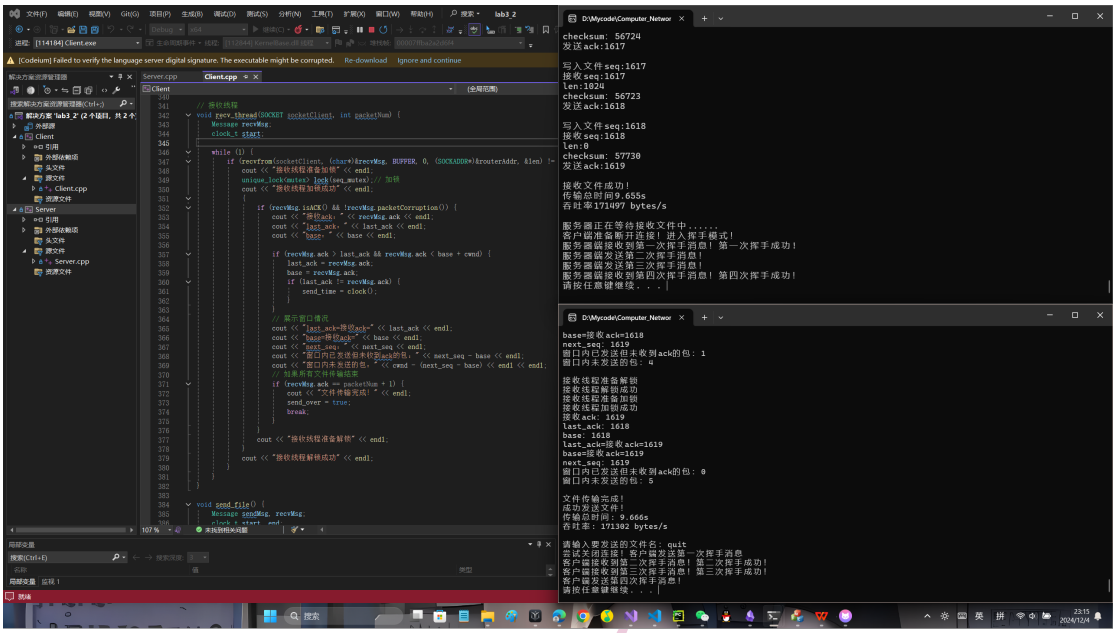
服务器端接收到 seq 为 1428 的数据包，返回 ack1448，表明 seq 为 1448 之前的数据包都已经成功接收，并准备好了下一个期望收到的数据包的 seq 为 1448。

设置 3% 的丢包率和 2ms 延迟，分析不同文件的传输时间和吞吐率。



文件名	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时间（s）	71.13	342.915	725.835	79.2
吞吐率（btyes/s）	26112.1	17201.1	16490	20906.7

断开连接：


checksum: 56724
发送ack:1617
写入文件seq:1617
接收seq:1617
len:1024
checksum: 56723
发送ack:1618
写入文件seq:1618
接收seq:1618
Len:0
checksum: 57730
发送ack:1619
接收文件成功!
传输总时间9.655s
吞吐率171497 bytes/s
服务器正在等待接收文件中.....
客户端准备断开连接! 进入挥手模式!
服务器端接收到第一次挥手消息! 第一次挥手成功!
服务器端发送第二次挥手消息!
服务器端发送第三次挥手消息!
服务器端接收到第四次挥手消息! 第四次挥手成功!
请按任意键继续. . .
base=接收ack=1618
next_seq: 1619
窗口内已发送但未收到ack的包: 1
窗口内未发送的包: 4
接收线程准备解锁
接收线程解锁成功
接收线程准备加锁
接收线程加锁成功
接收ack: 1619
last_ack: 1618
base: 1618
last_ack=接收ack=1619
base=接收ack=1619
next_seq: 1619
窗口内已发送但未收到ack的包: 0
窗口内未发送的包: 5
文件传输完成!
成功发送文件!
传输总时间: 9.666s
吞吐率: 171302 bytes/s
请输入要发送的文件名: quit
尝试关闭连接! 客户端发送第一次挥手消息
客户端接收到第二次挥手消息! 第二次挥手成功!
客户端接收到第三次挥手消息! 第三次挥手成功!
客户端发送第四次挥手消息!
请按任意键继续. . .