



南開大學
Nankai University

计算机学院
计算机网络实验报告

Lab3-3：基于 UDP 服务设计可靠传输
协议并编程实现

姓名：何禹姗

学号：2211421

专业：计算机科学与技术

2024 年 12 月 12 日

目录

1 实验要求	2
2 RENO 拥塞控制算法	2
2.1 慢启动阶段	2
2.2 拥塞避免阶段	2
2.3 快速恢复阶段	3
3 实验设计	3
3.1 数据报套接字 UDP	3
3.2 协议设计	3
3.3 差错检验	4
3.4 建立连接	5
3.5 断开连接	11
3.6 超时重传	17
3.7 数据传输	18
4 运行结果	30
4.1 连接建立	30
4.2 数据传输	31
4.3 断开连接	34

1 实验要求

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。

2 RENO 拥塞控制算法

在某段时间内，如果对网络中某一资源的需求超出了该资源所能提供的上限，则会导致网络性能的下降，这种情况被称为网络拥塞。例如当主机发送的数据过多过快时会造成网络中的路由器无法及时处理，因而引入了排队时延。

当出现网络拥塞时，为了降低网络的阻塞同时尽快传输数据而引入了拥塞控制。当发送完数据包后依据是否接收到 ACK 包来进行带宽的检测。如果接收到 ACK 包说明网络并未拥塞，可以提高发送速率；发生丢失事件则降低发送速率。

2.1 慢启动阶段

慢启动是建立连接后，采用的第一个调整发送速率的算法（模式）。在这个阶段，cwnd 通常被初始化为 1MSS，慢启动的目的就是尽快找到上限。

在慢启动阶段，发送方每接收到一个确认报文，就会将 cwnd 增加 1MSS 的大小：

初始 cwnd=1MSS，所以可发送一个 TCP 最大报文段，成功确认后， $cwnd = 2 \text{ MSS}$ ；

此时可发送两个 TCP 最大报文段，成功接收后， $cwnd = 4 \text{ MSS}$ ；

此时可发送四个 TCP 最大报文段，成功接收后， $cwnd = 8 \text{ MSS}$ 。

即在这个阶段，每发送完一次窗口内的所有数据包，窗口大小都翻倍（乘 2），在慢启动阶段窗口大小将以指数级别增长。

这个过程中何时结束这种指数增长，分为以下几种情况：

1. 若在慢启动阶段发生了数据传输超时，此时则将 ssthresh 的值设置为 $cwnd / 2$ ，将 cwnd 重新设置为 1MSS，重新开始慢启动过程，这个过程可以理解为试探上限。

2. 若 cwnd 的值增加到 $\geq ssthresh$ 时，此时即将达到上次窗口拥塞的大小，所以这个时候结束慢启动，改为拥塞避免模式。

3. 若发送方接收到了某个报文的三次重复 ACK（即触发了快速重传的条件），则进入到快速恢复阶段；同时， $ssthresh = cwnd / 2$ ，然后 $cwnd = ssthresh + 3MSS$ ；

2.2 拥塞避免阶段

刚进入拥塞避免模式时，cwnd 的大小近似的等于上次拥塞时的值的一半，即距离拥塞可能并不遥远。所以，拥塞避免是一个速率缓慢且线性增长的过程，在这个模式下，每经历一个 RTT，cwnd 的大小增加 1MSS。也就是说，窗口内的所有数据包全部发送一次，窗口大小加 1。假设 cwnd 包含 10 个报文的大小，则每接收到一个确认报文，cwnd 增加 $1/10 \text{ MSS}$ ，因此在收到对所有 10 个报文段的确认后，拥塞窗口的值将增加 1MSS。

何时结束拥塞避免这种线性增长，分为以下几种情况：

1. 在这个过程中，发生了数据传输超时，则表示网络拥塞，这时候，ssthresh 被修改为 $cwnd / 2$ ，然后 cwnd 被置为 1MSS，并进入慢启动阶段。

2. 若发送方接收到了某个报文的三次重复 ACK（即触发了快速重传的条件），此时也认为发生了拥塞，则进入到快速恢复阶段；同时， $ssthresh = cwnd / 2$ ，然后 $cwnd = ssthresh + 3MSS$ 。

2.3 快速恢复阶段

在快速恢复阶段，每接收到一个冗余的确认报文， $cwnd$ 就增加 $1MSS$ ，其余不变。

而当发生以下两种情况时，将退出快速恢复模式：

1. 在快速恢复过程中，计时器超时，那么 $ssthresh$ 被修改为 $cwnd / 2$ ，然后 $cwnd$ 被置为 $1MSS$ ，并进入慢启动阶段。

2. 若发送方接收到一条新的确认报文（不是重复的 ACK），则 $cwnd$ 被置为 $ssthresh$ ，然后进入到拥塞避免模式。

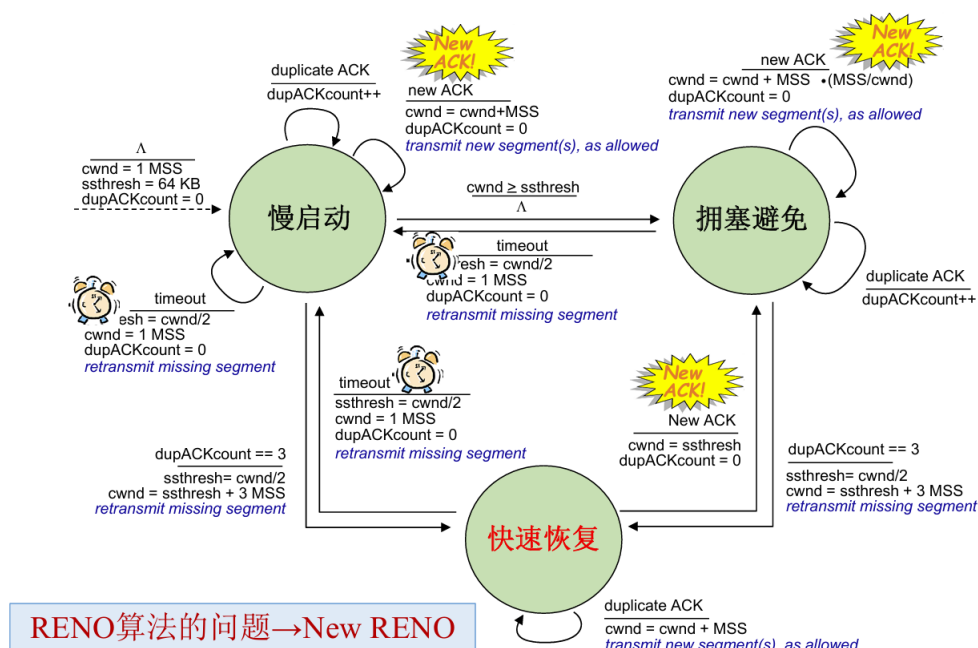


图 2.1: RENO 过程示意图

3 实验设计

3.1 数据报套接字 UDP

用户数据报协议是一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

3.2 协议设计

报文格式每个 UDP 报文分为 UDP 报头和 UDP 数据区两部分。报头由 4 个 16 位长（2 字节）字段组成，分别说明该报文的源端口、目的端口、报文长度和校验值。



图 3.2: 报文格式

在我的程序中，我的数据报由两部分组成，第一部分是 Packethead，即数据报头，里面包含 16 位的序列号，校验和，数据部分总长度，确认号和标志位。第二部分是数据报 Packet，里面包含数据报头和所携带的数据部分，长度上限为 2048 字节。

```

1  //数据报头
2  struct Packethead {
3      uint16_t seq;    // 序列号 16 位
4      uint16_t Check;  // 校验 16 位
5      uint16_t len;    // 数据部分总长度
6      uint16_t ack;    // 确认号
7      unsigned char flags; // 标志位
8
9      Packethead() : seq(0), Check(0), len(0), flags(0), ack(0) {}
10 };
11 //数据报
12 struct Packet {
13     Packethead head;
14     char data[2048]; // 数据部分
15
16     Packet() : head(), data() {}
17 };

```

3.3 差错检验

为了判断传输的数据是否正确，我们利用校验和进行差错检验。

UDP 校验和的计算方法是：

(1) 按每 16 位求和得出一个 32 位的数，如果这个 32 位的数，高 16 位不为 0，则高 16 位加低 16 位再得到一个 32 位的数；即如果结果超过 16 位，则将最高位的值加到最低位上，形成一个新的 16 位数。

(2) 重复上述步骤 (1) 直到高 16 位为 0, 将低 16 位取反, 得到校验和。如果最终结果为全 1 (即 0xFFFF), 则校验和为 0; 否则, 取反最终结果得到校验和。

将计算出的校验和值放入 UDP 头部的校验和字段中。

```
1 //差错检测
2 u_short packetcheck(u_short* packet, int packlength)
3 {
4     register u_long sum = 0;
5     int count = (packlength + 1) / 2; //两个字节的计算
6     u_short* buf = new u_short[packlength + 1];
7     memset(buf, 0, packlength + 1);
8     memcpy(buf, packet, packlength);
9     while (count--)
10     {
11         sum += *buf++;
12         if (sum & 0xFFFF0000)
13         {
14             sum &= 0xFFFF;
15             sum++;
16         }
17     }
18     return ~(sum & 0xFFFF);
19 }
```

3.4 建立连接

在程序中我基于 TCP 的三次握手协议建立连接。

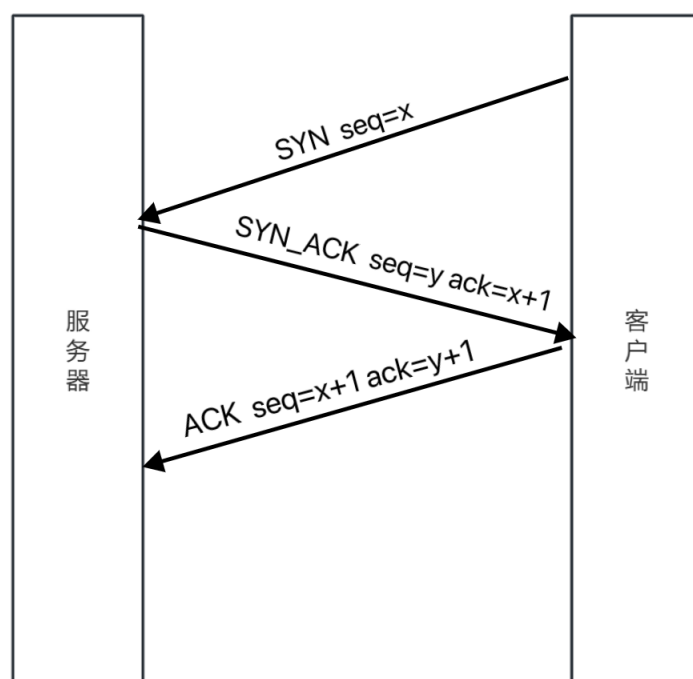


图 3.3: 三次握手示意图

第一次握手客户端向服务器发送数据报头标志位为 SYN 的数据包，表示想与服务器建立连接。(序列号 $seq=x$)

第二次握手服务器端向客户端发送数据报头标志位为 SYN_ACK 的数据包。(序列号 $seq=y$ ，确认号 $ack=x+1$)

标志位为 SYN（即第一次握手）的数据包发送至服务器端，由服务器端判断数据报序列号，标志位，校验和是否正确。如果正确，服务器端接收，且发送 SYN_ACK（即第二次握手）回客户端；如果不正确，服务器端不接收，客户端一段时间后未收到由服务器端发送的 SYN_ACK（即第二次握手），则重发标志位为 SYN（即第一次握手）的数据包。

同理，若数据包发送过程中被丢包，服务器端也无法顺利接收标志位为 SYN（即第一次握手）的数据包，此时客户端一段时间后未收到由服务器端发送的 SYN_ACK（即第二次握手），则重发标志位为 SYN（即第一次握手）的数据包。

第三次握手客户端向服务端发送标志位为 ACK 的数据包，与第一次握手同理，服务器端判断数据报序列号，标志位，校验和是否正确，如果正确服务器端则顺利接收数据包，此时握手完成，连接建立。(序列号 $seq=x+1$ ， $ack=y+1$)

客户端建立连接代码如下：

```

1  int clientHandshake(SOCKET s, sockaddr_in& clientAddr, int& sockLen) {
2  // 设置为非阻塞模式，避免卡在 recvfrom
3  u_long iMode = 1; // 0: 阻塞
4  ioctlsocket(s, FIONBIO, &iMode); // 非阻塞设置
5
6  Packet packet1;
  
```

```

7
8 // 发送 SYN 包
9 packet1.head.seq = 0;
10 packet1.head.flags = FLAG_SYN;
11 packet1.head.Check = packetcheck((u_short*)&packet1, sizeof(packet1));
12 int flag1 = 0;
13
14 //发送缓冲区
15 char* buffer1 = new char[sizeof(packet1)];
16 memcpy(buffer1, &packet1, sizeof(packet1));
17 flag1 = sendto(s, buffer1, sizeof(packet1), 0, (sockaddr*)&clientAddr,
18 sockLen);
19
20 if (flag1 == -1) { // 发送失败
21     std::cout << "[Client]: Failed to send SYN packet." << std::endl;
22     return 0;
23 }
24 clock_t start = clock(); //记录发送第一次握手时间
25 std::cout << "[Client]: SYN packet sent successfully." << std::endl;
26
27 // 等待 SYN_ACK 包
28 Packet packet2;
29
30 //缓冲区
31 char* buffer2 = new char[sizeof(packet2)];
32
33 bool synAckReceived = false;
34
35 while (recvfrom(s, buffer2, sizeof(packet2), 0, (sockaddr*)&clientAddr,
36 &sockLen)<=0) {
37     if (clock() - start > MAX_TIME) //超时, 重新传输第一次握手
38     {
39         std::cout << "[Client]:Timeout Retransmission,resending SYN
40 packet..." << std::endl;
41         sendto(s, buffer1, sizeof(packet1), 0, (sockaddr*)&clientAddr,
42 sockLen);
43         start = clock();
44     }
45 }
46 memcpy(&packet2, buffer2, sizeof(packet2));
47 u_short res = packetcheck((u_short*)&packet2, sizeof(packet2));
48 if (packet2.head.flags == FLAG_SYN_ACK && packet2.head.seq == 1 && res ==

```



```
49 0) {
50     std::cout << "[Client]: Received SYN_ACK packet with seq=" <<
51     packet2.head.seq << std::endl;
52     synAckReceived = true;
53 }
54 else {
55     std::cout << "[Client]: Failed to receive SYN_ACK packet." <<
56     std::endl;
57     return 0;
58 }
59 start = clock();
60
61 // 发送 ACK 包
62 Packet packet3;
63
64 packet3.head.seq = 2;
65 packet3.head.flags = FLAG_ACK;
66 packet3.head.Check = packetcheck((u_short*)&packet3, sizeof(packet3));
67 int flag3 = 0;
68
69 //发送缓冲区
70 char* buffer3 = new char[sizeof(packet3)];
71 memcpy(buffer3, &packet3, sizeof(packet3));
72
73 bool ackSentSuccessfully = false;
74
75 flag3 = sendto(s, buffer3, sizeof(packet3), 0, (sockaddr*)&clientAddr,
76 sockLen);
77
78 if (flag3 == -1) { // 发送失败
79     std::cout << "[Client]: Failed to send ACK packet." << std::endl;
80     return 0;
81 }
82
83 std::cout << "[Client]: ACK packet sent successfully." << std::endl;
84
85 iMode = 0; //0: 阻塞
86 ioctlsocket(s, FIONBIO, &iMode); //恢复成阻塞模式
87 return 1;
88 }
```

服务器端建立连接代码如下:

```

1  int serverHandshake(SOCKET s, sockaddr_in& ServerAddr, int& sockLen) {
2  //设置为非阻塞模式, 避免卡在 recvfrom
3  u_long iMode = 1; //0: 阻塞
4  ioctlsocket(s, FIONBIO, &iMode); //非阻塞设置
5
6  Packet packet1; //储存接收到的数据包
7  int flag1 = 0;
8
9  bool synReceived = false; //标记是否收到 SYN 包
10
11 char* buffer1 = new char[sizeof(packet1)];
12 // 等待 SYN 包
13 while (1) {
14
15     flag1 = recvfrom(s, buffer1, sizeof(packet1), 0,
16                     (sockaddr*)&ServerAddr, &sockLen);
17     //没收到就一直循环
18     if (flag1 <= 0)
19     {
20         //cout << "error" << endl;
21         continue;
22     }
23     //如果接收成功
24     memcpy(&(packet1), buffer1, sizeof(packet1.head));
25     u_short res = packetcheck((u_short*)&packet1, sizeof(packet1));
26     //cout << "success" << endl;
27     //检查接收到的是否为 SYN 包
28     //标志位为 FLAG_SYN 且序列号为 1, 且数据包正确
29     if (packet1.head.flags == FLAG_SYN && packet1.head.seq == 0 && res ==
30         0) {
31         std::cout << "[Server]: Received SYN packet with seq=" <<
32         packet1.head.seq << std::endl;
33         synReceived = true;
34         break;
35     }
36 }
37
38 //如果未收到 SYN 包, 输出一条消息表示失败
39 if (!synReceived) {
40     std::cout << "[Server]: Failed to receive SYN packet." << std::endl;
41     //return 0;

```

```

42 }
43
44 // 发送 SYN_ACK 包
45 Packet packet2;
46 packet2.head.seq = 1;
47 packet2.head.flags = FLAG_SYN_ACK;
48 packet2.head.Check = packetcheck((u_short*)&packet2, sizeof(packet2));
49
50 char* buffer2 = new char[sizeof(packet2)];
51 memcpy(buffer2, &packet2, sizeof(packet2));
52
53 int flag2 = sendto(s, buffer2, sizeof(packet2), 0,
54 (sockaddr*)&ServerAddr, sockLen);
55 if (flag2 == -1) { // 发送失败
56     std::cout << "[Server]: Failed to send SYN_ACK packet." << std::endl;
57     return 0;
58 }
59 clock_t start = clock(); // 记录第二次握手发送时间
60 std::cout << "[Server]: SYN_ACK packet sent successfully." << std::endl;
61
62
63 bool ackReceived = false; // 标记是否收到 ACK 包
64
65 // 等待 ACK 包
66 Packet packet3;
67 char* buffer3 = new char[sizeof(packet3)];
68 int flag3 = 0;
69
70 while (recvfrom(s, buffer3, sizeof(packet3), 0, (sockaddr*)&ServerAddr,
71 &sockLen) <= 0) {
72     if (clock() - start > MAX_TIME) // 超时, 重新传输第一次握手
73     {
74         return 1;
75         /*std::cout << "[Server]: Timeout Retransmission, resending SYN_ACK
76 packet..." << std::endl;
77         sendto(s, buffer2, sizeof(packet2), 0, (sockaddr*)&ServerAddr,
78 sockLen);
79         start = clock();*/
80     }
81 }
82 // 如果接收成功
83 memcpy(&(packet3), buffer3, sizeof(packet3.head));

```

```

84  u_short res = packetcheck((u_short*)&packet3, sizeof(packet3));
85
86  if (packet3.head.flags == FLAG_ACK && packet3.head.seq == 2 && res == 0) {
87      std::cout << "[Server]: Received ACK packet with seq=" <<
88      packet3.head.seq << std::endl;
89  }
90
91  iMode = 0; //0: 阻塞
92  ioctlsocket(s, FIONBIO, &iMode); //恢复成阻塞模式
93  return 1;
94  }

```

3.5 断开连接

在程序中我基于 TCP 的四次挥手协议断开连接。

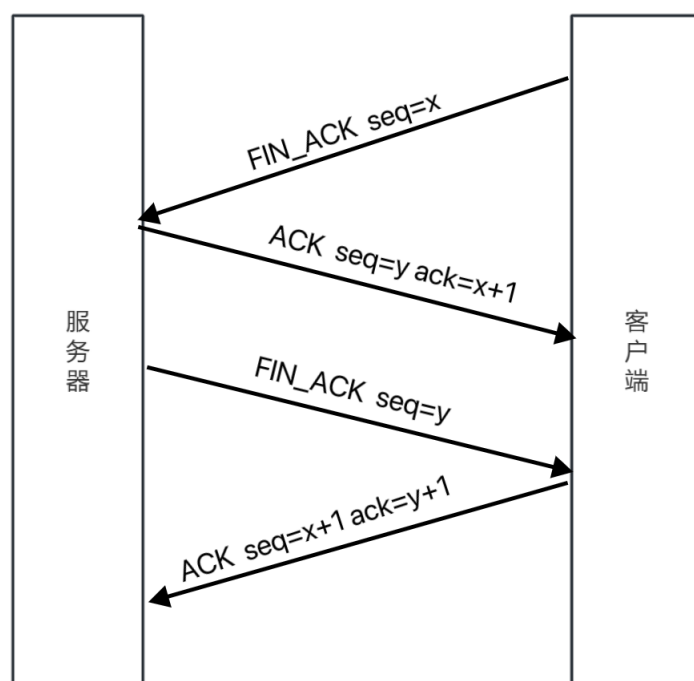


图 3.4: 四次挥手示意图

第一次挥手客户端向服务器发送数据报头标志位为 FIN_ACK 的数据包，表示想与服务器断开连接。（序列号 seq=x）

第二次挥手服务器端向客户端发送数据报头标志位为 ACK 的数据包。（序列号 seq=y，确认号 ack=x+1）

标志位为 FIN_ACK（即第一次挥手）的数据包发送至服务器端，由服务器端判断数据报序列号，标志位，校验和是否正确。如果正确，服务器端接收，且发送 ACK（即第二次挥手）回客户端；如果不正确，服务器端不接收，客户端一段时间后未收到由服务器端发送的 ACK（即第二次挥手），则重

发标志位为 FIN_ACK（即第一次挥手）的数据包。

同理，若数据包发送过程中被丢包，服务器端也无法顺利接收标志位为 FIN_ACK（即第一次挥手）的数据包，此时客户端一段时间后未收到由服务器端发送的 ACK（即第二次挥手），则重发标志位为 FIN_ACK（即第一次挥手）的数据包。

第三、四次挥手第三、四次挥手与一、二次相同，只不过改为服务器端发送数据报头标志位为 FIN_ACK 的数据包，客户端发送数据报头标志位为 ACK 的数据包。（第三次挥手序列号 seq=y；第四次挥手序列号 seq=x+1，确认号 ack=y+1）

客户端断开连接代码如下：

```

1  int clientCloseConnection(SOCKET s, sockaddr_in& ClientAddr, int&
2  sockLen) {
3  // 设置为非阻塞模式，避免卡在 recvfrom
4  u_long iMode = 1; // 0: 阻塞
5  ioctlsocket(s, FIONBIO, &iMode); // 非阻塞设置
6
7  Packet packet1;
8  int flag1 = 0;
9
10 // 发送 FIN_ACK 包
11 packet1.head.seq = 0;
12 packet1.head.flags = FLAG_FIN_ACK;
13 packet1.head.Check = packetcheck((u_short*)&packet1, sizeof(packet1));
14 char* buffer1 = new char[sizeof(packet1)];
15 memcpy(buffer1, &packet1, sizeof(packet1));
16
17 flag1 = sendto(s, buffer1, sizeof(packet1), 0, (sockaddr*)&ClientAddr,
18 sockLen);
19 if (flag1 == -1) {
20     std::cout << "[Client]: Failed to send FIN_ACK packet." << std::endl;
21     return 0;
22 }
23 clock_t start = clock();
24 std::cout << "[Client]: FIN_ACK packet sent successfully." << std::endl;
25
26 // 等待服务器发送的 ACK 包
27 Packet packet2;
28 char* buffer2 = new char[sizeof(packet2)];
29
30 bool ackReceived = false;
31
32 while (recvfrom(s, buffer2, sizeof(packet2), 0, (sockaddr*)&ClientAddr,
33 &sockLen)<=0) {

```

```
34     if (clock() - start > MAX_TIME)//超时, 重新传输第一次握手
35     {
36         std::cout << "[Client]:Timeout Retransmission,resending FIN_ACK
37         packet..." << std::endl;
38         sendto(s, buffer1, sizeof(packet1), 0, (sockaddr*)&ClientAddr,
39         sockLen);
40         start = clock();
41     }
42 }
43 memcpy(&packet2, buffer2, sizeof(packet2));
44 u_short res = packetcheck((u_short*)&packet2, sizeof(packet2));
45
46 if (packet2.head.flags == FLAG_ACK && packet2.head.seq == 1 && res == 0) {
47     std::cout << "[Client]: Received ACK packet with seq=" <<
48     packet2.head.seq << std::endl;
49     ackReceived = true;
50 }
51 else{
52     std::cout << "[Client]: Failed to receive ACK packet." << std::endl;
53     //return 0;
54 }
55
56 // 等待服务器发送的 FIN_ACK 包
57 bool finackReceived = false;
58 Packet packet3;
59 char* buffer3 = new char[sizeof(packet3)];
60
61 while(1) {
62     u_short res = packetcheck((u_short*)&packet3, sizeof(packet3));
63
64     if (recvfrom(s, buffer3, sizeof(packet3), 0, (sockaddr*)&ClientAddr,
65     &sockLen)<=0) {
66         //cout << " 未收到 FIN_ACK 包 " << endl;
67     }
68     memcpy(&packet3, buffer3, sizeof(packet3));
69     if (packet3.head.flags == FLAG_FIN_ACK && packet3.head.seq == 2 &&
70     res == 0) {
71         std::cout << "[Client]: Received FIN_ACK packet with seq=" <<
72         packet3.head.seq << std::endl;
73         finackReceived = true;
74         break;
75     }
```

```
76
77 }
78 start = clock();
79
80 if (!finackReceived) {
81     std::cout << "[Client]: Failed to receive FIN_ACK packet." <<
82     std::endl;
83     return 0;
84 }
85
86 // 发送 ACK 包
87 Packet packet4;
88 packet4.head.seq = 3;
89 packet4.head.flags = FLAG_ACK;
90 packet4.head.Check = packetcheck((u_short*)&packet4, sizeof(packet4));
91 char* buffer4 = new char[sizeof(packet4)];
92 memcpy(buffer4, &packet4, sizeof(packet4));
93 int flag4 = 0;
94
95 bool ackSentSuccessfully = false;
96
97 while (!ackSentSuccessfully) {
98     flag4 = sendto(s, buffer4, sizeof(packet4), 0,
99     (sockaddr*)&ClientAddr, sockLen);
100     if (flag4 != -1) { // 发送成功
101         std::cout << "[Client]: ACK packet sent successfully." <<
102         std::endl;
103         ackSentSuccessfully = true;
104     }
105     else {
106         std::cout << "[Client]: Failed to send ACK packet, retrying..."
107         << std::endl;
108         return 0;
109     }
110 }
111
112 iMode = 0; // 0: 阻塞
113 ioctlsocket(s, FIONBIO, &iMode); // 恢复成阻塞模式
114 return 1;
115 }
```

服务器端断开连接代码如下：

```

1  int serverCloseConnection(SOCKET s, sockaddr_in& ServerAddr, int&
2  sockLen) {
3  // 设置为非阻塞模式, 避免卡在 recvfrom
4  u_long iMode = 1; // 0: 阻塞
5  ioctlsocket(s, FIONBIO, &iMode); // 非阻塞设置
6
7  Packet packet1;
8  char* buffer1 = new char[sizeof(packet1)];
9  int flag = 0;
10
11 // 等待客户端发送的 FIN_ACK 包
12 bool finackReceived = false;
13
14 while(1) {
15     if (recvfrom(s, buffer1, sizeof(packet1), 0, (sockaddr*)&ServerAddr,
16     &sockLen) > 0) {
17         memcpy(&(packet1), buffer1, sizeof(packet1));
18         u_short res = packetcheck((u_short*)&packet1, sizeof(packet1));
19         if (packet1.head.flags == FLAG_FIN_ACK && packet1.head.seq == 0
20         && res == 0) {
21             std::cout << "[Server]: Received FIN_ACK packet with seq=" <<
22             packet1.head.seq << std::endl;
23             finackReceived = true;
24             break;
25         }
26     }
27     else {
28         continue;
29     }
30 }
31
32 if (!finackReceived) {
33     std::cout << "[Server]: Failed to receive FIN_ACK packet." <<
34     std::endl;
35     return 0;
36 }
37
38 // 发送 ACK 包
39 Packet packet2;
40 packet2.head.seq = 1;
41 packet2.head.flags = FLAG_ACK;

```



```
42 packet2.head.Check = packetcheck((u_short*)&packet2, sizeof(packet2));
43 char* buffer2 = new char[sizeof(packet2)];
44 memcpy(buffer2, &packet2, sizeof(packet2));
45
46 flag = sendto(s, buffer2, sizeof(packet2), 0, (sockaddr*)&ServerAddr,
47 sockLen);
48 if (flag != -1) { // 发送成功
49     std::cout << "[Server]: ACK packet sent successfully." << std::endl;
50 }
51 else {
52     std::cout << "[Server]: Failed to send ACK packet." << std::endl;
53 }
54
55 // 发送 FIN_ACK 包
56 Packet packet3;
57 packet3.head.seq = 2;
58 packet3.head.flags = FLAG_FIN_ACK;
59 packet3.head.Check = packetcheck((u_short*)&packet3, sizeof(packet3));
60 char* buffer3 = new char[sizeof(packet3)];
61 memcpy(buffer3, &packet3, sizeof(packet3));
62
63 bool finackSentSuccessfully = false;
64
65 while (!finackSentSuccessfully) {
66     flag = sendto(s, buffer3, sizeof(packet3), 0, (sockaddr*)&ServerAddr,
67 sockLen);
68     if (flag != -1) { // 发送成功
69         std::cout << "[Server]: FIN_ACK packet sent successfully." <<
70 std::endl;
71         finackSentSuccessfully = true;
72     }
73     else {
74         std::cout << "[Server]: Failed to send FIN_ACK packet,
75 retrying..." << std::endl;
76         std::this_thread::sleep_for(std::chrono::seconds(1));
77     }
78 }
79 clock_t start = clock();
80
81 if (flag == -1) { // 发送失败
82     std::cout << "[Server]: Failed to send FIN_ACK packet." << std::endl;
83     return 0;
```

```

84 }
85
86 // 等待客户端发送的 ACK 包
87 Packet packet4;
88 char* buffer4 = new char[sizeof(packet4)];
89 int flag4 = 0;
90
91 while (recvfrom(s, buffer4, sizeof(packet4), 0, (sockaddr*)&ServerAddr,
92 &sockLen) <= 0) {
93     if (clock() - start > MAX_TIME) // 超时
94     {
95         return 1;
96         /*std::cout << "[Server]:Timeout Retransmission,resending ACK
97 packet..." << std::endl;
98 sendto(s, buffer3, sizeof(packet3), 0, (sockaddr*)&ServerAddr,
99 sockLen);
100 start = clock();*/
101     }
102 }
103 //如果接收成功
104 memcpy(&(packet4), buffer4, sizeof(packet4.head));
105 u_short res = packetcheck((u_short*)&packet4, sizeof(packet4));
106
107 if (packet4.head.flags == FLAG_ACK && packet4.head.seq == 3 && res == 0) {
108     std::cout << "[Server]: Received ACK packet with seq=" <<
109     packet4.head.seq << std::endl;
110 }
111
112 iMode = 0; // 0: 阻塞
113 ioctlsocket(s, FIONBIO, &iMode); // 恢复成阻塞模式
114 return 1;
115 }

```

3.6 超时重传

本次实验基于滑动窗口设计了超时重传协议，每个数据都有一个唯一的序列号，用于标识数据包的顺序，接收方发送的确认信息（ACK）中包含确认号，表示已成功接收数据包的序列号，发送方为每个未确认的数据包设置一个超时计时器，如果在超时时间内没有收到确认消息，发送方会从该数据包开始将窗口内所有数据包重新发送。

```

1 if (clock() - start > MAX_TIME)
2 {

```

```

3  //超时进入慢启动状态
4  window.ssthresh = window.size / 2;
5  window.size = 1;
6  window.end = window.start + window.size - 1;
7  if (window.end >= packagenum)
8  {
9      window.end = packagenum;
10 }
11 window.state = SLOW_START;
12 window.ackcount = 0;
13 cout << endl;
14 cout << " 超时重传, 进入慢启动状态, 当前 ssthresh 值为: " << window.ssthresh << endl;
15 //将窗口内的所有数据重传
16 for (int j = window.start; j <= window.end; j++)
17 {
18     len = (j == packagenum ? (messagelen - ((packagenum - 1) * MAXSIZE)) : MAXSIZE);
19     Packet packet2;
20     char* buffer2 = new char[len + sizeof(packet2)];
21     packet2.head.len = len;
22     packet2.head.seq = j; //序列号
23     packet2.head.Check = 0;
24     u_short check = packetcheck((u_short*)&packet2, sizeof(packet2)); //计算校验和
25     packet2.head.Check = check;
26     packet2.head.flags = 0;
27     packet2.head.ack = 0;
28     memcpy(buffer2, &packet2, sizeof(packet2));
29     char* mes = message + (j-1) * MAXSIZE;
30     memcpy(buffer2 + sizeof(packet2), mes, len); //将数据复制到缓冲区的后部分
31     sendto(socketClient, buffer2, len + sizeof(packet2), 0, (sockaddr*)&servAddr, servAddrlen)
32
33     std::lock_guard<std::mutex> guardsend(coutMutex);
34     cout << " 发送文件大小为 " << len << " bytes!" << " Flag:" << int(packet2.head.flags) << "
35     start = clock();
36 }
37 window.maxseq = window.end;
38 seqnum = window.end + 1;
39 }

```

3.7 数据传输

数据传输中基于 RENO 算法实现拥塞控制。

1. 传输开始前, 我将阈值 ssthresh 设为 14, 窗口起始大小为 1, 开始传输后, 进入慢启动阶段,

窗口大小指数增长；

2. 当窗口大小 = 阈值 ssthresh 时，进入拥塞避免阶段，窗口大小线性增长；
3. 当收到三个重复的 ack 时，触发快速重传，进入快速恢复阶段，阈值 ssthresh = 窗口大小 / 2，窗口大小 = ssthresh + 3，同时将窗口内的数据全部重传；
4. 在此之后如果再收到重复的 ack，窗口大小加 1，如果收到了正确的 ack，将窗口大小变为 ssthresh 大小，恢复拥塞避免阶段；
5. 在数据传输过程中，如果数据传输超时，则进入慢启动阶段，将阈值 ssthresh = 窗口大小 / 2，窗口大小设为 1，同时将窗口内的所有数据重传。

对于发送方 client：为了实现数据的边收边传，发送端的发送数据和接收 ack 仍然使用双线程实现。

1. 维护一个发送窗口结构体，记录此时窗口起始 start 和窗口结束 end，设定窗口大小 size。

```

1  struct sendwindow {
2      int size;
3      int start;
4      int end;
5
6      sendwindow():size(20),start(-1),end(0){}
7  };

```

2. 发送窗口内的所有数据包，数据包序号递增，发送完第一个包就开始计时。
3. 当收到从接收端发回的 ACK 时，检查判断接收到的数据包的正确性，如果正确且 $\text{int}(\text{packet2.head.seq}) \geq \text{window.start} \% 256$ 即接收回的 ACK 序列号大于窗口起始号，则说明现在窗口内的数据包存在被确认的，此时窗口向后移，移到未被确认的第一个数据包，同时根据不同阶段改变窗口大小，将现在窗口内所有未发送的数据包发送，重新开始计时；如果接收到重复的 ACK，说明有数据包未成功发送，此时开始进行计数，当收到三个重复的 ack 时，触发快速重传，将窗口大小改变，同时进入快速恢复阶段，将窗口内的所有数据包重发，直到接收到正确的 ack，恢复拥塞避免阶段。
4. 如果发送端在设置的超时时间内未收到 ACK 确认数据包，则将窗口内的所有数据包重传，同时改变窗口大小，进入慢启动阶段，并重新开始计时。
5. 为了提高效率和资源利用率设置了多线程，即将发送端 ACK 的接收设置单独线程，不影响主线程的发送。因为维护了全局变量窗口的结构体，所以同时还需在 ACK 接收线程和主线程加了 `std::lock_guard<std::mutex>` 互斥锁，并在对象析构时自动释放该锁，可以确保在多线程环境下对共享资源的安全访问。

发送端代码 ACK 接收线程如下：

```

1  void handleAckReception(SOCKET& socketClient, SOCKADDR_IN& servAddr, int&
2  servAddrlen, int packagenum)
3  {
4      Packet packet2;
5      char* buffer2 = new char[sizeof(packet2)];
6      int num = 0;
7      while (window.start <= packagenum)

```

```

8  {
9      if (recvfrom(socketClient, buffer2, sizeof(packet2), 0,
10         (sockaddr*)&servAddr, &servAddrlen) > 0)
11      {
12          memcpy(&packet2, buffer2, sizeof(packet2));
13          u_short check2 = packetcheck((u_short*)&packet2, sizeof(packet2));
14
15          std::lock_guard<std::mutex> lock(windowMutex); // 加锁
16          if (int(check2) != 0)
17          {
18              //window.end = window.start + 1;
19
20              //std::lock_guard<std::mutex> guarderror(coutMutex);
21              cout << " 接收到错误的 ACK" << endl;
22              continue;
23          }
24          else
25          {
26              if (int(packet2.head.seq) >= window.start)
27              {
28                  //如果为慢启动状态, cwnd 翻倍
29                  if (window.state == SLOW_START)
30                  {
31                      int nowsize = window.size;
32                      num = 0;
33                      window.start = int(packet2.head.seq) + 1;
34                      window.size += 1;
35                      window.end = window.start + window.size - 1;
36                      if (window.end >= packagenum)
37                      {
38                          window.end = packagenum;
39                      }
40                      window.curacq = packet2.head.seq;
41                      window.ackcount = 0;
42                      std::lock_guard<std::mutex> guardack(coutMutex);
43                      cout << endl;
44                      cout << " 慢启动阶段, 窗口大小为: " << window.size << " 窗
45                         口大小变化为: " << window.size - nowsize << endl;
46                      cout << " 接收到 ACK, 确认数据包发送成功  Flag:" <<
47                         int(packet2.head.flags) << " SEQ:" <<
48                         int(packet2.head.seq) << " ACK:" <<
49                         int(packet2.head.ack) << " CHECK:" <<

```

```

50         int(packet2.head.Check) << endl;
51         if (window.size >= window.ssthresh)
52         {
53             //进入拥塞避免阶段
54             window.state = CON_AVOID;
55             cout << " 慢启动阶段进入拥塞避免阶段" << endl;
56         }
57     }
58     //拥塞避免阶段收到 ACK 线性增长
59     if (window.state == CON_AVOID)
60     {
61         int nowsize = window.size;
62         num += 1;
63         window.start = int(packet2.head.seq) + 1;
64         if (num == window.size)
65         {
66             //cout << " 拥塞避免增加窗口大小" << endl;
67             window.size += 1;
68             num = 0;
69         }
70         window.end = window.start + window.size - 1;
71         if (window.end >= packagenum)
72         {
73             window.end = packagenum;
74         }
75         window.curacq = packet2.head.seq;
76         window.ackcount = 0;
77         std::lock_guard<std::mutex> guardack(coutMutex);
78         cout << endl;
79         cout << " 拥塞避免阶段, 窗口大小为: " << window.size << "
80             窗口大小变化为: " << window.size - nowsize << endl;
81         cout << " 接收到 ACK, 确认数据包发送成功  Flag:" <<
82         int(packet2.head.flags) << " SEQ:" <<
83         int(packet2.head.seq) << " ACK:" <<
84         int(packet2.head.ack) << " CHECK:" <<
85         int(packet2.head.Check) << endl;
86     }
87     //快速恢复阶段
88     if (window.state == FAST_RECOVERY)
89     {
90         int nowsize = window.size;
91         num = 0;

```

```

92         window.state = CON_AVOID;
93         window.start = int(packet2.head.seq) + 1;
94         window.size = window.ssthresh;
95         window.end = window.start + window.size - 1;
96         if (window.end >= packagenum)
97         {
98             window.end = packagenum;
99         }
100         window.curacq = packet2.head.seq;
101         window.ackcount = 0;
102         std::lock_guard<std::mutex> guardack(coutMutex);
103         cout << endl;
104         cout << " 当前为快速恢复阶段，窗口大小为: " << window.size
105         << " 窗口大小变化为: " << window.size - nowsize << endl;
106         cout << " 接收到 ACK，确认数据包发送成功  Flag:" <<
107         int(packet2.head.flags) << " SEQ:" <<
108         int(packet2.head.seq) << " ACK:" <<
109         int(packet2.head.ack) << " CHECK:" <<
110         int(packet2.head.Check) << endl;
111         cout << " 快速恢复阶段进入拥塞避免阶段" << endl;
112     }
113     //std::lock_guard<std::mutex> guardack(coutMutex);
114     //cout << " 接收到 ACK，确认数据包发送成功  Flag:" <<
115     int(packet2.head.flags) << " SEQ:" <<
116     int(packet2.head.seq) << " ACK:" << int(packet2.head.ack)
117     << " CHECK:" << int(packet2.head.Check) << " 窗口起始: "
118     << window.start << " 窗口结束: " << window.end << " 窗口大
119     小:" << window.size << endl;
120
121 }
122 else
123 {
124     if (packet2.head.seq == window.curacq)
125     {
126         int nowsize = window.size;
127         window.ackcount += 1;
128         if (window.ackcount > 3)
129         {
130             window.size += 1;
131         }
132         std::lock_guard<std::mutex> guardack(coutMutex);
133         cout << " 收到重复的 ack，序列号为: "<< packet2.head.seq

```

```

134         << " 此时 ackcount: " << window.ackcount << " 窗口大小: "
135         << window.size << " 窗口大小变化为: " << window.size -
136         nowsize << endl;
137     }
138     //一旦受到重复的 3 个 ACK 就进入快速恢复状态
139     if (window.ackcount == 3)
140     {
141         int nowsize = window.size;
142         window.state = FAST_RECOVERY;
143         window.ssthresh = window.size / 2;
144         window.size = window.ssthresh + 3;
145         window.end = window.start + window.size - 1;
146         if (window.end >= packagenum)
147         {
148             window.end = packagenum;
149         }
150         window.resend = 1;
151         std::lock_guard<std::mutex> guardack(coutMutex);
152         cout << endl;
153         cout << " 收到 3 个重复 ACK, 序列号为: " << packet2.head.seq
154         << " 重传数据包, 进入快速恢复阶段, 窗口大小为: " <<
155         window.size << " 窗口大小变化为: " << window.size -
156         nowsize << endl;
157     }
158     continue;
159 }
160 }
161 //锁自动释放
162 }
163 }
164 delete[] buffer2;
165 }

```

发送端代码发送数据包线程如下:

```

1 void send(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen,
2 char* message, int messagelen)
3 {
4     u_long mode = 1;
5     ioctlsocket(socketClient, FIONBIO, &mode);
6     int seqnum = 1; //记录下一个要发送的数据包序列号
7     int packagenum = (messagelen / MAXSIZE) + (messagelen % MAXSIZE != 0);

```



```

8  cout << packagenum << endl;
9  int len = 0;
10 //sendwindow window;
11 window.start = 1;
12 window.size = 1;
13 window.end = 1;
14 window.state = SLOW_START;
15 window.ssthresh = 14;
16 window.maxseq = 0;
17 window.curacq = 0;
18 window.ackcount = 0;
19 window.resend = 0;
20 clock_t start = clock();
21
22 // 启动 ACK 接收线程
23 std::thread ackThread(handleAckReception, std::ref(socketClient),
24 std::ref(servAddr), std::ref(servAddrLen), std::ref(packagenum));
25 //ackThread.detach();
26
27 //std::lock_guard<std::mutex> lock(windowMutex);
28 while (seqnum <= packagenum+1)
29 {
30     if (window.maxseq < window.end)
31     {
32         for (int i = window.maxseq + 1 ; i <= window.end;i++)
33         {
34             len = (i == packagenum ? (messagelen - ((packagenum - 1) *
35             MAXSIZE)) : MAXSIZE);
36             Packet packet1;
37             char* buffer1 = new char[len + sizeof(packet1)];
38             packet1.head.len = len;
39             packet1.head.seq = seqnum; //序列号
40             packet1.head.Check = 0;
41             u_short check = packetcheck((u_short*)&packet1,
42             sizeof(packet1)); //计算校验和
43             packet1.head.Check = check;
44             packet1.head.flags = 0;
45             packet1.head.ack = 0;
46             memcpy(buffer1, &packet1, sizeof(packet1));
47             char* mes = message + (i - 1) * MAXSIZE;
48             memcpy(buffer1 + sizeof(packet1), mes, len); //将数据复制到缓冲区
49             的后部分

```

```

50         sendto(socketClient, buffer1, len + sizeof(packet1), 0,
51             (sockaddr*)&servAddr, servAddrlen); //发送
52         std::lock_guard<std::mutex> guardsend(coutMutex);
53         cout << " 发送文件大小为 " << len << " bytes!" << " Flag:" <<
54             int(packet1.head.flags) << " SEQ:" << int(packet1.head.seq)
55             << " ACK:" << int(packet1.head.ack) << " CHECK:" <<
56             int(packet1.head.Check) << endl;
57         start = clock();
58         seqnum = i+1;
59         window.maxseq = i;
60         if (window.resend == 1)
61         {
62             break;
63         }
64     }
65 }
66 if (window.resend == 1)
67 {
68     for (int j = window.start; j <= window.end; j++)
69     {
70         len = (j == packagenum ? (messagelen - ((packagenum - 1) *
71             MAXSIZE)) : MAXSIZE);
72         Packet packet2;
73         char* buffer2 = new char[len + sizeof(packet2)];
74         packet2.head.len = len;
75         packet2.head.seq = j; //序列号
76         packet2.head.Check = 0;
77         u_short check = packetcheck((u_short*)&packet2,
78             sizeof(packet2)); //计算校验和
79         packet2.head.Check = check;
80         packet2.head.flags = 0;
81         packet2.head.ack = 0;
82         memcpy(buffer2, &packet2, sizeof(packet2));
83         char* mes = message + (j - 1) * MAXSIZE;
84         memcpy(buffer2 + sizeof(packet2), mes, len); //将数据复制到缓冲区
85         的后部分
86         sendto(socketClient, buffer2, len + sizeof(packet2), 0,
87             (sockaddr*)&servAddr, servAddrlen); //发送
88
89         std::lock_guard<std::mutex> guardsend(coutMutex);
90         cout << " 发送文件大小为 " << len << " bytes!" << " Flag:" <<
91             int(packet2.head.flags) << " SEQ:" << int(packet2.head.seq)

```

```

92         << " ACK:" << int(packet2.head.ack) << " CHECK:" <<
93         int(packet2.head.Check) << endl;
94         start = clock();
95     }
96     window.resend = 0;
97     window.maxseq = window.end;
98     seqnum = window.end + 1;
99 }
100 if (clock() - start > MAX_TIME)
101 {
102     //超时进入慢启动状态
103     window.ssthresh = window.size / 2;
104     window.size = 1;
105     window.end = window.start + window.size - 1;
106     if (window.end >= packagenum)
107     {
108         window.end = packagenum;
109     }
110     window.state = SLOW_START;
111     window.ackcount = 0;
112     cout << endl;
113     cout << " 超时重传, 进入慢启动状态, 当前 ssthresh 值为: " <<
114     window.ssthresh << endl;
115     for (int j = window.start; j <= window.end; j++)
116     {
117         len = (j == packagenum ? (messagelen - ((packagenum - 1) *
118         MAXSIZE)) : MAXSIZE);
119         Packet packet2;
120         char* buffer2 = new char[len + sizeof(packet2)];
121         packet2.head.len = len;
122         packet2.head.seq = j; //序列号
123         packet2.head.Check = 0;
124         u_short check = packetcheck((u_short*)&packet2,
125         9sizeof(packet2)); //计算校验和
126         packet2.head.Check = check;
127         packet2.head.flags = 0;
128         packet2.head.ack = 0;
129         memcpy(buffer2, &packet2, sizeof(packet2));
130         char* mes = message + (j-1) * MAXSIZE;
131         memcpy(buffer2 + sizeof(packet2), mes, len); //将数据复制到缓冲区的后部分
132         sendto(socketClient, buffer2, len + sizeof(packet2), 0,

```

```

134         (sockaddr*)&servAddr, servAddrlen); //发送
135
136         std::lock_guard<std::mutex> guardsend(coutMutex);
137         cout << " 发送文件大小为 " << len << " bytes!" << " Flag:" <<
138         int(packet2.head.flags) << " SEQ:" << int(packet2.head.seq)
139         << " ACK:" << int(packet2.head.ack) << " CHECK:" <<
140         int(packet2.head.Check) << endl;
141         start = clock();
142     }
143     window.maxseq = window.end;
144     seqnum = window.end + 1;
145 }
146 if (window.start > packagenum)
147 {
148     break;
149 }
150 else
151 {
152     continue;
153 }
154 }
155 ackThread.join();
156
157 // 等待所有 ACK
158 /*std::unique_lock<std::mutex> lock(ackMutex);
159 std::this_thread::sleep_for(std::chrono::milliseconds(100));*/
160
161 //发送结束信息
162 Packet packet5;
163 char* Buffer5 = new char[sizeof(packet5)];
164 packet5.head.flags = OVER; //结束信息
165 packet5.head.Check = 0;
166 u_short temp = packetcheck((u_short*)&packet5, sizeof(packet5));
167 packet5.head.Check = temp;
168
169 memcpy(Buffer5, &packet5, sizeof(packet5));
170 sendto(socketClient, Buffer5, sizeof(packet5), 0, (sockaddr*)&servAddr,
171 servAddrlen);
172 //cout << " 发送文件完毕! " << endl;
173 start = clock();
174
175 mode = 0;

```

```

176  ioctlsocket(socketClient, FIONBIO, &mode); //改回阻塞模式
177
178  return;
179  }

```

对于接收方 server: 1. 声明一个变量 seq 记录希望得到的数据包序列号, 对于接收到的数据包进行判断是否正确, 如果收到的数据包序列号正确且校验和正确, 则将此数据包写入文件缓冲区, 发送 ack=seq+1 回发送端, 同时 seq+1。

2. 如果收到的数据包校验和错误或序列号错误, 那么丢弃此数据包且不发送 ACK, 此时发送端收不到对应的 ACK 便会重传。

3. 直到收到发送端发送的 over 数据包表示文件传输结束, 返回 ACK 确认, 接收文件完毕。

4. 由于路由器有发送文件个数的限制, 影响数据传输速率, 我在服务器端设置了模拟丢包。

接收端代码如下:

```

1  int RecvMessage(SOCKET& sockServ, SOCKADDR_IN& ClientAddr, int&
2  ClientAddrLen, char* message)
3  {
4  u_long mode = 1;
5  ioctlsocket(sockServ, FIONBIO, &mode); // 非阻塞模式
6  int filesize = 0; //文件长度
7
8  int seq = 0; //序列号
9  int ack = 1;
10
11  srand(time(0)); // 初始化随机数生成器
12
13  while (1)
14  {
15      Packet packet1;
16      char* Buffer1 = new char[MAXSIZE + sizeof(packet1)];
17
18      while (recvfrom(sockServ, Buffer1, sizeof(packet1) + MAXSIZE, 0,
19      (sockaddr*)&ClientAddr, &ClientAddrLen) <= 0); //接收报文长度
20
21      memcpy(&packet1, Buffer1, sizeof(packet1));
22
23      // 模拟丢包, 假设丢包概率
24      if (rand() % 100 < 5) {
25          delete[] Buffer1;
26          continue;
27      }
28

```

```
29 //判断是否是结束
30 if (packet1.head.flags == OVER && packetcheck((u_short*)&packet1,
31 sizeof(packet1)) == 0)
32 {
33     cout << " 文件接收完毕" << endl;
34     break;
35 }
36 //处理数据报文
37 else if (packet1.head.flags == 0 &&
38 packetcheck((u_short*)&packet1, sizeof(packet1)) == 0)
39 {
40     //判断是否接受的是别的包
41     if (seq != int(packet1.head.seq))
42     {
43         //cout << "error" << endl;
44         continue; //丢弃该数据包
45     }
46
47     //cout << "seq: " << seq << "head.seq: " << packet1.head.seq
48     << endl;
49     //取出 buffer 中的内容
50     int curr_size = packet1.head.len;
51
52     cout << " 接收到文件大小为 " << curr_size << " bytes! Flag:" <<
53     int(packet1.head.flags) << " SEQ : " << int(packet1.head.seq)
54     << " ACK : " << int(packet1.head.ack) << " CHECK : " <<
55     int(packet1.head.Check) << endl;
56
57     memcpy(message + filesize, Buffer1 + sizeof(packet1),
58     curr_size);
59     //cout << "size" << sizeof(message) << endl;
60     filesize = filesize + curr_size;
61
62     //返回 ACK
63     Packet packet2;
64     char* Buffer2 = new char[sizeof(packet2)];
65
66     packet2.head.flags = FLAG_ACK;
67     //packet2.head.len = 0;
68     packet2.head.seq = seq++;
69     packet2.head.ack = ack++;
70     packet2.head.Check = 0;
```

```

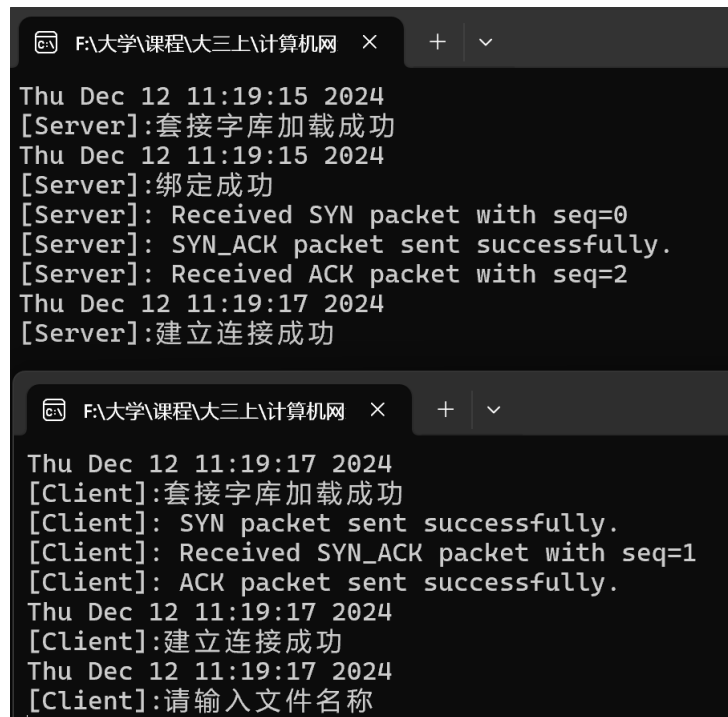
71     packet2.head.Check = packetcheck((u_short*)&packet2,
72     sizeof(packet2));
73     memcpy(Buffer2, &packet2, sizeof(packet2));
74     //重发该包的 ACK
75     sendto(sockServ, Buffer2, sizeof(packet2), 0,
76     (sockaddr*)&ClientAddr, ClientAddrLen);
77
78     cout << " 发送 ACK, 确认数据包发送成功  Flag:" <<
79     int(packet2.head.flags) << " SEQ : " << int(packet2.head.seq)
80     << " ACK : " << int(packet2.head.ack) << " CHECK : " <<
81     int(packet2.head.Check) << endl;
82     if (seq > 255)
83     {
84         seq = seq - 256;
85     }
86     if (ack > 255)
87     {
88         ack = ack - 256;
89     }
90 }
91 else {
92     if (packetcheck((u_short*)&packet1, sizeof(packet1)) != 0) {
93         cout << "error" << endl;
94         system("pause");
95     }
96     cout << " 错误" << endl;
97 }
98 }
99
100 mode = 0;
101 ioctlsocket(sockServ, FIONBIO, &mode); // 阻塞模式
102
103 return filesize;
104 }

```

4 运行结果

4.1 连接建立

经过三次握手连接建立运行结果如下：



```
F:\大学\课程\大三上\计算机网 × + v
Thu Dec 12 11:19:15 2024
[Server]:套接字库加载成功
Thu Dec 12 11:19:15 2024
[Server]:绑定成功
[Server]: Received SYN packet with seq=0
[Server]: SYN_ACK packet sent successfully.
[Server]: Received ACK packet with seq=2
Thu Dec 12 11:19:17 2024
[Server]:建立连接成功

F:\大学\课程\大三上\计算机网 × + v
Thu Dec 12 11:19:17 2024
[Client]:套接字库加载成功
[Client]: SYN packet sent successfully.
[Client]: Received SYN_ACK packet with seq=1
[Client]: ACK packet sent successfully.
Thu Dec 12 11:19:17 2024
[Client]:建立连接成功
Thu Dec 12 11:19:17 2024
[Client]:请输入文件名称
```

图 4.5: 连接建立

4.2 数据传输

客户端发送数据结果如下：1. 慢启动阶段，根据结果可以看出，每传过去一个数据包窗口大小加1，即窗口内所有数据传输一次窗口大小翻倍。

```
慢启动阶段，窗口大小为：2 窗口大小变化为：1
接收到ACK，确认数据包发送成功 Flag:2 SEQ:1 ACK:2 CHECK:26413
发送文件大小为 10240 bytes! Flag:0 SEQ:2 ACK:0 CHECK:16176

慢启动阶段，窗口大小为：3 窗口大小变化为：1
接收到ACK，确认数据包发送成功 Flag:2 SEQ:2 ACK:3 CHECK:26411
发送文件大小为 10240 bytes! Flag:0 SEQ:3 ACK:0 CHECK:16175

慢启动阶段，窗口大小为：4 窗口大小变化为：1
接收到ACK，确认数据包发送成功 Flag:2 SEQ:3 ACK:4 CHECK:26409
发送文件大小为 10240 bytes! Flag:0 SEQ:4 ACK:0 CHECK:16174

慢启动阶段，窗口大小为：5 窗口大小变化为：1
接收到ACK，确认数据包发送成功 Flag:2 SEQ:4 ACK:5 CHECK:26407
发送文件大小为 10240 bytes! Flag:0 SEQ:5 ACK:0 CHECK:16173

慢启动阶段，窗口大小为：6 窗口大小变化为：1
接收到ACK，确认数据包发送成功 Flag:2 SEQ:5 ACK:6 CHECK:26405
发送文件大小为 10240 bytes! Flag:0 SEQ:6 ACK:0 CHECK:16172

慢启动阶段，窗口大小为：7 窗口大小变化为：1
接收到ACK，确认数据包发送成功 Flag:2 SEQ:6 ACK:7 CHECK:26403
```

图 4.6: 慢启动阶段

2. 拥塞避免阶段，根据结果可以看出，窗口内所有数据包传输一次，窗口大小加1。如图所示，窗

口大小为 4，传输 4 个数据包后，窗口大小变为 5，符合线性增长。

```

拥塞避免阶段，窗口大小为：4 窗口大小变化为：1
接收到ACK，确认数据包发送成功  Flag:2 SEQ:31 ACK:32 CHECK:26353

拥塞避免阶段，窗口大小为：4 窗口大小变化为：0
接收到ACK，确认数据包发送成功  Flag:2 SEQ:32 ACK:33 CHECK:26351
发送文件大小为 10240 bytes! Flag:0 SEQ:32 ACK:0 CHECK:16146
发送文件大小为 10240 bytes! Flag:0 SEQ:33 ACK:0 CHECK:16145
发送文件大小为 10240 bytes! Flag:0 SEQ:34 ACK:0 CHECK:16144
发送文件大小为 10240 bytes! Flag:0 SEQ:35 ACK:0 CHECK:16143

拥塞避免阶段，窗口大小为：4 窗口大小变化为：0
接收到ACK，确认数据包发送成功  Flag:2 SEQ:33 ACK:34 CHECK:26349
发送文件大小为 10240 bytes! Flag:0 SEQ:36 ACK:0 CHECK:16142
发送文件大小为 10240 bytes! Flag:0 SEQ:37 ACK:0 CHECK:16141
发送文件大小为 10240 bytes! Flag:0 SEQ:38 ACK:0 CHECK:16140

拥塞避免阶段，窗口大小为：4 窗口大小变化为：0
接收到ACK，确认数据包发送成功  Flag:2 SEQ:34 ACK:35 CHECK:26347

拥塞避免阶段，窗口大小为：5 窗口大小变化为：1
接收到ACK，确认数据包发送成功  Flag:2 SEQ:35 ACK:36 CHECK:26345
发送文件大小为 10240 bytes! Flag:0 SEQ:39 ACK:0 CHECK:16139
发送文件大小为 10240 bytes! Flag:0 SEQ:40 ACK:0 CHECK:16138

```

图 4.7: 拥塞避免阶段

3. 收到 3 个重复 ack，根据结果可以看出，收到 3 个重复的 ack，进入快速恢复阶段，阈值变为原窗口大小/2，窗口大小变为当前阈值加 3，即 $14/2+3=10$ ，符合预期。

此时，当继续受到重复的 ack 时，窗口大小加 1；当接收到正确的 ack 时，进入拥塞避免阶段，窗口大小变为阈值，即 $14/2=7$ ，符合预期。

```

发送文件大小为 10240 bytes! Flag:0 SEQ:27 ACK:0 CHECK:16151
收到重复的ack，序列号为：21 此时ackcount: 1 窗口大小：14 窗口大小变化为：0
收到重复的ack，序列号为：21 此时ackcount: 2 窗口大小：14 窗口大小变化为：0
收到重复的ack，序列号为：21 此时ackcount: 3 窗口大小：14 窗口大小变化为：0

收到3个重复ACK，序列号为：21 重传数据包，进入快速恢复阶段，窗口大小为：10 窗口大小变化为：-4
收到重复的ack，序列号为：21 此时ackcount: 4 窗口大小：11 窗口大小变化为：1
收到重复的ack，序列号为：21 此时ackcount: 5 窗口大小：12 窗口大小变化为：1
发送文件大小为 10240 bytes! Flag:0 SEQ:28 ACK:0 CHECK:16150
发送文件大小为 10240 bytes! Flag:0 SEQ:22 ACK:0 CHECK:16156
发送文件大小为 10240 bytes! Flag:0 SEQ:23 ACK:0 CHECK:16155
收到重复的ack，序列号为：21 此时ackcount: 6 窗口大小：13 窗口大小变化为：1

当前为快速恢复阶段，窗口大小为：7 窗口大小变化为：-6
接收到ACK，确认数据包发送成功  Flag:2 SEQ:22 ACK:23 CHECK:26371
快速恢复阶段进入拥塞避免阶段

```

图 4.8: 收到 3 个重复 ack

4. 触发超时重传时，进入慢启动阶段，阈值为窗口大小/2，窗口大小为 1，符合预期。

```

收到重复的ack, 序列号为: 138 此时ackcount: 6 窗口大小: 11 窗口大小变化为: 1
发送文件大小为 10240 bytes! Flag:0 SEQ:146 ACK:0 CHECK:16032
超时重传, 进入慢启动状态, 当前sssthresh值为: 5
发送文件大小为 10240 bytes! Flag:0 SEQ:139 ACK:0 CHECK:16039
发送文件大小为 10240 bytes! Flag:0 SEQ:140 ACK:0 CHECK:16038
发送文件大小为 10240 bytes! Flag:0 SEQ:141 ACK:0 CHECK:16037

```

图 4.9: 超时重传

服务器端接收数据结果如下:

```

Microsoft Visual Studio 调试 × + v
接收到文件大小为 10240 bytes! Flag:0 SEQ : 175 ACK : 0 CHECK : 16003
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 175 ACK : 176 CHECK : 26065
接收到文件大小为 10240 bytes! Flag:0 SEQ : 176 ACK : 0 CHECK : 16002
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 176 ACK : 177 CHECK : 26063
接收到文件大小为 10240 bytes! Flag:0 SEQ : 177 ACK : 0 CHECK : 16001
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 177 ACK : 178 CHECK : 26061
接收到文件大小为 10240 bytes! Flag:0 SEQ : 178 ACK : 0 CHECK : 16000
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 178 ACK : 179 CHECK : 26059
接收到文件大小为 10240 bytes! Flag:0 SEQ : 179 ACK : 0 CHECK : 15999
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 179 ACK : 180 CHECK : 26057
接收到文件大小为 10240 bytes! Flag:0 SEQ : 180 ACK : 0 CHECK : 15998
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 180 ACK : 181 CHECK : 26055
接收到文件大小为 10240 bytes! Flag:0 SEQ : 181 ACK : 0 CHECK : 15997
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 181 ACK : 182 CHECK : 26053
接收到文件大小为 3913 bytes! Flag:0 SEQ : 182 ACK : 0 CHECK : 22323
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 182 ACK : 183 CHECK : 26051
文件接收完毕
Thu Dec 12 11:23:49 2024
[Server]:文件名接收成功
Thu Dec 12 11:23:49 2024
[Server]:文件内容接收成功
[Server]: Received FIN_ACK packet with seq=0
[Server]: ACK packet sent successfully.
[Server]: FIN_ACK packet sent successfully.
[Server]: Received ACK packet with seq=3
Thu Dec 12 11:23:49 2024
[Server]:断开连接成功

F:\大学\课程\大三上\计算机网络\实验\实验3-3\lab3-3 server\x64\Debug\lab3-3
按任意键关闭此窗口...

```

图 4.10: 服务器端接收数据

服务器端接收数据运行输出接收到文件大小, 标志位, 序列号, 确认号, 校验和, 接收完毕后输出文件接收完毕。

接收文件结果如下:

名称	修改日期	类型	大小
x64	2024/12/1 11:36	文件夹	
1.jpg	2024/12/12 11:23	JPG 文件	1,814 KB
2.jpg	2024/12/12 0:36	JPG 文件	5,761 KB
3.jpg	2024/12/12 0:36	JPG 文件	11,689 KB
helloworld.txt	2024/12/12 0:36	文本文档	1,617 KB
lab3-3 server.cpp	2024/12/6 16:09	C++ Source	17 KB
lab3-3 server.vcxproj	2024/12/1 12:08	VCXPROJ 文件	7 KB
lab3-3 server.vcxproj.filters	2024/12/1 9:38	FILTERS 文件	1 KB
lab3-3 server.vcxproj.user	2024/12/1 9:38	Per-User Project O...	1 KB

图 4.11: 本地接收文件结果

文件成功传输到服务器端，并且文件大小一致。

丢包设置由于发现路由器有传输文件个数的限制，会影响程序传输速率，我在服务器端进行手动丢包。

```
1 // 模拟丢包，假设丢包概率
2 if (rand() % 100 < 5) {
3     delete[] Buffer1;
4     continue;
5 }
```

此时丢包率为 5%。

4.3 断开连接

经过四次挥手连接断开运行结果如下：

```

Microsoft Visual Studio 调试  x  +  v
Microsoft Visual Studio 调试  x  +  v

接收到文件大小为 10240 bytes! Flag:0 SEQ : 17 拥塞避免阶段, 窗口大小为: 2 窗口大小变化为: 0
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 175 接收到ACK, 确认数据包发送成功 Flag:2 SEQ:178 ACK:179 CHECK:26059
接收到文件大小为 10240 bytes! Flag:0 SEQ : 17 发送文件大小为 10240 bytes! Flag:0 SEQ:179 ACK:0 CHECK:15999
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 176
接收到文件大小为 10240 bytes! Flag:0 SEQ : 17 拥塞避免阶段, 窗口大小为: 3 窗口大小变化为: 1
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 177 接收到ACK, 确认数据包发送成功 Flag:2 SEQ:179 ACK:180 CHECK:26057
接收到文件大小为 10240 bytes! Flag:0 SEQ : 17 发送文件大小为 10240 bytes! Flag:0 SEQ:180 ACK:0 CHECK:15998
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 178
接收到文件大小为 10240 bytes! Flag:0 SEQ : 17 拥塞避免阶段, 窗口大小为: 3 窗口大小变化为: 0
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 179 接收到ACK, 确认数据包发送成功 Flag:2 SEQ:180 ACK:181 CHECK:26055
接收到文件大小为 10240 bytes! Flag:0 SEQ : 18 拥塞避免阶段, 窗口大小为: 3 窗口大小变化为: 0
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 180 接收到ACK, 确认数据包发送成功 Flag:2 SEQ:181 ACK:182 CHECK:26053
接收到文件大小为 10240 bytes! Flag:0 SEQ : 18 发送文件大小为 10240 bytes! Flag:0 SEQ:181 ACK:0 CHECK:15997
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 181 发送文件大小为 3913 bytes! Flag:0 SEQ:182 ACK:0 CHECK:22323
接收到文件大小为 3913 bytes! Flag:0 SEQ : 182 发送文件大小为 3913 bytes! Flag:0 SEQ:182 ACK:0 CHECK:22323
发送ACK, 确认数据包发送成功 Flag:2 SEQ : 182
文件接收完毕 拥塞避免阶段, 窗口大小为: 4 窗口大小变化为: 1
Thu Dec 12 11:23:49 2024 接收到ACK, 确认数据包发送成功 Flag:2 SEQ:182 ACK:183 CHECK:26051
[Server]:文件名接收成功 Thu Dec 12 11:23:49 2024
[Server]:文件内容接收成功 [Client]:文件内容传输成功
[Server]: Received FIN_ACK packet with seq=0 传输总时间为:0s
[Server]: ACK packet sent successfully. 吞吐率为:inf byte/s
[Server]: FIN_ACK packet sent successfully. [Client]: Received ACK packet with seq=1
[Server]: Received ACK packet with seq=3 [Client]: Received FIN_ACK packet with seq=2
Thu Dec 12 11:23:49 2024 [Client]: ACK packet sent successfully.
[Server]:断开连接成功 Thu Dec 12 11:23:49 2024
[System]:断开连接成功

F:\大学\课程\大三上\计算机网络\实验\实验3-3\l F:\大学\课程\大三上\计算机网络\实验\实验3-3\lab3-3 client\x64\Debug\lab
按任意键关闭此窗口. . . 按任意键关闭此窗口. . .

```

图 4.12: 断开连接