



南开大学  
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

计算机网络实验报告

---

## 基于滑动窗口的流量控制机制

---

张洋 2111460

年级：2021 级

专业：信息安全

指导教师：吴英

2023 年 12 月 1 日

# 目录

<b>1 实验要求</b>	<b>2</b>
<b>2 前期准备</b>	<b>2</b>
2.1 Go-Back-N(GBN) 流量控制机制	2
2.2 基本原理	2
2.3 GBN 交互示例	3
2.3.1 发送端丢包	3
2.3.2 接收端丢包	3
<b>3 实验代码及解释</b>	<b>3</b>
3.1 数据报格式	4
3.2 建立连接	5
3.3 断开连接	5
3.4 差错检验	5
3.5 日志输出	6
3.6 累积重传 (GBN)	6
3.6.1 总体流程	6
3.6.2 发送端	7
3.6.3 接收端	12
3.7 运行界面	14
3.8 性能测试	17
3.8.1 更改丢包率	17
3.8.2 更改延时	17
3.8.3 更改窗口大小	17
<b>4 问题与总结</b>	<b>18</b>
4.1 单线程更改为多线程	18
4.2 窗口大小	18

## 1 实验要求

在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，发送窗口和接收窗口采用相同大小，支持累积确认，完成给定测试文件的传输。

- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 流水线协议：多个序列号
- 发送缓冲区、接收缓冲区
- 累积确认：Go Back N
- 日志输出：收到/发送数据包的序号、ACK、校验和等，发送端和接收端的窗口大小等情况，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 Go-Back-N(GBN) 流量控制机制

Go-Back-N（GBN）流量控制机制相较于实验 3-1 实现的停等机制有一些显著的改进，主要体现在数据传输效率和网络利用率方面。

GBN 使用滑动窗口的机制，允许发送方连续发送多个数据包，而停等机制一次只能发送一个数据包。这使得 GBN 在网络带宽上更有效利用，提高了数据传输效率。

GBN 允许发送方在等待确认的同时继续发送后续的数据包，形成了一种流水线传输的机制。这意味着在网络上始终存在多个数据包在传输，而不像停等机制那样每次只有一个。这提高了网络的利用率，减少了传输时间。

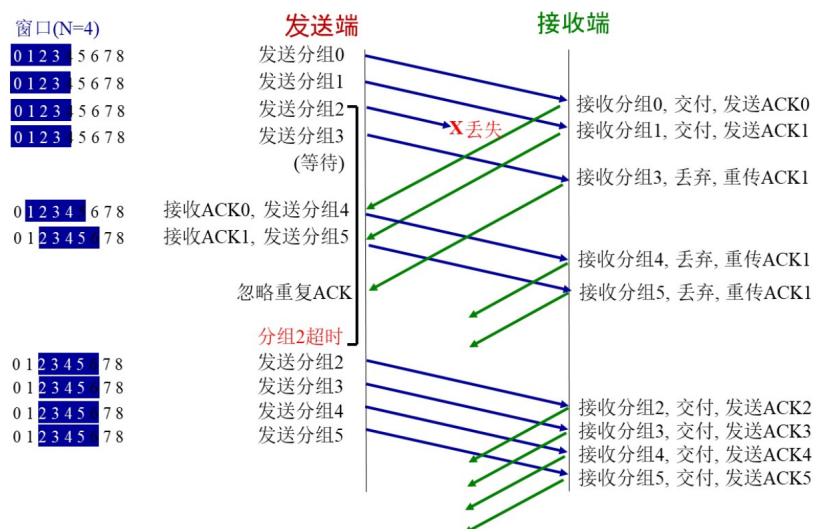
GBN 能够处理少量的丢包，因为发送方能够快速检测到丢失的数据包并进行重传。而停等机制中，如果一个数据包丢失，整个流程会被阻塞，直到确认收到。

### 2.2 基本原理

- 窗口大小：GBN 协议使用了一个发送窗口（Send Window）定义了可以连续发送的最大数据包数量。
- 序列号：每个数据包都有一个唯一的序列号。序列号用于标识数据包的顺序，确保接收方能够正确地按序接收和组装数据。
- 确认和超时：发送方发送窗口中的数据包，并等待接收方的确认。如果在一定时间内没有收到确认，发送方将重新发送窗口内的所有数据包（Go-Back-N 的“Go-Back”部分）。
- 滑动窗口：发送窗口和接收窗口都是滑动的。当接收方成功接收到一个数据包时，它将移动接收窗口，告知发送方可以接收下一个数据包。发送方同样移动发送窗口，以允许发送新的数据包。
- 累积确认：接收方发送的确认表示它已成功接收并正确排序了所有具有较低序列号的数据包。这是因为 Go-Back-N 协议中，只能按序接收，无法选择性地接收某个序列号之前的数据包。



## 2.3 GBN 交互示例



GBN 只确认接收端成功接收的数据包序列中的最大连续序号，如果有一个数据包丢失了，那么在这个数据包之后的所有数据包都需要被重新发送。

### 2.3.1 发送端丢包

当发送端的一个数据包在传输过程中丢失，接收端就不会发送对应的 ACK。发送端设置了一个计时器，如果在一定的时间内没有收到对应的 ACK，发送端就会假设数据包已经丢失，并进行重传。在 GBN 中，发送端会重传该丢失数据包以及其后所有已发送但未确认的数据包。

### 2.3.2 接收端丢包

当接收端的 ACK 在传输过程中丢失，它会确认接收到的最后一个按顺序的包。例如，如果接收端已经接收并确认了序列号为 1, 2, 3 的数据包，但序列号为 4 的 ack 丢失，接收端后续收到了序列号为 5 的数据包，接收端会继续发送 ACK5。

## 3 实验代码及解释

3.1-3.5 中的内容沿用实验 3-1 中的代码，此次实验中实现的内容可以直接[点此跳转](#)到 3.6 实现 GBN 的地方。

### 3.1 数据报格式

我设计的数据报格式如下图所示：

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	...	...	...

```

1 struct Message
2 {
3     //校验和
4     unsigned short checkNum; //2字节
5     //序号 Seq num
6     unsigned short SeqNum; //2字节
7     //确认号 Ack num
8     unsigned short AckNum; //2字节
9     //标志
10    unsigned short flag; //2字节
11    //数据大小
12    unsigned short size; //2字节
13    //源IP、目的IP
14    unsigned int SrcIP, DestIP; //4字节、4字节
15    //源端口号、目的端口号
16    unsigned short SrcPort, DestPort; //2字节、2字节
17
18    BYTE msgData[MaxMsgSize];
19    Message();
20    bool checksum();
21    void setchecksum();
22 };

```

报文头部

flag 段：

SYN: 在 TCP 连接建立过程中（三次握手），SYN 标志被设置以表示初始的序列号。发送 SYN 的一方想要建立一个连接，并在这个 SYN 包中包含它自己的初始序列号。

ACK: ACK 标志被设置表示这个 TCP 包是一个确认包，并且确认号字段是有效的。接收方通过发送 ACK 包，告知发送方它已经接收到了哪些数据。

FIN: FIN 标志被设置表示发送方没有更多的数据需要发送，即它想要关闭连接的一方。这是 TCP 连接终止过程（四次挥手）的一部分。

RST: RST 标志被设置表示 TCP 连接应该被立即重置。这通常在发生错误或者需要立即中断一个连接时使用。

```

1 const unsigned short SYN = 0x1;
2 const unsigned short ACK = 0x2;
3 const unsigned short FIN = 0x4;

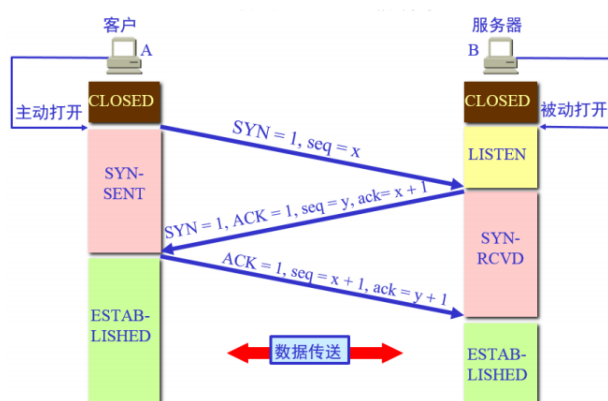
```

```
4 const unsigned short RST = 0x8;
```

flag 标志位

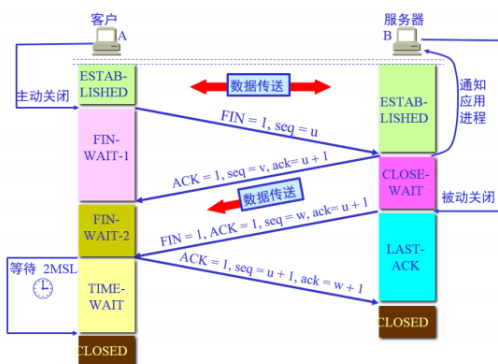
### 3.2 建立连接

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。



### 3.3 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。



### 3.4 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。

```

      1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
      1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
    -----
    ① 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
    -----
      1 0 1 1 1 0 1 1 1 0 1 1 1 0 0
      0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

```

```

1 u_short checksum(u_short* mes, int size) {
2     int count = (size + 1) / 2;
3     u_short* buf = (u_short*)malloc(size + 1);
4     memset(buf, 0, size + 1);
5     memcpy(buf, mes, size);
6     u_long sum = 0;
7     while (count--) {
8         sum += *buf++;
9         if (sum & 0xffff0000) {
10             sum &= 0xffff;
11             sum++;
12         }
13     }
14     return ~(sum & 0xffff);
15 }

```

校验和

### 3.5 日志输出

在客户端会输出一个传输日志，记录了传输总时长和吞吐率。

```

1 //计算传输时间和吞吐率
2 cout << "===== 输出日志 =====" <<
   endl;
3 int endTime = clock();
4 cout << "总体传输时间为：" << (endTime - startTime) / CLOCKS_PER_SEC << " s" <<
   endl;
5 cout << "吞吐率：" << ((float)fileSize) / ((endTime - startTime) /
   CLOCKS_PER_SEC) << " byte/s" << endl << endl;

```

传输日志

### 3.6 累积重传 (GBN)

#### 3.6.1 总体流程

- 客户端：先发送文件名和文件大小，文件名通过报文数据段传递，文件大小通过报文的 size 字段传递。然后依次发送文件内的数据部分，分为最大装载报文和剩余部分的报文。
- 服务器端：收到文件名和文件大小，根据文件大小计算要收到多少个报文依次按顺序等待接收，当收到了所有报文即结束接收。

### 3.6.2 发送端

设置两个线程，一个用来发送，一个用于接收。

- 变量

`int base = 0` : 基序号

`int nextseqnum = 0` : 下一个发送的序号

`int msgStart` : 计时器

`bool over = 0` : 表示数据传输是否结束。当接收线程收到最后一个报文的 `ack`，表示传输结束。此时设置全局变量为 `true`，则发送线程停止发送。

`bool sendAgain = 0` : 表示是否需要三次快速重传。当接收线程收到三次 `ACK`，则设置变量为 `true`，从而发送线程即可开始重新发送报文段。

- 发送线程（主线程）

`nextseqnum < base + N` : 窗口中有就绪报文，可以发送，每发送一个报文，`nextseqnum` 加一。

`timeout` : 超时重传，重传从 `base` 到 `nextseqnum - 1` 的所有报文并重新计时。

**三次 ACK** : 快速重传，重传从 `base` 到 `nextseqnum - 1` 的所有报文并重新计时

```

1 void sendThread(string filename, SOCKADDR_IN routerAddr, SOCKET clientSocket)
2 {
3     int startTime = clock();
4     //截取文件名
5     string realname = "";
6     for (int i = filename.size() - 1; i >= 0; i--)
7     {
8         if (filename[i] == '/' || filename[i] == '\\')
9             break; //去掉文件名开头的空格或换行符
10        realname += filename[i];
11    }
12    realname = string(realname.rbegin(), realname.rend()); //将倒序的文件变回正序
13    // 打开文件，读成字节流
14    ifstream fin(filename.c_str(), ifstream::binary);
15    if (!fin) {
16        printf("无法打开文件! \n");
17        return;
18    }
19    //文件读取到fileBuffer
20    // 创建fileBuffer的BYTE数组，用于存储文件内容，数组大小为 MaxFileSize
21    BYTE* fileBuffer = new BYTE[MaxFileSize];
22    unsigned int fileSize = 0;
23    // 从输入流 fin 中获取一个字节，并存储到变量 byte 中
24    BYTE byte = fin.get();
25    // 循环读取文件内容，直到文件流 fin 的状态变为 false (文件结束)
26    while (fin) {
27        // 将读取到的字节存储到 fileBuffer 数组中，并更新文件大小 fileSize

```



```

28     fileBuffer[fileSize++] = byte;
29     // 继续从输入流 fin 中获取下一个字节
30     byte = fin.get();
31 }
32 fin.close();
33 int batchNum = fileSize / MaxMsgSize; // 全装满的报文个数
34 int leftSize = fileSize % MaxMsgSize; // 不能装满的剩余报文大小
35 // 创建接受消息线程
36 int msgSum = leftSize > 0 ? batchNum + 2 : batchNum + 1;
37 // 判断消息总数, 除了整个文件外, 另加一个说明文件名和文件大小的数据包
38 parameters param;
39 param.routerAddr = routerAddr;
40 param.clientSocket = clientSocket;
41 param.msgSum = msgSum;
42 HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)recvThread,
    &param, 0, 0); // 创建线程, 用于接收ack
43
44 while (1)
45 {
46     // rdt_send(data)
47     if (sendwindow.nextseqnum < sendwindow.base + sendwindow.size &&
        sendwindow.nextseqnum <
        msgSum) // 将要发送的数据号在窗口内, 并且不是最后一个序列号
48     {
49         // make_pkt
50         Message sendMsg;
51         if (sendwindow.nextseqnum == 0) // 第一个分组, 说明文件名和文件大小的消息
52         {
53             sendMsg.SrcPort = ClientPORT; // 原端口
54             sendMsg.DestPort = RouterPORT; // 目的端口
55             sendMsg.size = fileSize; // 文件大小
56             sendMsg.flag += RST; // 文件名
57             sendMsg.SeqNum = sendwindow.nextseqnum;
58             for (int i = 0; i < realname.size(); i++) // 填充报文数据段
59                 sendMsg.msgData[i] = realname[i]; // 文件名写入 Message 的数据部分
60             sendMsg.msgData[realname.size()] = '\0'; // 字符串结尾补\0
61             sendMsg.setchecksum(); // 设置校验和
62         }
63         else if (sendwindow.nextseqnum == batchNum + 1 && leftSize >
            0) // 发送最后一个包
64         {
65             sendMsg.SrcPort = ClientPORT; // 源端口
66             sendMsg.DestPort = RouterPORT; // 目的端口
67             sendMsg.SeqNum = sendwindow.nextseqnum; // 发送序列号
68             for (int j = 0; j < leftSize; j++)
69             {
70                 sendMsg.msgData[j] = fileBuffer[batchNum * MaxMsgSize +
                    j]; // 写入文件数据
71             }

```

```

72         sendMsg.setchecksum();//设置校验和
73     }
74     else//最大装载的数据包
75     {
76         sendMsg.SrcPort = ClientPORT;//原端口
77         sendMsg.DestPort = RouterPORT;//目的端口
78         sendMsg.SeqNum = sendwindow.nextseqnum;//发送序列号
79         for (int j = 0; j < MaxMsgSize; j++)
80             {//写入文件数据
81                 sendMsg.msgData[j] = fileBuffer[(sendwindow.nextseqnum - 1) *
82                     MaxMsgSize + j];
83             }
84         sendMsg.setchecksum();//设置校验和
85     }
86     sendto(clientSocket, (char*)&sendMsg, sizeof(sendMsg), 0,
87         (sockaddr*)&routerAddr, sizeof(SOCKADDR_IN));
88     {
89         std::lock_guard<std::mutex>lock(outputMutex);
90         cout << "[send]: Seq = " << sendMsg.SeqNum << ", checksum = " <<
91             sendMsg.checkNum << endl;
92     }
93
94     if (sendwindow.base == sendwindow.nextseqnum)
95     {
96         msgStart = clock();
97     }
98     //发送报文段，没有接受ACK窗口不滑动
99     //超时
100     if (clock() - msgStart > MAX_WAIT_TIME || sendAgain)//超时或ack重复3次
101     {
102         if (sendAgain) {
103             {
104                 std::lock_guard<std::mutex>lock(outputMutex);
105                 SetTextColor(Red);
106                 cout << "连续收到三次重复ACK，快速重传....." << endl;
107                 SetTextColor(White);
108             }
109         }
110         //重发当前缓冲区的message
111         Message sendMsg;
112         for (int i = 0; i < sendwindow.nextseqnum - sendwindow.base; i++)
113         {
114             int sendnum = sendwindow.base + i;
115             if (sendnum == 0)
116             {
117                 sendMsg.SrcPort = ClientPORT;
118                 sendMsg.DestPort = RouterPORT;

```

```

118         sendMsg.size = fileSize;
119         sendMsg.flag += RST;
120         sendMsg.SeqNum = sendnum;
121         for (int i = 0; i < realname.size(); i++)//填充报文数据段
122             sendMsg.msgData[i] = realname[i];
123         sendMsg.msgData[realname.size()] = '\0';//字符串结尾补\0
124         sendMsg.setchecksum();
125     }
126     else if (sendnum == batchNum + 1 && leftSize > 0)
127     {
128         sendMsg.SrcPort = ClientPORT;
129         sendMsg.DestPort = RouterPORT;
130         sendMsg.SeqNum = sendnum;
131         for (int j = 0; j < leftSize; j++)
132         {
133             sendMsg.msgData[j] = fileBuffer[batchNum * MaxMsgSize + j];
134         }
135         sendMsg.setchecksum();
136     }
137     else
138     {
139         sendMsg.SrcPort = ClientPORT;
140         sendMsg.DestPort = RouterPORT;
141         sendMsg.SeqNum = sendnum;
142         for (int j = 0; j < MaxMsgSize; j++)
143         {
144             sendMsg.msgData[j] = fileBuffer[(sendnum - 1) * MaxMsgSize +
145                                             j];
146         }
147         sendMsg.setchecksum();
148     }
149     sendto(clientSocket, (char*)&sendMsg, sizeof(sendMsg), 0,
150            (sockaddr*)&routerAddr, sizeof(SOCKADDR_IN));
151     {
152         std::lock_guard<std::mutex>lock(outputMutex);
153         SetTextColor(Red);
154         cout << "Seq = " << sendMsg.SeqNum <<
155              "的报文段已超时，正在重传....." << endl;
156         SetTextColor(White);
157     }
158     msgStart = clock();
159     sendAgain = 0;
160 }
161 if (over == 1)//已收到所有ack
162 {
163     break;

```

```

164     }
165 }
166 CloseHandle(hThread);
167 SetTextColor(Green);
168 cout << "已发送并确认所有报文，文件传输成功！" << endl;
169 SetTextColor(White);
170 }

```

### 传输日志

- 接收线程

**接收到未损坏报文：**recvMsg.AckNum >= base 时更新 base，新 base = recvMsg.AckNum + 1；recvMsg.AckNum < base 时不更新 base。

**接收到最后一个确认报文：**设置全局变量 over 标识为 true，传给主线程

**三次冗余：**设置 wrongACK 记录上一次接收到的 ACK，wrongCount 记录重复 ACK 的数量，当 wrongCount 为 3 时，设置全局变量 sendAgain 为 true，传给主线程。

**接收到损坏报文：**丢弃

```

1  DWORD WINAPI recvThread(PVOID pParam)
2  {
3      parameters* para = (parameters*)pParam; //将pParam转换为parameters结构体指针
4      SOCKADDR_IN routerAddr = para->routerAddr;
5      SOCKET clientSocket = para->clientSocket;
6      int msgSum = para->msgSum; //消息的数量
7      int AddrLen = sizeof(routerAddr);
8
9      int wrongACK = -1; //失序的ACK变量
10     int wrongCount = 0; //错误计数器
11     while (1) //等待接收
12     {
13         //rdt_rcv
14         Message recvMsg;
15         int r = recvfrom(clientSocket, (char*)&recvMsg, sizeof(recvMsg), 0,
16             (sockaddr*)&routerAddr, &AddrLen);
17         if (r > 0)
18         {
19             if (recvMsg.checksum()) //检查校验和
20             {
21                 if (recvMsg.AckNum >= sendwindow.base) //确认号ack大于等于窗口左侧
22                     sendwindow.base = recvMsg.AckNum +
23                         1; //窗口右移，base指向等待确认的数据包
24                 msgStart = clock(); //开始计时
25                 if (sendwindow.base != sendwindow.nextseqnum) //base与nextseqnum不重合
26                 {
27                     std::lock_guard<std::mutex>lock(outputMutex);
28                     cout << "[recv]: Ack = " << recvMsg.AckNum << endl;
29                 }
30             }
31         }
32     }
33 }

```

```

28         //打印窗口情况
29         {
30             std::lock_guard<std::mutex>lock(outputMutex);
31             SetTextColor(Red);
32             cout << "[当前窗口情况]: 窗口大小: " << sendwindow.size <<
                 " , 已发送但未收到ACK: " << sendwindow.nextseqnum -
                 sendwindow.base
33                 << " , 尚未发送: " << sendwindow.size - (sendwindow.nextseqnum
                 - sendwindow.base) << endl;
34             SetTextColor(White);
35         }
36         //判断结束的情况
37         if (recvMsg.AckNum == msgSum - 1) //最后一个数据包的ack已接收到
38         {
39             SetTextColor(Green);
40             cout << "已接收到最后一个分组的ACK, 传输完成! " << endl;
41             SetTextColor(White);
42             over = 1;
43             return 0;
44         }
45         //快速重传
46         if (wrongACK != recvMsg.AckNum)
47         { //wrongACK等于上一次接收到的累积确认的ack, 那么这次收到的ACK一定大于wrongACK,
48           //不可能小于, 如果等于则说明数据包丢失
49             wrongCount = 0;
50             wrongACK = recvMsg.AckNum;
51         }
52         else
53         {
54             wrongCount++;
55         }
56         if (wrongCount == 3)
57         {
58             //重发
59             sendAgain = 1;
60         }
61     }
62     //若校验失败或ack不对, 则忽略, 继续等待
63 }
64 }
65 return 0;
66 }

```

传输日志

### 3.6.3 接收端

接收文件名和文件大小, 根据文件大小计算一共将要收到几个报文, 并依次接收这些报文。收到 `recvMsg.SeqNum == expectedSeq` 时, 回复确认报文 `ack = recvMsg.SeqNum`; 收到 `recvMsg.SeqNum`

!= expectedSeq 时, 回复确认报文 ack = expectedSeq - 1。

```

1 bool recvMessage(Message& recvMsg, SOCKET serverSocket, SOCKADDR_IN routerAddr, int&
   expectedSeq)
2 {
3     int AddrLen = sizeof(routerAddr);
4     while (1)
5     {
6         int r = recvfrom(serverSocket, (char*)&recvMsg, sizeof(recvMsg), 0,
           (sockaddr*)&routerAddr, &AddrLen);
7         if (r > 0)
8         {
9             //成功收到消息检查校验和、序列号
10            if (recvMsg.checksum() && (recvMsg.SeqNum == expectedSeq))
11            {
12                //回复ACK
13                Message reply;
14                reply.SrcPort = ServerPORT; //原端口
15                reply.DestPort = RouterPORT; //目的端口
16                reply.flag += ACK; //设置ACK=1
17                reply.AckNum = recvMsg.SeqNum; //ack=seq
18                reply.setchecksum(); //设置校验和
19                sendto(serverSocket, (char*)&reply, sizeof(reply), 0,
           (sockaddr*)&routerAddr, sizeof(SOCKADDR_IN));
20                cout << "[recv]: " << "Seq = " << recvMsg.SeqNum << ", checksum = " <<
           recvMsg.checkNum << endl << "[send]: Ack = " << reply.AckNum <<
           endl;
21                expectedSeq++;
22                return true;
23            }
24            //如果seq!=期待值, 则返回累计确认的ack (expectedSeq-1)
25            else if (recvMsg.checksum() && (recvMsg.SeqNum != expectedSeq))
26            {
27                //回复ACK
28                Message reply;
29                reply.SrcPort = ServerPORT;
30                reply.DestPort = RouterPORT;
31                reply.flag += ACK;
32                reply.AckNum = expectedSeq - 1; //ack=期待值减一
33                reply.setchecksum();
34                sendto(serverSocket, (char*)&reply, sizeof(reply), 0,
           (sockaddr*)&routerAddr, sizeof(SOCKADDR_IN));
35                cout << "[GBN recv]: Seq = " << recvMsg.SeqNum << endl << "[GBN
           send]: Ack = " << reply.AckNum << endl;
36            }
37        }
38        else if (r == 0)
39        {
40            return false;

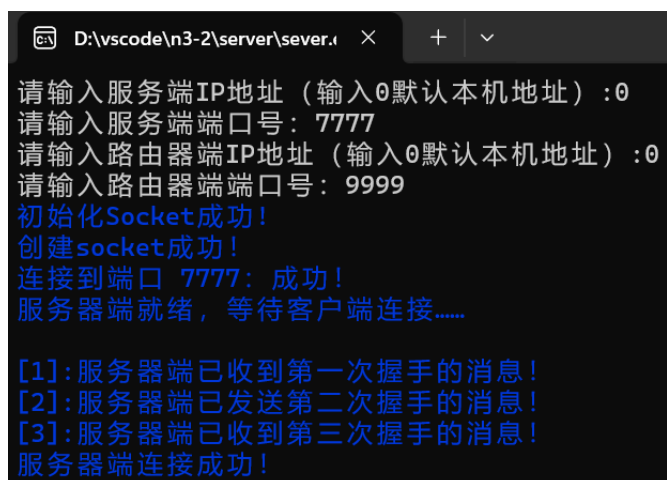
```

```
41     }  
42 }  
43 }
```

传输日志

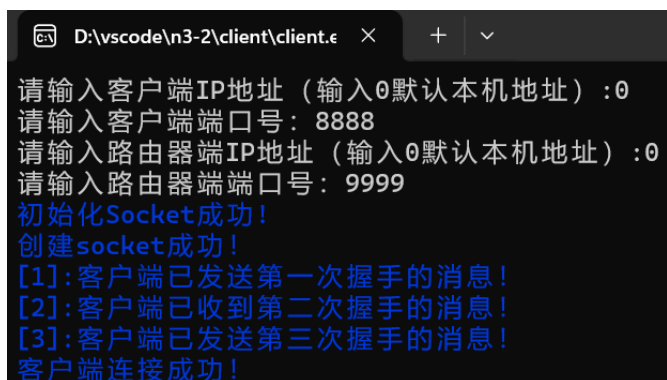
### 3.7 运行界面

服务器端建立连接



```
D:\vscode\n3-2\server\server.i x + v  
请输入服务端IP地址（输入0默认本机地址）：0  
请输入服务端端口号：7777  
请输入路由器端IP地址（输入0默认本机地址）：0  
请输入路由器端口号：9999  
初始化Socket成功！  
创建socket成功！  
连接到端口 7777：成功！  
服务器端就绪，等待客户端连接.....  
  
[1]:服务器端已收到第一次握手的消息！  
[2]:服务器端已发送第二次握手的消息！  
[3]:服务器端已收到第三次握手的消息！  
服务器端连接成功！
```

客户端建立连接



```
D:\vscode\n3-2\client\client.e x + v  
请输入客户端IP地址（输入0默认本机地址）：0  
请输入客户端端口号：8888  
请输入路由器端IP地址（输入0默认本机地址）：0  
请输入路由器端口号：9999  
初始化Socket成功！  
创建socket成功！  
[1]:客户端已发送第一次握手的消息！  
[2]:客户端已收到第二次握手的消息！  
[3]:客户端已发送第三次握手的消息！  
客户端连接成功！
```

发送端(客户端)日志输出

```
请输入文件路径:
1.jpg
[send]: Seq = 0, checksum =49096
[send]: Seq = 1, checksum =50796
[send]: Seq = 2, checksum =29173
[send]: Seq = 3, checksum =34782
[send]: Seq = 4, checksum =24957
[send]: Seq = 5, checksum =5831
[send]: Seq = 6, checksum =41101
[recv]: Ack = 0
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 7, checksum =19075
[recv]: Ack = 1
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 8, checksum =4756
[recv]: Ack = 2
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 9, checksum =61446
[recv]: Ack = 3
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 10, checksum =24518
[recv]: Ack = 4
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 11, checksum =53103
[recv]: Ack = 5
```

接收端 (服务器端) 日志输出

```
[recv]: Seq = 0
[send]: Ack = 0
开始接收数据段, 共 185 个最大装载报文段
[recv]: Seq = 1, checksum =50796
[send]: Ack = 1
数据报1接收成功
[recv]: Seq = 2, checksum =29173
[send]: Ack = 2
数据报2接收成功
[recv]: Seq = 3, checksum =34782
[send]: Ack = 3
数据报3接收成功
[recv]: Seq = 4, checksum =24957
[send]: Ack = 4
数据报4接收成功
[recv]: Seq = 5, checksum =5831
[send]: Ack = 5
数据报5接收成功
```

发送端 (客户端) 三次 ack 重传



```

[send]: Seq = 38, checksum =20210
[recv]: Ack = 32
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 39, checksum =28695
[recv]: Ack = 33
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 6, 尚未发送: 1
[send]: Seq = 40, checksum =64009
[recv]: Ack = 33
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 7, 尚未发送: 0
[recv]: Ack = 33
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 7, 尚未发送: 0
[recv]: Ack = 33
[当前窗口情况]: 窗口大小: 7, 已发送但未收到ACK: 7, 尚未发送: 0
连续收到三次重复ACK, 快速重传.....

```

接收端 (服务器端) 三次 ack 重传

```

[recv]: Seq = 33, checksum =5157
[send]: Ack = 33
数据报33接收成功
[GBN recv]: Seq = 35
[GBN send]: Ack = 33
[GBN recv]: Seq = 36
[GBN send]: Ack = 33
[GBN recv]: Seq = 37
[GBN send]: Ack = 33
[GBN recv]: Seq = 38
[GBN send]: Ack = 33
[GBN recv]: Seq = 39
[GBN send]: Ack = 33
[GBN recv]: Seq = 40
[GBN send]: Ack = 33
[recv]: Seq = 34, checksum =34758
[send]: Ack = 34
数据报34接收成功

```

发送端 (客户端) 超时重传

```

Seq = 36的报文段已超时, 正在重传.....
Seq = 37的报文段已超时, 正在重传.....
Seq = 38的报文段已超时, 正在重传.....
Seq = 39的报文段已超时, 正在重传.....
Seq = 40的报文段已超时, 正在重传.....

```

客户端断开连接

```

关闭连接...
[1]:客户端已发送第一次挥手的消息!
[2]:客户端已收到第二次挥手的消息!
[3]:客户端已收到第三次挥手的消息!
[4]:客户端已发送第四次挥手的消息!
client端2MSL等待...

```

服务器端断开连接

```

文件传输成功，正在写入文件.....
文件写入成功！
[1]:服务端已收到第一次挥手的消息！
[2]:服务器端已发送第二次挥手的消息！
[3]:服务器端已发送第三次挥手的消息！

```

## 3.8 性能测试

### 3.8.1 更改丢包率

在路由器丢包率设置为 2%，5%，延时设置为 1ms，窗口大小设置为 7，设置窗口中传输的最大字节数为 10000 时，四个文件的时延和吞吐率如下表所示。

	2%		5%	
文件名	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）
1.jpg	10	185735	21	88445.4
2.jpg	33	178743	64	92164.1
3.jpg	82	145963	123	97308.9
Helloworld.txt	7	236544	21	78848

可以看到当丢包率由 2% 到 5% 时，传输时间增加一倍。

### 3.8.2 更改延时

在路由器延时设置为 1ms，3ms，丢包率设置为 2%，窗口大小设置为 7，设置窗口中传输的最大字节数为 10000 时，四个文件的时延和吞吐率如下表所示。

	1ms		3ms	
文件名	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）
1.jpg	10	185735	15	123824
2.jpg	33	178743	37	159419
3.jpg	82	145963	90	132989
Helloworld.txt	7	236544	13	127370

### 3.8.3 更改窗口大小

在窗口大小设置为 7，30，路由器延时设置为 1ms，丢包率设置为 2%，设置窗口中传输的最大字节数为 10000 时，四个文件的时延和吞吐率如下表所示。

	7		30	
文件名	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）
1.jpg	10	185735	42	44222.7
2.jpg	33	178743	100	58985.1
3.jpg	82	145963	220	54404.5
Helloworld.txt	7	236544	45	36795.7

## 4 问题与总结

### 4.1 单线程更改为多线程

在单线程实现中，发送数据和接收确认两个过程是串行进行的。也就是说，在发送完一个数据包后，需要等待接收到这个数据包的确认信息，然后再发送下一个数据包。这种方式的问题在于，如果数据包的确认信息因为网络延迟或者丢包等问题没有及时到达，那么就会阻塞后续的数据包发送，导致整体的数据传输效率降低。

在多线程实现中，发送数据和接收确认两个过程是并行进行的。也就是说，发送线程和接收线程是独立的，可以同时运行。在发送线程中，可以不断地发送数据包，不需要等待每个数据包的确认信息。在接收线程中，可以实时接收并处理确认信息。这种方式可以充分利用网络带宽，提高数据传输效率。

### 4.2 窗口大小

可以看到性能测试部分随着窗口大小的增长吞吐率并没有呈现出线性增长的趋势。一般来说，随着窗口的增大，吞吐率也会相应提高，但同时可能会面临一些潜在问题。以下是我通过查阅资料找到的一些可能的影响和应对办法：

- 吞吐率的提高：较大的发送窗口允许同时发送多个数据包，从而增加了网络的利用率，提高了吞吐率。
- 丢包率的影响：随着窗口的增大，网络中的丢包可能导致较大窗口中的多个数据包需要重传，从而降低吞吐率。
- 累计重传次数增加：较大的窗口可能导致一次丢失需要重传的数据包数量增多，因此累计重传的次数可能会相应增加。
- 网络延迟的影响：大窗口可能引入较大的网络延迟，因为发送方需要等待窗口中的所有数据包都被确认才能发送下一批数据。

对策：

- 动态调整窗口大小：可以根据网络状况动态调整发送窗口的大小。在网络较差或丢包率较高时，可以选择较小的窗口，以减少累计重传的影响。
- 拥塞控制机制：结合拥塞控制机制，及时降低发送速率，防止在拥塞情况下大窗口导致的丢包率升高。
- 选择合适的超时时间：调整累计重传的超时时间，确保在网络条件下仍然能够有效地重传丢失的数据包，而不至于过早或过晚触发。
- 快速重传机制：引入快速重传机制，及时重传丢失的数据包，而不必等待累计重传的触发时机。