南開大學
Nankai University

计算机学院

计算机网络实验报告

基于 UDP 服务设计可靠传输协议 (3-3)

姓名：杨馨仪

学号：2011440

专业：计算机科学与技术

2022 年 12 月 25 日

目录

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。

- RENO 算法；
- 也可以自行设计协议或实现其他拥塞控制算法；
- 给出实现的拥塞控制算法的原理说明；
- 有必要日志输出（须显示窗口大小改变情况）。

2 协议设计

在 3-1 实验中我使用数据报套接字实现了面向连接的可靠数据传输。在建立连接的过程中实现了类似于 TCP 的三次握手，实现了从客户端到服务器端以二进制单向传输文件，实现了对消息类型、校验和、序列号等成员的差错检验，实现了 rdt3.0 协议的确认重传功能，采用了停等机制的流量控制方法，断开连接实现了类似于 TCP 的四次挥手功能。

在 3-2 实验中我实现了基于滑动窗口的流量控制机制，采用累计确认和超时重传，差错重传，实现了文件的传输。

本次实验我将在前两次实验的基础上，使用 RENO 算法实现拥塞控制。

2.1 网络拥塞

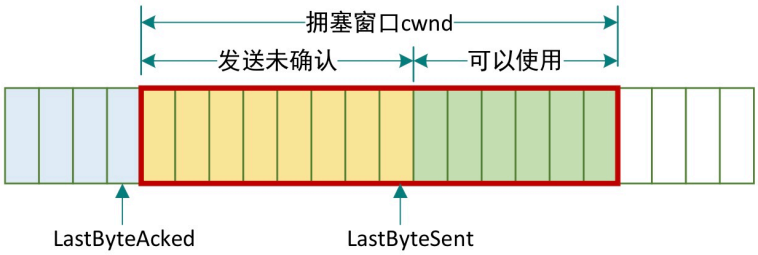
我们都知道，网络错综复杂，在这个复杂的网络中，很少有两台主机是直接相连。尽管如此，我们还是可以通过网络与其他主机通信，这是因为我们发送到网络中的数据，当达到网络中的一个节点时（假设是路由器），它会根据数据包含的地址，帮我们将数据转发到离目的地更近的路由器，或直接转发到目的地。但是，这些路由器不是直接就可以转发的，它们需要先将接收到的数据放入自己的内存（可能还要做一些处理），再从中取出进行转发。

这里就面临一个问题：路由器的内存是有限的，若同一时间到达某个路由器的数据太多，这个路由器将无法接收所有的数据，只能将一部分丢弃；或者同一台路由器数据太多，后面到达的数据将要等待较长的时间才会被转发。网络中的数据太多，导致某个路由器处理不过来或处理地太慢，这就是**网络拥塞**。若是对于 TCP 这种有重传机制的传输协议，当发生数据丢失时，重传数据将延长数据到达的时间；同时，高频率的重传，也将导致网络的拥塞得不到缓解。**拥塞控制**，就是在网络中发生拥塞时，减少向网络中发送数据的速度，防止造成恶性循环；同时在网络空闲时，提高发送数据的速度，最大限度地利用网络资源。

2.2 拥塞控制

1. 如何限制数据的发送速率?
2. 如何检测网络中是否拥塞?
3. 采用什么算法来调整速率（什么时候调整，调整多少）？

对于第一个问题。在我们设计的协议中不是发送一个数据段，接收到确认后再发送另一个数据段，它采用的是流水线的方式。每一个数据段都有一个序号，而我们维护一个发送窗口来发送数据。所有序号位于这个窗口内的数据段都会被一次性发送，而不需要等待之前发送的数据段被确认。而每当最早发送出去的数据段被确认，窗口就会向前移动，直到移动到第一个没有被确认的序号，这时候又会有新的数据段序号被包含在窗口中，然后被发送出去。**所以限制数据发送速率最好的方式就是限制窗口的大小**。客户端会跟踪和维护一个叫做拥塞窗口的变量，用来进行拥塞控制。拥塞窗口被称为 cwnd。在发送端，所有被发送但是还没收到确认的数据段必须落在这个窗口中，所有，当网络拥塞时，程序将减小 cwnd，而网络通畅时，增大 cwnd，以此来控制数据发送的速率。



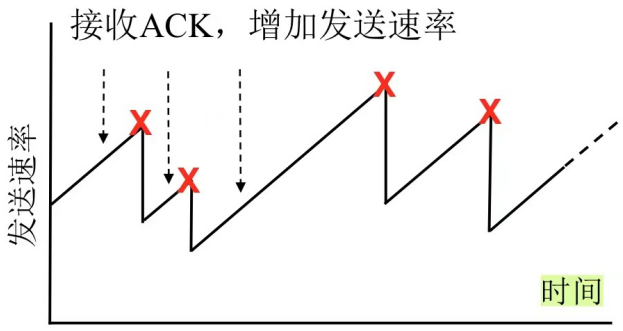
LastByteSent - LastByteAcked ≤ CongestionWindow (cwnd)

对于第二个问题，程序将通过数据发送的一些现象来推测网络是否拥塞，比如：

- 若发送一条数据段后，成功接收到了接收方的确认报文，则可以认为网络没有拥塞；
- 若发送出一条数据段后，在规定时间内没有收到确认报文（丢失或时延太大），则可以认为网络出现了拥塞
- 若连续收到接收方对同一条报文的三次冗余确认，则可以推测那条报文丢失，即发生了拥塞

归根到底，判断拥塞的方式就是检测有没有丢包。

对于第三个问题，如何调整发送速率——在没有丢包时慢慢提高拥塞窗口 cwnd 的大小，当发生丢包事件时，减少 cwnd 的大小。调整拥塞窗口的主要算法有慢启动，拥塞避免以及快速恢复，根据情况在这三者之间切换。下面我就来一一介绍。



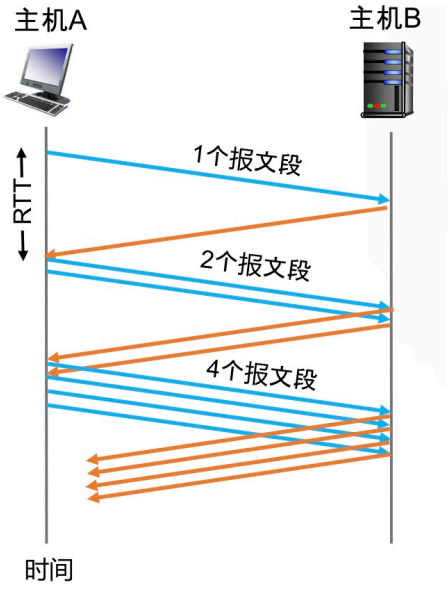
2.3 慢启动阶段

在讲解之前，首先得知道一些名词：

- MSS：最大报文段长度，双方发送的报文段中，包含的数据部分的最大字节数；
- cwnd：拥塞窗口，发送但还没有得到确认的报文的序号都在这个区间；
- RTT：往返时间，发送方发送一个报文，到接收这个报文的确认报文所经历的时间；
- ssthresh：慢启动阈值，慢启动阶段，若 cwnd 的大小达到这个值，将转换到拥塞避免模式；

慢启动是建立连接后，采用的第一个调整发送速率的模式。在这个阶段，cwnd 通常被初始化为 1MSS，这个值比较小，在这个时候，网络一般还有足够的富余，而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，于是：

- 初始 cwnd=1MSS，所以可以发送一个最大报文段，成功确认后，cwnd = 2MSS；
- 此时可以发送两个最大报文段，成功接收后，cwnd = 4 MSS；
- 此时可以发送四个最大报文段，成功接收后，cwnd = 8 MSS.....



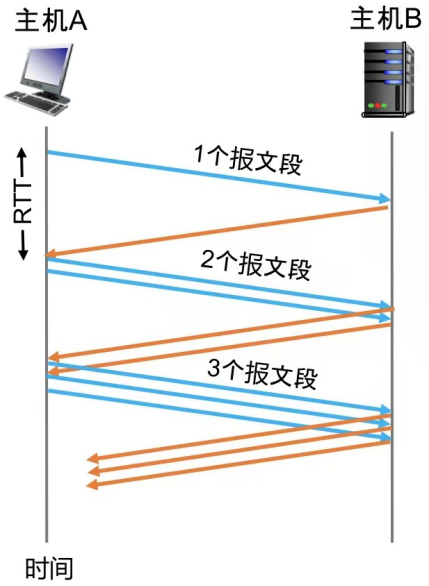
由于是一次性将窗口内的所有报文发出，所以所有报文都到达并被确认的时间，近似的等于一个 RTT。所以在这个阶段，拥塞窗口 cwnd 的长度将在每个 RTT 后翻倍，也就是发送速率将以指数级别增长。那这个过程什么时候改变呢，这又分几种情况：

- 第一种：若在慢启动的过程中，发生了数据传输超时，则此时将 ssthresh 的值设置为 cwnd / 2，然后将 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；
- 第二种：第一步试探出来的上限 ssthresh 将用在此处。若 cwnd 的值增加到 >= ssthresh 时，此时若继续使用慢启动的翻倍增长方式可能有些鲁莽，所以这个时候结束慢启动，改为拥塞避免模式；
- 第三种：若发送方接收到了某个报文的三次冗余确认（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，毕竟发生快速重传也可以认为是发生拥塞导致的丢包，然后 cwnd = ssthresh + 3MSS；

以上就是慢启动的过程，下面来介绍拥塞避免。

2.4 拥塞避免阶段

刚进入这个模式时，cwnd 的大小近似的等于上次拥塞时的值的一半，也就是说当前的 cwnd 很接近产生拥塞的值。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，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 MSS。



这个线性增长的过程什么时候结束，分为两种情况：

- 第一种：在这个过程中，发生了超时，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；
- 第二种：若发送方接收到了某个报文的三次冗余确认（即触发了快速重传的条件），此时也认为发生了拥塞，则，ssthresh 被修改为 cwnd / 2，然后 cwnd 被置为 ssthresh + 3MSS，并进入快速恢复模式；

我们可以看到，慢启动和拥塞避免在接收到三个冗余的确认报文时，处理方式是一样的：判断发生了拥塞，并减小 ssthresh 的大小，但是 cwnd 的大小却不见得有减小多少，这是因为虽然发送方通过接收三次冗余确认报文，判断可能存在拥塞，但是既然可以收到冗余的确认报文，表示拥塞不会太严重，甚至已经不再拥塞，所以对 cwnd 的减小不是这么剧烈。

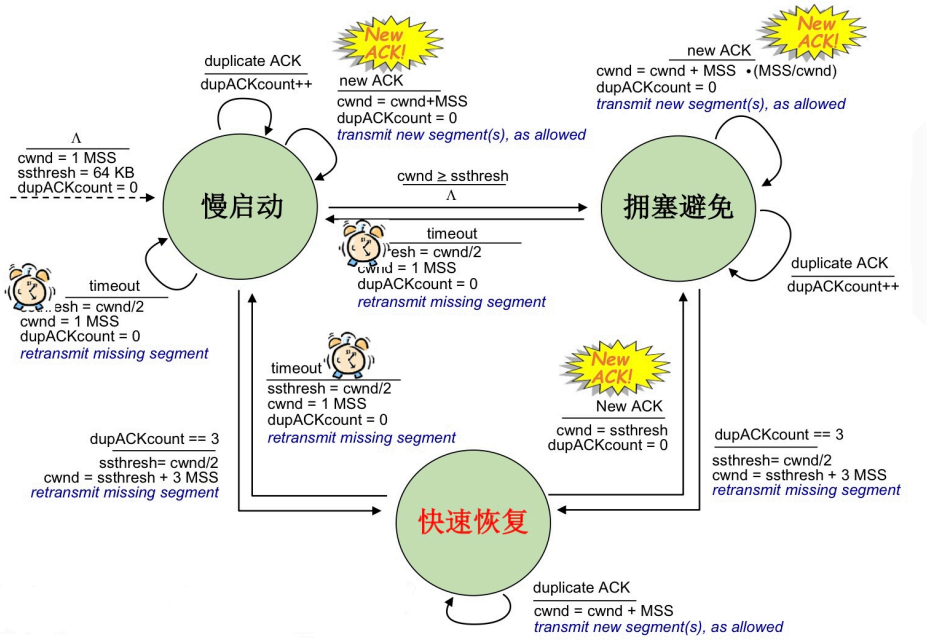
2.5 快速恢复阶段

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：

- 第一种：在快速恢复过程中，计时器超时，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；
- 第二种：若发送方接收到一条新的确认报文（不是冗余确认），则 cwnd 被置为 ssthresh，然后进入到拥塞避免模式；

这里有一个疑问，进入到此模式的条件就是接收到三次冗余的确认报文，判断报文丢失，那为什么再次接收到冗余确认报文时，cwnd 还是要增长呢？此时再次收到一条冗余的确认报文，表示发送端发出的报文又有一条离开网络到达了接收端（虽然不是接收端当前想要的一条），这说明网络中腾出了一条报文的空间，所以允许发送端再向网络中发送一条报文。但是由于当前序号最小的报文丢失，导致拥塞窗口 cwnd 无法向前移动，于是只好将 cwnd 增加 1MSS，于是发送端又可以发送一条数据段，提高了网络的利用率。

2.6 三种模式相互转换状态图



详细设计方式将在下一部分进行具体介绍。

3 模块功能介绍

3.1 UDP 协议的基本框架

在这里我们以客户端为例，介绍 UDP 协议的基本框架。在此次客户端的设计中我才用了双线程的模式，其中一个线程负责接收报文，而主线程负责发送报文，大致结构如下代码所示：

```
1  DWORD WINAPI recvMsgThread(LPVOID Iparam)//接收消息的线程
2  {
3    SOCKET sockClient = *(SOCKET*)Iparam;
4    Package p1;
5    int addrlen = sizeof(SOCKADDR);
6    while (1)
7    {
8      if (recvfrom(sockClient, (char*)&p1, sizeof(p1), 0, (SOCKADDR*)&addrSrv,
         &addrlen) > 0)
9      {
10       ...
```

```
    }
  }
  return 0L;
}

void main()
{
  WSADATA wsaData;
  WSAStartup(MAKEWORD(2, 2), &wsaData);

  SOCKET sockClient = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  SOCKADDR_IN addrSrv = { 0 };
  addrSrv.sin_family = AF_INET;
  addrSrv.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");//设置为本地回环地址
  addrSrv.sin_port = htons(4001);

  SOCKADDR_IN addrClient;
  int len = sizeof(SOCKADDR);

  HandShake(sockClient, addrSrv);

  hMutex = CreateMutex(NULL, FALSE, L"screen");
  CloseHandle(CreateThread(NULL, 0, recvMsgThread, (LPVOID)&sockClient, 0, 0));

  while (1)
  {
    ...// 以二进制方式打开文件
    SendMsg(data, sockClient, addrSrv, dataLen, Buf[0]);
  }
  WaveHand(sockClient, addrSrv) == true
  closesocket(sockClient);
  WSACleanup();
}
```

由上述代码我们可以分析出：首先在主线程中，客户端（服务器端）开始工作，初始化 Socket DLL，协商使用的 Socket 版本，创建一个 socket，并绑定到 UDP 传输层服务。然后先对服务器端的地址信息（包括 IP 地址与端口号）进行初始化，服务器端使用 bind 函数将本地地址绑定到服务器 socket 上。接着进行三次握手操作并创建线程，之后进入循环，对于用户输入的想要传输给服务器的文件进行传输。然后在用户的指示下，结束数据传输，主动断开连接，进行四次挥手操作。最后关闭服务器 socket，并且释放 socket DLL 资源。

在第二线程中，主要负责对从服务器端接收到的消息进行处理。

服务器端的结构与此类似，只不过没有第二线程，不是读取文件而是存入文件上有所区别。

3.2 数据报格式介绍

```
struct HeadMsg {
```

```
    u_long dataLen;
    u_short len;
    u_short checkSum;
    unsigned char type;
    unsigned char seq;
    unsigned char fileNum;
    unsigned char fileTyp;
};

struct Package {
    HeadMsg hm;
    char data[8000];
};
```

本次实验中我定义了两个结构体，分别为 HeadMsg 用于存放首部，Package 用于存放整个数据报。

HeadMsg 中有七个成员：

- dataLen，4 个字节，用于存放整个文件的长度
- len，2 个字节，用于存放本数据报的数据部分长度（数据不能超过 8191 字节）
- checkSum，2 个字节，用于存放校验和
- type，1 个字节，用于存放标志位（包括 SYN，SYN_ACK，ACK，FIN_ACK，PSH，NAK 等）
- seq，1 个字节，用来存放序列号（0-255）
- fileNum，1 个字节，用来存放此次发送的文件编号
- fileTyp，1 个字节，用于存放文件类型

Package 有两个成员：

- hm，16 个字节，用于存放数据报头部
- data，8000 个字节，用于存放数据部分

3.3 数据报发送

本次实验实现重点为累计确认，超时重传，滑动窗口等。

在客户端与服务器端实现三次握手建立连接之后，客户端开始出传输文件，用序列号 seq 来标记传输的报文，server 回复 ACK，如果客户端收到服务器端发来的 ACK 的序列号 seq 与自己待确认的 seq 相等，则表示已经收到，窗口根据 ack 按照相应的大小向前滑动。使用队列记录每个已经发出并且没有收到回复报文 ack 的发送，判断队列中第一个记录的时间是否超时，超时则清空队列，并将窗口中的内容进行重传操作，ssthresh 被修改为 cwnd / 2，然后 cwnd 被置为 1MSS，并进入慢启动阶段。具体实现如以下几个部分所示。

3.3.1 读取文件

```
char Buf[BUFFER_SIZE] = {};
cin.getline(Buf, BUFFER_SIZE);
char* data;
if (strcmp(Buf, "1") == 0 || strcmp(Buf, "2") == 0 || strcmp(Buf, "3") == 0 ||
    strcmp(Buf, "4") == 0)
{
  char file[100] = "..\\test\\";
  if (Buf[0] == '1' || Buf[0] == '2' || Buf[0] == '3')
    sprintf(file, "%s%c.jpg", file, Buf[0]);
  else
    sprintf(file, "%s%s", file, "helloworld.txt");
  ifstream in(file, ifstream::in | ios::binary);
  int dataLen = 0;
  if (!in)
  {
    printf("%s [ INFO ] Client: can't open the file! Please retry\n", timei());
    continue;
  }
  // 文件读取到data
  BYTE t = in.get();
  char* data = new char[100000000];
  memset(data, 0, sizeof(data));
  while (in)
  {
    data[dataLen++] = t;
    t = in.get();
  }
  in.close();
  printf("read over\n");
```

首先根据交互获得用户想要发送的文件，然后将文件打开，将数据读取到 data，并存入文件数据的总长度，再将文件关闭。如此完成文件读取的过程。

3.3.2 发送信息与超时重传

对于发送消息的过程：我们首先计算该文件应当被切分为多少个数据报，然后初始化 head 和 tail 两个变量用来标记当前窗口大小，head 用来指向下一个待确认的数据报，tail 指向已经发送的数据报。然后我们循环将这些切分好的数据报发送给服务器端，注意在这个过程中，我们需要判断当前发送出去的数据报个数是否超过了窗口的大小 WINDOW_SIZE 以及是否已经发送结束。如果满足发送的条件，我们为每一个数据报设置头部信息，其中包括序列号，校验和，标志位等等。然后发送信息，把 tail+1 代表又发出了一个数据报，并且将刚刚发出的数据报的序列号与发送时间记录到队列之中。

如果我们发现当前的时间减去队列中第一个数据报的发送时间已经大于等待时间 WAIT_TIME 了，那么我们重新将 tail 赋值为 head-1，代表需要将窗口内的内容进行重传，ssthresh 被修改为 cwnd / 2，然后 cwnd 被置为 1MSS，并进入慢启动阶段。

接下来来看客户端具体的实现。

```
bool SendMsg(char* data, SOCKET sockClient, SOCKADDR_IN addrSrv, int dataLen, char
    fileNum)
{
  int pcknum = ceil(dataLen / 8000.0);
  head = 0, tail = -1; //head为等待确认的pkg tail为已经发送的最后一个pkg
  while (head <= pcknum - 1)
  {
    WaitForSingleObject(hMutex, INFINITE);
    if (tail - head + 1 < (int)cwnd && tail != pcknum - 1)
        //如果没超过窗口大小，且没发到结尾
    {
      // 设置信息头
      Package p;
      p.hm.dataLen = dataLen;
      p.hm.seq = (tail + 1) % 256;
      p.hm.type = PSH;
      p.hm.checkSum = 0;
      p.hm.fileNum = fileNum;
      if (fileNum == '1' || fileNum == '2' || fileNum == '3')
        p.hm.fileTyp = JPG;
      else
        p.hm.fileTyp = TXT;
      if (tail != pcknum - 2)
        p.hm.len = 8000;
      else
        p.hm.len = dataLen % 8000;
      memcpy(p.data, data + (tail + 1) * 8000, p.hm.len); //把本个包的数据存进去
      // 计算校验和
      p.hm.checkSum = checkSumVerify((u_short*)&p, sizeof(p));
      SendPkg(p, sockClient, addrSrv);
      tail++;
      printf("%s [ SEND ] Client: WINDOW SIZE: %d, SSTHRESH: %d, WINDOW HEAD: %d,
          WINDOW TAIL: %d, TOTAL NUM: %d\n", timei(), (int)floor(cwnd), ssthresh,
          head, tail, pcknum);
      timer_list.push(make_pair(tail + 1, clock()));
    }
    ReleaseMutex(hMutex);

    //判断发送时间
    WaitForSingleObject(hMutex, INFINITE);
    if (timer_list.size() != 0)
    {
      if ((clock() - timer_list.front().second) > WAIT_TIME)
      {
        tail = head - 1;
        ssthresh = (int)max((cwnd / 2), 1);
        cwnd = 1;
        dupACKcount = 0;
        printf("%s [ ERR  ] Client: wait for a long time error\n", timei());
```

```
46          printf("%s [ INFO ] Client: WINDOW SIZE: %d, SSTHRESH: %d, WINDOW HEAD: %d,
                WINDOW TAIL: %d, TOTAL NUM: %d\n", timei(), (int)floor(cwnd), ssthresh,
                head, tail, pcknum);
47          while (timer_list.size()) timer_list.pop();
48        }
49      }
50      ReleaseMutex(hMutex);
51    }
52    return true;
53  }
```

在上面 SendMsg 函数中调用了 SendPkg 函数，接下来是对这个函数的具体介绍：

其实对于发送数据这个步骤来说，这个函数只是完成了 sendto 函数的工作。但该函数在三次握手与四次挥手中也会用到，里面大部分内容与握手挥手相关，在此统一做出讲解，后续不再赘述。

客户端 SendPkg 函数

```
1   bool SendPkg(Package p, SOCKET sockClient, SOCKADDR_IN addrSrv)
2   {
3     char Type[10];
4     switch (p.hm.type) {
5     case 1: strcpy(Type, "SYN"); break;
6     case 4: strcpy(Type, "ACK"); break;
7     case 8: strcpy(Type, "FIN_ACK"); break;
8     case 16:strcpy(Type, "PSH"); break;
9     }
10    // 发送消息
11    while (sendto(sockClient, (char*)&p, sizeof(p), 0, (SOCKADDR*)&addrSrv,
          sizeof(SOCKADDR)) == -1)
12    {
13      printf("%s [ ERR  ] Client: send [%s] ERROR:%s Seq=%d\n", timei(), Type,
            strerror(errno), p.hm.seq);
14    }
15    printf("%s [ INFO ] Client: [%s] Seq=%d\n", timei(), Type, p.hm.seq);
16
17    if (!strcmp(Type, "ACK") || !strcmp(Type, "PSH"))
18      return true;
19    // 开始计时
20    clock_t start = clock();
21    // 等待接收消息
22    Package p1;
23    int addrlen = sizeof(SOCKADDR);
24    while (true) {
25      if (recvfrom(sockClient, (char*)&p1, sizeof(p1), 0, (SOCKADDR*)&addrSrv,
            &addrlen) > 0 && clock() - start <= WAIT_TIME) {
26        // 收到消息需要验证消息类型、序列号和校验和
27        u_short ckSum = checkSumVerify((u_short*)&p1, sizeof(p1));
28        if ((p1.hm.type == SYN_ACK && !strcmp(Type, "SYN")) && p1.hm.seq == seq &&
              ckSum == 0)
```

```
        {
          printf("%s [ GET  ] Client: receive [SYN, ACK] from Server\n", timei());
          return true;
        }
        else if ((p1.hm.type == ACK && (!strcmp(Type, "FIN_ACK") || !strcmp(Type,
            "PSH"))) && p1.hm.seq == seq && ckSum == 0)
        {
          printf("%s [ GET  ] Client: receive [ACK] from Server\n", timei());
          return true;
        }
        else {
          SendPkg(p, sockClient, addrSrv);
          return true;
        }
      }
      else {
        SendPkg(p, sockClient, addrSrv);
        return true;
      }
    }
}
```

这里以客户端的 SendPkg 为例。首先这个函数的参数包括打包好的 Package p, 客户端的 socket 以及服务器端的地址。首先我们获取数据报中首部的标志位信息，并将其以字符串的形式存放在 Type 中。然后发送这个数据报给服务器端。如果这个数据报的标志位只有 ACK 置 1 或只有 PSH 置一，那么直接返回；否则开始计时，使用 recvfrom 函数接收消息。如果当前数据报标志位为 SYN，并且接收到了 SYN_ACK 的数据报，在确认序列号与校验和无误后，直接返回；如果当前数据报标志位为 FIN_ACK，并且接收到了 ACK 的数据报，在确认序列号与校验和无误后，直接返回；如果不是以上三种情况，若序列号或校验和错误则触发差错重传；如果超时没有接收到数据报，则超时重传。

以上是它的整个功能，部分还要在三次握手和四次挥手中用到的，在此介绍过之后，之后不再赘述。

3.3.3 服务器接收消息

对于服务器，他在完成一系列初始化工作后，便等待由客户端发来的数据报，我们采用一个死循环来接受被切分的数据报，接收到数据报之后，我们首先确认他的类型是否是 PSH，并且确认其校验和无误，然后在判断传递过来的这个数据报的序号是否是我们期待的想要收到的那个序列号 lastAck，如果是的话，我们新建一个数据报，将类型设置为 ACK，将序列号设置为已经确认收到的序列号即 lastAck, 并把 lastAck+1(不要忘记要模 256)，设置它的校验和，并把它发送给客户端；如果不是我们期待收到的数据报，那么我们也创建一个数据报，但把他的序列号设置为 lastAck-1，然后发送给客户端。如果接收到的数据长度等于文件总长度了，那么代表文件接收成功。

```
RecvData RecvMsg(SOCKET sockSrv, SOCKADDR_IN addrClient)
{
  Package p1;
  int addrlen = sizeof(SOCKADDR);
  int totalLen = 0;
```

```
6   RecvData rd;
7   int lastAck = 0;
8   rd.data = new char[100000000];
9   // 等待接收消息
10  while (true) {
11    // 收到消息需要验证校验和及序列号
12    if (recvfrom(sockSrv, (char*)&p1, sizeof(p1), 0, (SOCKADDR*)&addrClient,
          &addrlen) > 0)
13    {
14      if (p1.hm.type == FIN_ACK)
15      {
16        cout <<
              "----------------------------DISCONNECTION--------------------------" <<
              endl;
17        rd.fileNum = '0';
18        WaveHand(sockSrv, addrClient, p1);
19        break;
20      }
21
22      Package p2;
23      p2.hm.fileTyp = p2.hm.fileNum = p2.hm.checkSum = p2.hm.dataLen = p2.hm.len = 0;
24
25      int ck = !checkSumVerify((u_short*)&p1, sizeof(p1));
26      if (p1.hm.type == PSH && ck == 1)
27      {
28        if (p1.hm.seq != lastAck)
29        {
30          p2.hm.type = ACK;
31          p2.hm.seq = lastAck -1;
32          p2.hm.checkSum = checkSumVerify((u_short*)&p2, sizeof(p2));
33          while (sendto(sockSrv, (char*)&p2, sizeof(p2), 0, (SOCKADDR*)&addrClient,
                sizeof(SOCKADDR)) == -1)
34            printf("%s [ ERR  ] Server: send [ACK] ERROR:%s Seq=%d\n", timei(),
                  strerror(errno), p2.hm.seq);
35          printf("%s [ INFO ] Server: [ACK] Seq=%d\n", timei(), p2.hm.seq);
36          continue;
37        }
38        p2.hm.type = ACK;
39        p2.hm.seq = lastAck;
40        lastAck = (lastAck + 1) % 256;
41        p2.hm.checkSum = checkSumVerify((u_short*)&p2, sizeof(p2));
42        while (sendto(sockSrv, (char*)&p2, sizeof(p2), 0, (SOCKADDR*)&addrClient,
              sizeof(SOCKADDR)) == -1)
43          printf("%s [ ERR  ] Server: send [ACK] ERROR:%s Seq=%d\n", timei(),
                strerror(errno), p2.hm.seq);
44        printf("%s [ INFO ] Server: [ACK] Seq=%d\n", timei(), p2.hm.seq);
45        memcpy(rd.data + totalLen, (char*)&p1 + sizeof(p1.hm), p1.hm.len);
46        totalLen += (int)p1.hm.len;
47        rd.dataLen = p1.hm.dataLen;
```

```
48          rd.fileNum = p1.hm.fileNum;
49          if (totalLen == p1.hm.dataLen)
50          {
51            Sleep(3000);
52            break;
53          }
54        }
55      }
56    }
57    return rd;
58  }
```

3.3.4 客户端接收消息

那么客户端对于由服务器端发来的消息需要进行确认，同样在接收到数据报后要先对数据报类型与校验和进行检验，确认无误后，再检查它的序列号。

如果此时 dupACKcount >= 3，说明已经接收端到三次重复的 ACK 了，如果这次还是重复的 ACK，那么我们将窗口大小 cwnd 增大 1MSS. 如果这时我们收到了新的 ACK，则把 cwnd 赋为阈值 ssthresh 的大小，将 dupACKcount 赋为 1。

如果此时 dupACKcount < 3，判断此次是否是重复的 ACK，相应的将 dupACKcount 增加。如果此时是新的 ACK，判断其属于慢启动阶段还是拥塞避免阶段，按照相应的规则改变窗口大小。如果这时 dupACKcount == 3 了，那么 ssthresh 设为 cwnd/2,cwnd = ssthresh + 3，重新发送窗口内内容。

在完成状态转换以及窗口大小和阈值的变化之后，我们判断序列号的无误，有以下两种情况。

- 接收到的序列号比 head 模 256 要大，这是代表，接收到的序列号之前的那些数据报都已经被确认。
- 由于模运算的关系，接收到的序列号比 head 模 256 要小，但在整个文件中的位置是在 head 之后的，这种情况我们要特别处理 head 将要指向的内容。

对于两种无误的情况，我们要将队列中的已经被确认的数据报的发送时间 pop 出来，并且将 head 指向接下来待确认的数据报处。

如果我们发现接收到的类型或校验和有误，这时我们将 tail 赋为 head-1，将窗口内容进行重传处理。

客户端接收线程

```
DWORD WINAPI recvMsgThread(LPVOID Iparam)//接收消息的线程
{
  SOCKET sockClient = *(SOCKET*)Iparam;//获取客户端的SOCKET参数
  Package p1;
  int addrlen = sizeof(SOCKADDR);
  while (1)
  {
    if (recvfrom(sockClient, (char*)&p1, sizeof(p1), 0, (SOCKADDR*)&addrSrv,
        &addrlen) > 0)
```

```
    {
      int ck = !checkSumVerify((u_short*)&p1, sizeof(p1));
      WaitForSingleObject(hMutex, INFINITE);
      if (p1.hm.type != ACK && ck != 1)  //类型错误或校验和有误
      {
        tail = head - 1;
        printf("%s [ ERR  ] Client: receive wrong PKG from Server   Seq = %d\n",
            timei(), p1.hm.seq);
        printf("%s [ INFO ] Client: WINDOW SIZE: %d, SSTHRESH: %d, WINDOW HEAD: %d,
            WINDOW TAIL: %d\n", timei(), (int)floor(cwnd), ssthresh, head, tail);
        continue;
      }
      else
      {
        if (dupACKcount >= 3)
        {
          if (p1.hm.seq == head - 1)
            cwnd += 1;
          else
          {
            cwnd = ssthresh;
            dupACKcount = 1;
          }
        }
        else
        {
          if (p1.hm.seq == head - 1)
          {
            dupACKcount++;
          }
          else {
            if (cwnd < ssthresh)
            {
              cwnd++;
              dupACKcount = 1;
            }
            if (cwnd >= ssthresh)
            {
              cwnd += 1 / cwnd;
              dupACKcount = 1;
            }
          }
          if (dupACKcount == 3)
          {
            ssthresh = (int)max((cwnd / 2),1);
            cwnd = ssthresh + 3;
            tail = head - 1;
            while (timer_list.size()) timer_list.pop();
          }
```

```
56
57          }
58
59          int acknum = 0;
60          if (p1.hm.seq >= head % 256)
61          {
62            acknum = p1.hm.seq - head % 256 + 1;
63            head = head + acknum;
64            while (timer_list.size() != 0 && acknum != 0)
65            {
66              timer_list.pop();
67              acknum--;
68            }
69            printf("%s [ GET  ] Client: receive [ACK] from Server   Seq = %d\n",
                  timei(), p1.hm.seq);
70            printf("%s [ INFO ] Client: WINDOW SIZE: %d, SSTHRESH: %d, WINDOW HEAD: %d,
                  WINDOW TAIL: %d\n", timei(), (int)floor(cwnd), ssthresh, head, tail);
71          }
72          else if (head % 256 > 256 - floor(cwnd) && p1.hm.seq < floor(cwnd) - (256 -
                head % 256))
73          {
74            acknum = 256 - head % 256 + p1.hm.seq + 1;
75            head = head + acknum;
76            while (timer_list.size() != 0 && acknum != 0)
77            {
78              timer_list.pop();
79              acknum--;
80            }
81            printf("%s [ GET  ] Client: receive [ACK] from Server   Seq = %d\n",
                  timei(), p1.hm.seq);
82            printf("%s [ INFO ] Client: WINDOW SIZE: %d, SSTHRESH: %d, WINDOW HEAD: %d,
                  WINDOW TAIL: %d\n", timei(), (int)floor(cwnd), ssthresh, head, tail);
83          }
84        }
85        ReleaseMutex(hMutex);
86      }//释放互斥量锁
87    }
88    return 0L;
89  }
```

3.4 建立连接——三次握手

三次握手连接建立过程:

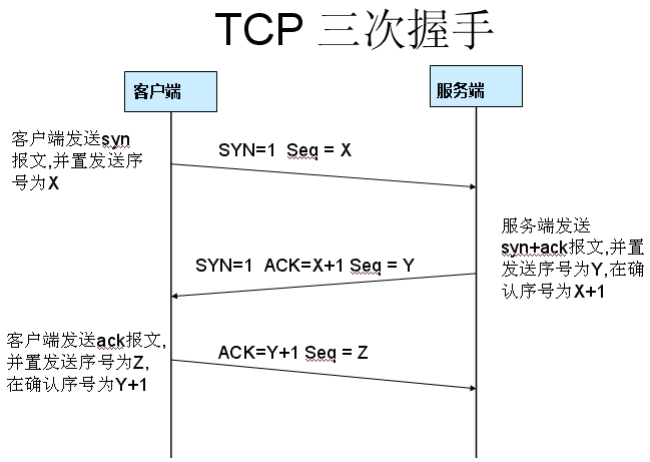
Step1: Client 将标志位 SYN 置为 1，产生 seq 值为 J，并将该数据包发送给 Server，Client 进入 SYN_SENT 状态，等待 Server 确认，这是第一次握手。

Step2: Server 收到数据包后由标志位 SYN=1 知道 Client 请求建立连接，Server 将标志位 SYN 和 ACK 都置为 1，产生序列号 seq 为 J，代表确认收到客户端的建立连接请求。并将该数据包发送给

Client 以确认连接请求，Server 进入 SYN_RCVD 状态, 这是第二次握手。

Step3： Client 收到确认后，检查 seq 是否为 J，ACK,SYN 标志位是否为 1，如果正确则将标志位 ACK 置为 1，并将该数据包发送给 Server，Server 检查 ACK 是否为 1，如果正确则连接建立成功，Client 和 Server 进入 ESTABLISHED 状态，完成三次握手，随后 Client 与 Server 之间可以开始传输数据了。

连接建立，开始通讯。



然后我们来看一看客户端与服务器端的握手函数：

客户端 HandShake 函数

```
bool HandShake(SOCKET sockClient, SOCKADDR_IN addrSrv)
{
  char sendBuf[BUFFER_SIZE] = {};
  cin.getline(sendBuf, BUFFER_SIZE);
  if (strcmp(sendBuf, "connect") != 0)
    return false;
  Package p1;
  p1.hm.type = SYN;
  p1.hm.seq = seq;
  p1.hm.checkSum = 0;
  p1.hm.checkSum = checkSumVerify((u_short*)&p1, sizeof(p1));
  int len = sizeof(SOCKADDR);
  SendPkg(p1, sockClient, addrSrv);
  seq = (seq + 1) % 256;
  p1.hm.type = ACK;
  p1.hm.seq = seq;
  p1.hm.checkSum = 0;
  p1.hm.checkSum = checkSumVerify((u_short*)&p1, sizeof(p1));

  if (sendto(sockClient, (char*)&p1, sizeof(p1), 0, (SOCKADDR*)&addrSrv,
      sizeof(SOCKADDR)) != -1)
  {
    printf("%s [ INFO ] Client: [ACK] Seq=%d\n", timei(), seq);
    seq = (seq + 1) % 256;
    return true;
```

```
  }
  else
  {
    printf("%s [ ERR  ] Client: send [ACK] ERROR\n", timei());
    return false;
  }
}
```

服务器端 HandShake 函数

```
bool HandShake(SOCKET sockSrv, SOCKADDR_IN addrClient)
{
  Package p2;
  int len = sizeof(SOCKADDR);
  while (true)
  {
    if (recvfrom(sockSrv, (char*)&p2, sizeof(p2), 0, (SOCKADDR*)&addrClient, &len) >
        0)
    {
      int ck = checkSumVerify((u_short*)&p2, sizeof(p2));
      if (p2.hm.type == SYN && ck == 0)
      {
        printf("%s [ GET  ] Server: receive [SYN] from Client\n", timei());
        Package p3;
        p3.hm.type = SYN_ACK;
        p3.hm.seq = (lastAck + 1) % 256;
        lastAck = (lastAck + 2) % 256;
        p3.hm.checkSum = 0;
        p3.hm.checkSum = checkSumVerify((u_short*)&p3, sizeof(p3));
        SendPkg(p3, sockSrv, addrClient);
        break;
      }
      else
      {
        printf("%s [ ERR  ] Server: receive [SYN] ERROR\n", timei());
        return false;
      }
    }
  }
  return true;
}
```

通过服务器端与客户端的多次交互，我们可以发现实现了开始所说的三次握手功能，并且其中也包含有差错检验与确认重传，超时重传的相关功能。

3.5 断开连接——四次挥手

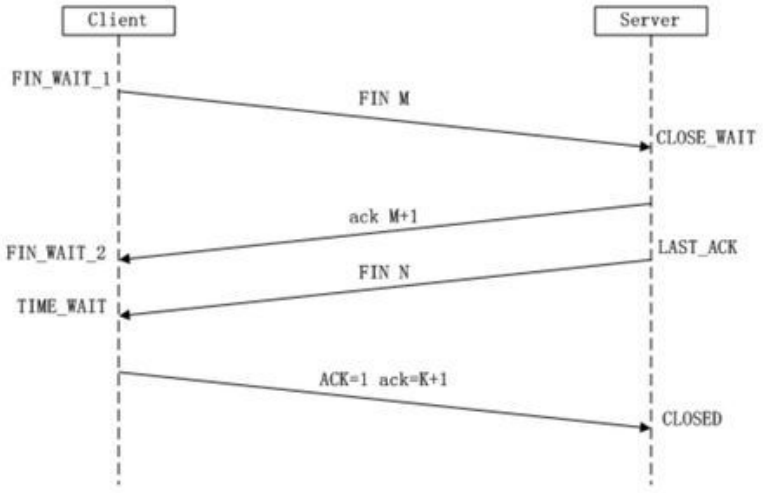
四次挥手即终止连接，就是指断开一个连接时，需要客户端和服务端总共发送 4 个包以确认连接的断开。

Step1: Client 发送一个 FIN，用来关闭 Client 到 Server 的数据传送，Client 进入 FIN_WAIT_1 状态。

Step2: Server 收到 FIN 后，发送一个 ACK 给 Client，序号为收到序号，Server 进入 CLOSE_WAIT 状态。

Step3: Server 发送一个 FIN，用来关闭 Server 到 Client 的数据传送，Server 进入 LAST_ACK 状态。

Step4: Client 收到 FIN 后，Client 进入 TIME_WAIT 状态，接着发送一个 ACK 给 Server，序号为收到序号，Server 进入 CLOSED 状态，完成四次挥手。



实验中我们通过 WaveHand 函数来实现四次挥手功能，在其中依旧调用了 SendMsg 函数。

客户端 WaveHand 函数

```
bool WaveHand(SOCKET sockClient, SOCKADDR_IN addrSrv)
{
  Package p1;
  p1.hm.type = FIN_ACK;
  p1.hm.seq = seq;
  p1.hm.checkSum = 0;
  p1.hm.checkSum = checkSumVerify((u_short*)&p1, sizeof(p1));
  int len = sizeof(SOCKADDR);

  SendPkg(p1, sockClient, addrSrv);
  Package p4;

  while (true)
  {
    if (recvfrom(sockClient, (char*)&p4, sizeof(p4), 0, (SOCKADDR*)&addrSrv, &len) >
        0)
    {
```

```
        if (p4.hm.type == FIN_ACK)
        {
          printf("%s [ GET  ] Client: receive [FIN, ACK] from Server\n", timei());
          p4.hm.type = ACK;
          p4.hm.checkSum = 0;
          p4.hm.checkSum = checkSumVerify((u_short*)&p4, sizeof(p4));
          SendPkg(p4, sockClient, addrSrv);
          break;
        }
        else
        {
          printf("%s [ ERR  ] Server: receive [FIN, ACK] ERROR\n", timei());
          return false;
        }
      }
    }
    return true;
}
```

服务器端 WaveHand 函数

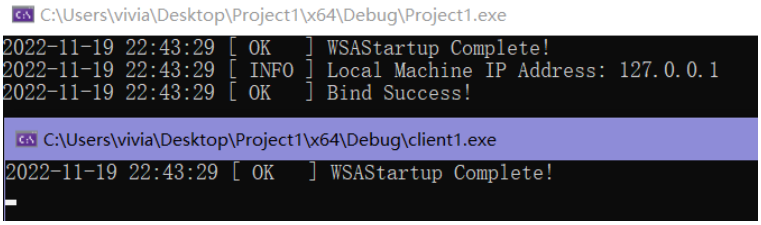
```
bool WaveHand(SOCKET sockSrv, SOCKADDR_IN addrClient, Package p2)
{
    int len = sizeof(SOCKADDR);
    u_short ckSum = checkSumVerify((u_short*)&p2, sizeof(p2));
    if (p2.hm.type == FIN_ACK && ckSum == 0)
    {
      printf("%s [ GET  ] Server: receive [FIN, ACK] from Client\n", timei());
      p2.hm.type = ACK;
      p2.hm.seq = (lastAck + 1) % 256;
      lastAck = (lastAck + 2) % 256;
      p2.hm.checkSum = 0;
      p2.hm.checkSum = checkSumVerify((u_short*)&p2, sizeof(p2));
      SendPkg(p2, sockSrv, addrClient);
    }
    else
    {
      printf("%s [ ERR  ] Server: receive [FIN, ACK] ERROR\n", timei());
      return false;
    }

    Package p3;
    p3.hm.type = FIN_ACK;
    p3.hm.checkSum = 0;
    p3.hm.checkSum = checkSumVerify((u_short*)&p2, sizeof(p2));
    SendPkg(p3, sockSrv, addrClient);
    return true;
}
```

局部代码不能体现完整流程，但大致的框架如上所示，具体实现详见源码。

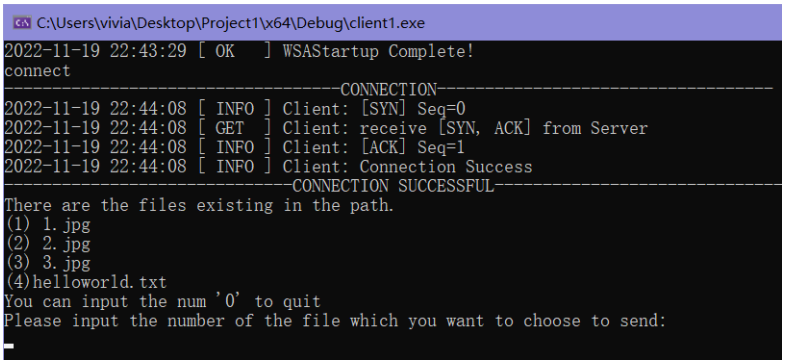
4 程序界面展示及运行说明

step1: 首先设置丢包率如下所示：

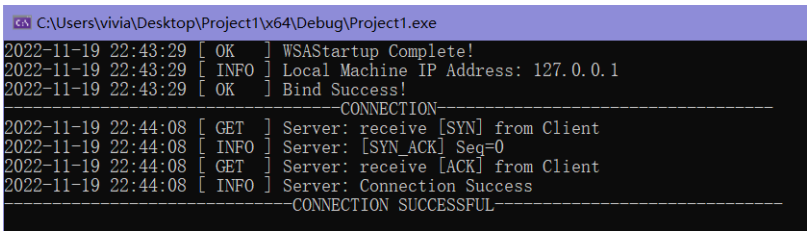
step2: 通过 visual studio 运行程序，如下是客户端与服务器端的一些初始打印信息



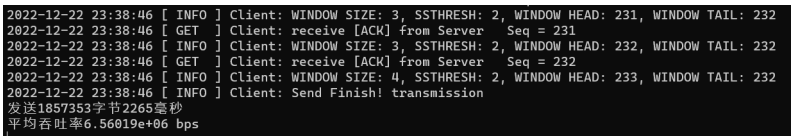
step3: 在客户端输入 connect，进行三次握手，建立连接，如下为客户端的截图



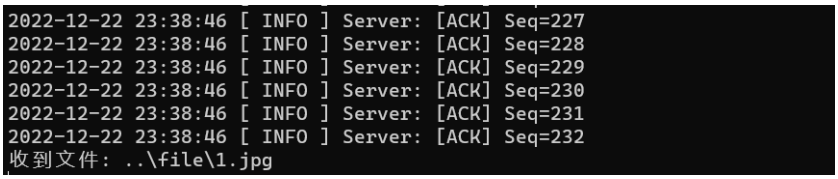
如下为服务器端的截图



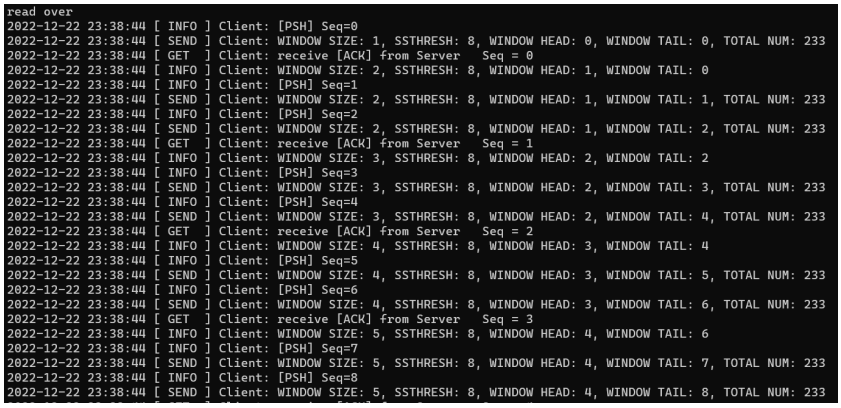
step4: 在客户端输入 1，开始对第一张图片进行可靠性传输；传输完毕显示发送时延，发送字节数以及平均吞吐量等信息



服务器端显示收到文件



我们可以看到滑动窗口的变化，当窗口大小 cwnd 小于阈值 ssthresh 时，属于慢启动阶段，这时每接收到一个新的 ACK，窗口大小就增大 1MSS，head 后移一位。当窗口内还有未发送的数据报时，会一直发送数据报，每发送一个数据报，tail 不断后移直到达到窗口大小 WINDOW_SIZE，以此类推



当窗口大小 cwnd 达到阈值 ssthresh 时，进入拥塞避免阶段，此时我们发现收到新的 ACK 时，不会将窗口的大小 +1 了，这时窗口大小呈线性增长。

```
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=13
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 7, WINDOW TAIL: 13, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=14
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 7, WINDOW TAIL: 14, TOTAL NUM: 233
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 7
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 8, WINDOW TAIL: 14
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=15
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 8, WINDOW TAIL: 15, TOTAL NUM: 233
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 8
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 9, WINDOW TAIL: 15
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=16
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 9, WINDOW TAIL: 16, TOTAL NUM: 233
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 9
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 10, WINDOW TAIL: 16
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=17
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 10, WINDOW TAIL: 17, TOTAL NUM: 233
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 10
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 11, WINDOW TAIL: 17
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=18
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 11, WINDOW TAIL: 18, TOTAL NUM: 233
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 11
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 8, SSTHRESH: 8, WINDOW HEAD: 12, WINDOW TAIL: 18
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=19
```

直到发生了丢包事件，如下图所示我们迟迟未收到第 30 个包的 ACK 确认信息，当我们重复收到三次第 29 个包的确认信息时，我们将重新设置阈值大小为原窗口大小的一半，窗口大小为新阈值大小 +3。然后将 tail 指向 head-1 的部分，重新发送窗口中的内容。

```
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 10, SSTHRESH: 8, WINDOW HEAD: 30, WINDOW TAIL: 38, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=39
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 10, SSTHRESH: 8, WINDOW HEAD: 30, WINDOW TAIL: 39, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=30
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 30, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=31
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 31, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=32
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 32, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=33
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 33, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=34
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 34, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=35
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 35, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=36
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 36, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=37
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 37, TOTAL NUM: 233
```

我们可以看到此时在快速恢复阶段，当还在不断收到重复的 ACK 时，窗口的大小会不断的增加。直到我们终于收到了第 30 个 ACK 确认信息，这时设置窗口大小为阈值大小，回到拥塞避免阶段。

```
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 36, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=37
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 8, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 37, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=38
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 9, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 38, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=39
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 10, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 39, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=40
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 11, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 40, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=41
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 12, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 41, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=42
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 13, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 42, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=43
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 14, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 43, TOTAL NUM: 233
2022-12-22 23:38:44 [ INFO ] Client: [PSH] Seq=44
2022-12-22 23:38:44 [ SEND ] Client: WINDOW SIZE: 15, SSTHRESH: 5, WINDOW HEAD: 30, WINDOW TAIL: 44, TOTAL NUM: 233
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 30
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 5, SSTHRESH: 5, WINDOW HEAD: 31, WINDOW TAIL: 44
2022-12-22 23:38:44 [ GET  ] Client: receive [ACK] from Server   Seq = 31
2022-12-22 23:38:44 [ INFO ] Client: WINDOW SIZE: 5, SSTHRESH: 5, WINDOW HEAD: 32, WINDOW TAIL: 44
```

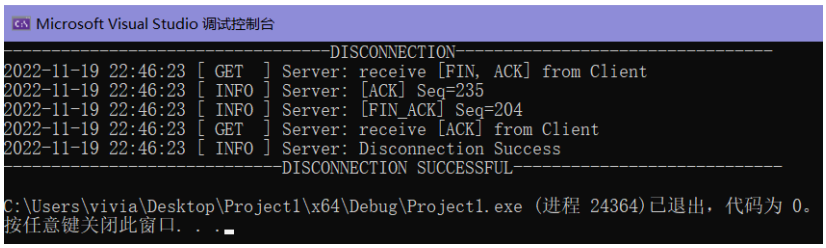
step5: 输入 5，提示没有第五个文件

```
5
2022-11-19 22:46:05 [ ERR  ] Client: Invalidate Input
```

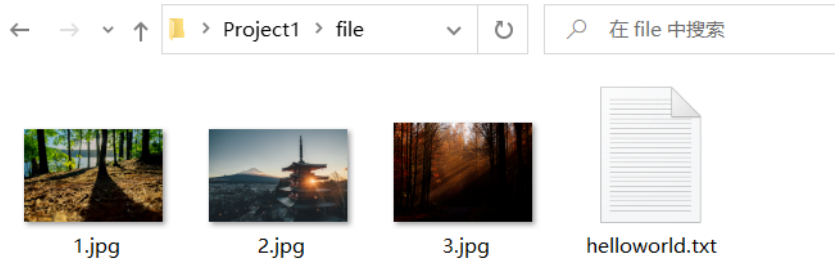
step6: 输入 0，进行四次挥手，断开连接，如下是客户端截图

```
Microsoft Visual Studio 调试控制台
--------------------------------DISCONNECTION--------------------------------
2022-11-19 22:46:23 [ INFO ] Client: [FIN_ACK] Seq=235
2022-11-19 22:46:23 [ GET  ] Client: receive [ACK] from Server
2022-11-19 22:46:23 [ GET  ] Client: receive [FIN, ACK] from Server
2022-11-19 22:46:23 [ INFO ] Client: [ACK] Seq=204
2022-11-19 22:46:23 [ INFO ] Client: Disconnection Success
-----------------------------DISCONNECTION SUCCESSFUL-----------------------------

C:\Users\vivia\Desktop\Project1\x64\Debug\client1.exe (进程 23464)已退出，代码为 0。
按任意键关闭此窗口. . .
```

如下是服务器端截图



step7: 打开文件夹，可以看到成功传输的图片与文本文件



5 实验反思

在本次实验中，按照课上所讲的 RENO 算法状态转换图，成功实现了拥塞控制，经检验，窗口大小及阈值变化无误，达到了期待的效果。