

计算机网络实验报告

实验：Lab3-3 基于UDP服务设计可靠传输协议

学号：2012682

姓名：韩佳迅

计算机网络实验报告

实验：Lab3-3 基于UDP服务设计可靠传输协议

学号：2012682

姓名：韩佳迅

一、实验内容

二、协议设计

(一) 拥塞控制

new RENO 协议：

(1) *RENO* 协议

(2) *New RENO* 协议

(二) 报文格式【同3-2】

(三) 数据传输：滑动窗口、累计确认、快速重传【同3-2】

1. 滑动窗口

2. 累计确认

接收端：

发送端：

3. 快速重传

GBN状态机

(四) 建立连接：三次握手【同3-2】

(五) 关闭连接：四次挥手【同3-2】

三、各模块功能与具体实现

(一) 报文段【同3-2】

结构体实现报文段

UDP校验和

(二) 数据传输：拥塞控制、滑动窗口、累计确认、快速重传

客户端：

(1) 多线程实现

(2) 拥塞控制的状态机

服务器端

(三) 建立连接【同3-2】

客户端

服务器端

(四) 关闭连接【同3-2】

客户端

服务器端

三、程序界面与运行

建立连接

传输数据

关闭连接

一、实验内容

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

二、协议设计

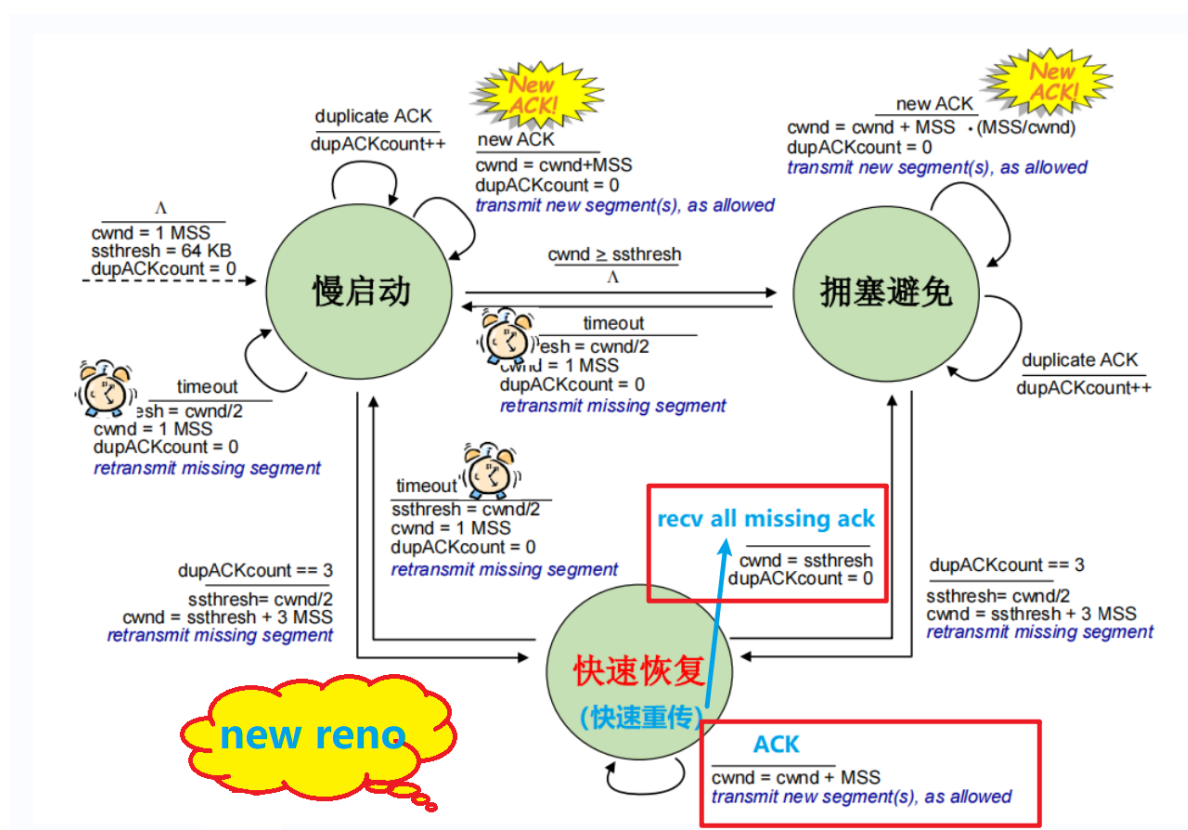
(一) 拥塞控制

本次实验基于 **new RENO** 协议，实现了**拥塞控制**、**快速重传**。

网络拥塞：主机发送的数据过多或过快，造成网络中的路由器（或其他设备）无法及时处理，从而引入时延或丢弃。

拥塞控制：拥塞控制通过调整单次发送的分组数量（单次发送量，cwnd）来处理网络拥塞。它的实现主要是通过增减单次发送量逐步调整，使之逼近当前网络的承载量。

new RENO 协议：



(1) RENO 协议

Reno协议规定，在处理网络拥塞时，存在三种状态：慢启动、拥塞避免、快速恢复（快速重传）。

• 慢启动

含义：

在连接刚建立时，为了试探网络的拥塞情况，先只发送一个报文段。

慢启动开始后，再逐渐增大拥塞窗口 $cwnd$ ，每经过一个传输轮次， $cwnd$ 加倍（即 $cwnd$ 的值随传输轮次线性增长）这样可以使分组注入网络的速率更加合理。

算法规则：

- 初始：在连接刚建立并开始发送报文段的时候，先令拥塞窗口 $cwnd = 1$ （一个最大报文段长度 MSS ）
- new ACK：每当收到一个 new ACK 时， $cwnd$ 加一，即增大一个 MSS
- duplicate ACK：当收到重复的ACK时，记录重复ACK及重复次数
- 超时：拥塞窗口 $cwnd$ 缩减为 $1 MSS$ ，阈值 $ssthresh$ 变为当前窗口大小的一半

状态转换：

- ——> **拥塞避免**：当 $cwnd$ 增大到了阈值 $ssthresh$ ，则进入拥塞避免阶段
- ——> **快速恢复**：当收到三次冗余的ACK，则进入快速恢复阶段，并且重传未收到确认的报文段

阈值 $ssthresh$ 变为当前窗口大小的一半

$cwnd$ 变为 $ssthresh + 3 MSS$

• 拥塞避免

含义：

拥塞避免状态的思路是让 $cwnd$ 缓慢增大（相较于慢启动和快速恢复阶段）。

算法规则：

- new ACK：每经过一个往返时延就把 $cwnd$ 加一（而非加倍），使得 $cwnd$ 按线性规律缓慢增长。
- duplicate ACK：当收到重复的ACK时，记录重复ACK及重复次数

状态转换：

- ——> **慢启动**：超时进入慢启动阶段

拥塞窗口 $cwnd$ 缩减为 $1 MSS$ ，

阈值 $ssthresh$ 变为当前窗口大小的一半

- ——> **快速恢复**：当收到三次冗余的ACK，则进入快速恢复阶段，并且重传未收到确认的报文段

阈值 $ssthresh$ 变为当前 $cwnd$ 的一半

$cwnd$ 变为 $ssthresh + 3 MSS$

• 快速恢复（快速重传）

含义：

当发送端收到三个冗余ACK：

说明网络可能有报文丢失，此时发送端会重发尚未确认的报文，而不用等待计时器超时。

也说明当前网络可能存在拥塞，为了预防拥塞，阈值会被缩减为 $cwnd$ 的一半，同时 $cwnd$ 也会缩小

在此阶段，如果还能收到 ACK，说明网络还没有发生严重拥塞，因此还可以适当提高窗口 $cwnd$ 大小

算法规则：

- duplicate ACK：当收到重复的ACK时，将 $cwnd$ 增大一个MSS，人为地扩充拥塞窗口用以反映已经离开网络的附加数据段。
- new ACK：设定 $cwnd$ 值为 $ssthresh$ ，可视为给窗口“放气”。

状态转换：

-  ——> 慢启动：超时进入慢启动阶段

拥塞窗口 $cwnd$ 缩减为 1 MSS，

阈值 $ssthresh$ 变为当前窗口大小的一半

-  ——> 拥塞避免：收到 new ACK

$cwnd$ 缩减为 $ssthresh$

(2) New Reno 协议

在实际的网络中，一旦发生拥塞，会丢弃大量的包。如果采用 **Reno** 算法，它会认为网络中发生了多次拥塞，则会多次将 $cwnd$ 和 $ssthresh$ 减半，造成吞吐量极具下降，当发送窗口小于3时，将无法产生足够的 ACK 来触发快重传而导致超时重传，产生较大影响。

New reno 的改进：**Reno** 快速恢复算法中，发送方只要收到一个新的ACK就会退出快速恢复状态而进入拥塞避免阶段，**New reno** 算法中，只有当所有丢失的包都重传并收到确认后才会退出快速恢复状态。

（状态机对应上图中的红色标记部分）

(二) 报文格式【同3-2】

报文格式如下所示：



报文格式分为首部和数据部分：

首部：

- 1—4字节：源IP
- 5—8字节：目的IP
- 9—10字节：源端口号
- 11—12字节：目的端口号
- 13—16字节：序号
- 17—20字节：确认号
- 21—24字节：数据大小
- 25—26字节：标志
- 27—28字节：校验和

数据：

- 剩余字节流是数据部分

报文段的校验：

在报文段里设置校验和字段，发送时设置校验和，收到时检验校验和是否正确。若不正确，说明报文损坏。

(三) 数据传输：滑动窗口、累计确认、快速重传【同3-2】

本次实验基于 **Go-Back-N (GBN)** 协议，添加了**三次快速重传**，实现了**滑动窗口**、**累计确认**。

流水线：由于停等机制在每一次发送报文时，都需要等待上个报文的确认报文被接收到，才可以继续发送，因此会产生很长的等待时延，效率低下。而流水线协议则指的是，在确认未返回之前允许发送多个分组，从而提高传输效率。

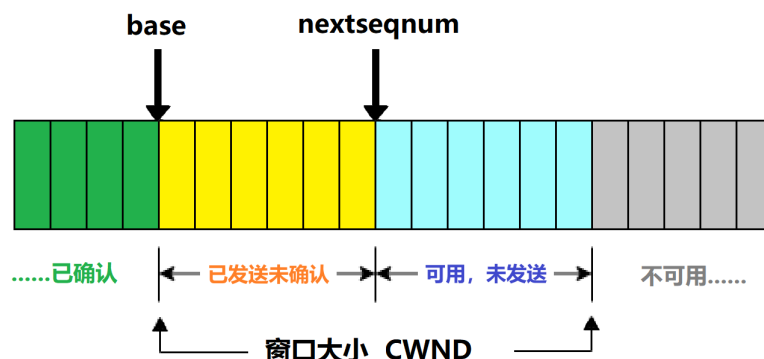
为了实现流水线协议，在停等机制的基础上需要实现：

- **序列号的扩展：**从0、1二值，需要扩展到更大的范围才够表示

- **发送端、接收端的缓冲区：**由于每次发送不止一个报文，且不确定一次发送的多个报文是否被确认，因此需要在发送端、接收端设置缓冲区。

1. 滑动窗口

发送窗口如下：



实验中的流量控制采用发送端（固定）滑动窗口的方法。

- 设置窗口大小固定为 **CWND**。
- 设置两个指针，来控制窗口的滑动和下一个序号的发送：
 - **base**：基序号，是滑动窗口的开始位置，指向已发送为确认的第一个序号，或窗口的第一个序号
 - **nextseqnum**：指向下一个要发送的序号
- **理想情况：**
 1. 初始时，**base** 和 **nextseqnum** 都指向第一个序号。
 2. 由于此时窗口中还有报文可以继续发送，因此发送端继续发送报文，每发送一个，**nextseqnum** 右移，直至窗口中可发送的报文都已发送完。
 3. 当发送端收到确认报文，整个窗口右移，**base** 右移（移动到当前已经累计确认的最后一个报文）。
 4. 当窗口移动后，新窗口内有了可以发送的报文，则继续发送新报文。
 5. 持续上述步骤，直到所有报文发送完成。

- **超时重传：**

滑动窗口每发送报文时，会设置定时器。当超时未收到报文时，会重传当前窗口内所有已经发送的报文，即 **base ~ nextseqnum** 的所有报文，而 **base**、**nextseqnum** 位置不变。（除此之外，还实现了快速重传，见下文）

- **失序：**

见累计确认部分。

2. 累计确认

接收端：

理想情况：接收端每接收到发来的报文，若该报文的序号 **等于** 接收端期待接收的报文序号，则回复一个确认报文，确认号 $ack = seq$ ，并接收该报文段，将其交付给上层应用。

失序：当接收端收到了 **不等于** 期待序号值的报文，则回复一个确认报文，确认号 $ack =$ 接收端累计确认的最后一个报文号（也就是 期待接收的报文序号 的前一个值），并丢弃该报文段。

发送端：

当发送端收到确认报文时：

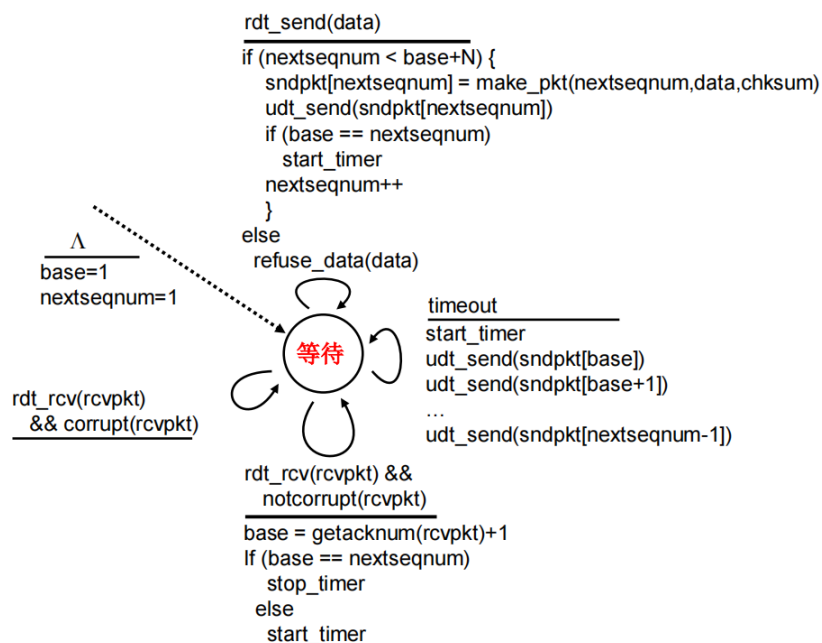
- 首先检查确认报文是否 $ack < base$ ，若小于，说明这是上一次发送的失序确认报文，不移动窗口
- 若确认报文 $ack \geq base$ ，说明这是对此窗口内的确认， $base$ 移动到 $ack+1$ 的位置，窗口右移

3. 快速重传

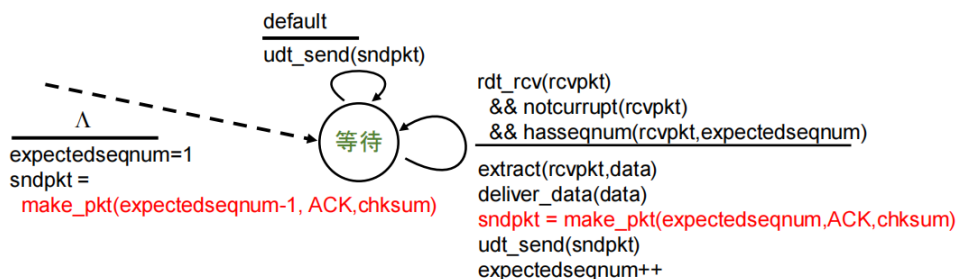
由于接收端每次收到失序的报文时，会回复 $ack =$ 期待值的确认报文，因此当报文丢失或失序时，发送端会连续收到多个重复且冗余的 ACK 报文端。因此，实验中规定，当连续收到三次冗余的 ACK 报文时，可认为报文丢失，需要重传当前窗口内 $base \sim nextseqnum$ 的所有报文。

GBN状态机

- 发送端：

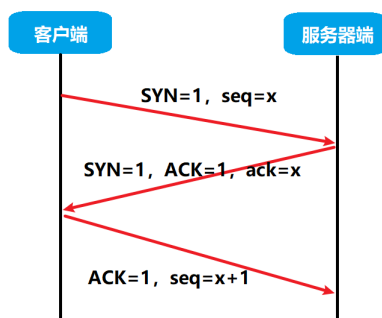


- 接收端：



(四) 建立连接：三次握手【同3-2】

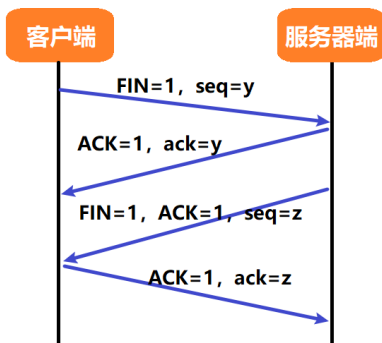
本程序在标准三次握手的基础上，进行了部分改进，最终实现的结构示意图如下：



1. 客户端向服务器端发送数据包：
 - SYN=1, seq=x
2. 服务器端向客户端回复数据包：
 - SYN=1, ACK=1
 - ack=x 【这里对标准三次握手进行了更改：回复的ack = 上一个包发来的seq】
3. 客户端向服务器端发送数据包：
 - ACK=1
 - seq=x+1 【序列号+1】

(五) 关闭连接：四次挥手【同3-2】

本程序在标准四次挥手的基础上，进行了部分改进，最终实现的结构示意图如下：



1. 客户端向服务器端发送数据包：

- FIN=1, seq=y
- 2. 服务器端向客户端回复数据包:
 - ACK=1, ack=y【回复的ack = 上一个包发来的seq】
- 3. 服务器端向客户端发送数据包:
 - FIN=1, ACK=1
 - seq=z
- 4. 客户端向服务器端发送数据包:
 - ACK=1, ack=z【回复的ack = 上一个包发来的seq】

三、各模块功能与具体实现

（一）报文段【同3-2】

结构体实现报文段

- 首部和数据部分由结构体的属性实现
 - 注意：标志位字段使用一个unsigned short变量 *flag* 实现
 - 设置SYN、ACK、FIN字段分别为0x1、0x2、0x4，置位时直接加到 *flag* 上即可
- 功能实现：
 - 设置校验和
 - 检验校验和

UDP校验和

- 设置校验和：
 1. 校验和全部填充为0
 2. 剩余数据部分填充为0
 3. 按16bit为单位反码求和：
 - 【首部+数据】每16位求和得到一个32位数
 - 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
 - 重复直到高16位为0
 - 将低16位取反，得到校验和
 4. 得到的16位结果填充到结构体的校验和字段
- 检验校验和：
 1. 按照设置校验和的方式，将接收到的结构体按16bit为单位反码求和
 2. 如果得出的结果低16位全1，则校验成功

//（该部分使用3-1代码，未做修改）

(二) 数据传输：拥塞控制、滑动窗口、累计确认、快速重传

文件传输总体流程：

- 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

客户端：

- 读取文件内容：
 - 使用 ifstream 以二进制方式打开文件，将文件内容读到BYTE类型数组中

(1) 多线程实现

- 设置两个线程，一个用来发送，一个用于接收：

- **两线程共享变量：**

- `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冗余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

- **主线程：发送**

while循环持续准备发送报文：

- `nextseqnum < base + N` 时：窗口中有就绪报文，可以发送
 make_pkt 创建要发送的报文并 send 发送
 nextseqnum++ 发送一个报文，可发送的序列号++
- timeout 时：超时重传

重传从 base 到 nextseqnum - 1 的所有报文

重新计时

- 三次冗余ACK：快速重传

重传从 base 到 nextseqnum - 1 的所有报文

重新计时

◦ 新线程：接收

while循环持续准备发送报文：

- 接收到未损坏报文：

recvMsg.AckNum >= base：更新base，新base = recvMsg.AckNum + 1

recvMsg.AckNum < base：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

(2) 拥塞控制的状态机

- 使用一个全局变量 `status` 来控制当前所处的状态：
 - `status` 为 0、1、2 分别表示处于慢启动、拥塞控制、快速恢复的状态
- 分别在接收线程和发送线程做状态控制：

接收线程：

每收到一个ACK报文段：

- `status = 0`：慢启动

recvACK != duplicateAck：CWND++

duplicate ack：duplicate count++

duplicate count = 3：进入快速恢复阶段 `status = 2`，调整 CWND 和 ssthresh

并且给全局变量 `sendAgain` 置1，会通知给发送线程进行快速重发

设置 `ResendNum` 为当前的nextseqnum-1，即重传的最后一个报文段

CWND >= ssthresh：进入拥塞避免阶段 `status = 1`

- `status = 1`：拥塞避免

recvACK != duplicateAck: 此阶段需要实现线性增长，因此需要增设一个变量

duplicate ack: duplicate count++

duplicate count = 3: 进入快速恢复阶段 `status = 2`，调整 CWND 和 ssthresh

并且给全局变量 sendAgain 置1，会通知给发送线程进行快速重发

设置 `ResendNum` 为当前的nextseqnum-1，即重传的最后一个报文段

- `status = 2`: 快速恢复

duplicate count = 3: 给全局变量 sendAgain 置1，会通知给发送线程进行快速重发

recvACK >= ResendNum: 已收到所有重传报文的确认，进入拥塞避免阶段
`status = 1`

//接收ack的线程

```
DWORD WINAPI recvThread(PVOID pParam)
{
    parameters* para = (parameters*)pParam;
    SOCKADDR_IN serverAddr = para->serverAddr;
    SOCKET clientSocket = para->clientSocket;
    int msgSum = para->msgSum;
    int AddrLen = sizeof(serverAddr);
    int wrongACK = -1;
    int wrongCount = 0;

    while (1)
    {
        //rdt_rcv
        Message recvMsg;
        int recvByte = recvfrom(clientSocket,
                                (char*)&recvMsg,
                                sizeof(recvMsg), 0,
                                (sockaddr*)&serverAddr,
                                &AddrLen);

        if (recvByte > 0)
        {
            //成功收到消息，且notcorrupt
            if (recvMsg.check())
            {
                if (recvMsg.AckNum >= base)
                    base = recvMsg.AckNum + 1;
                if (base != nextseqnum)
                    msgStart = clock();
                cout << "client已收到【Ack = "
                     << recvMsg.AckNum << "】的确认报文" << endl;
                //打印窗口情况
            }
        }
    }
}
```

```

cout << "【当前窗口情况】 窗口总大小: " << CWND
    << ", 已发送但未收到ACK: " << nextseqnum - base
    << ", 尚未发送: "
    << CWND - (nextseqnum - base) << "\n";
//判断结束的情况
if (recvMsg.AckNum == msgSum - 1)
{
    cout << "\nover-----" << endl;
    over = 1;
    return 0;
}

//=====RENO拥塞控制=====
switch (status)
{
case 0://慢启动
    cout << "-----当前处于慢启动状态" << endl;
    //new ack
    if (wrongACK != recvMsg.AckNum)
    {
        CWND++;
        wrongCount = 0;
        wrongACK = recvMsg.AckNum;
    }
    else //duplicate ack
        wrongCount++;

    if (wrongCount == 3)//进入快速恢复
    {
        //重发
        sendAgain = 1;//全局变量
        status = 2;
        ssthresh = (CWND + 1) / 2;//向上取整
        CWND = ssthresh + 3;
        cout << "\n=====
            << "\n.....进入快速恢复状态\n"
            << "ssthresh 为 " << ssthresh
            << ", CWND为 " << CWND << endl
            << ======"n\n";
        ResendNum = nextseqnum - 1;
        break;
    }
    else if (CWND >= ssthresh)//进入拥塞避免
    {
        congestionControl = 0;
        status = 1;
        cout << "\n=====
            << "\n.....进入拥塞避免状态\n"

```

```

        << "sssthresh 为 " << sssthresh
        << ", CWND为 " << CWND << endl
        << "====="< n< n";

    }
    break;

case 1://拥塞避免
    cout << "-----当前处于拥塞避免状态! " << endl;
    //new ack
    if (wrongACK != recvMsg.AckNum)
    {
        congestionControl++;
        if (congestionControl >= CWND)
        {
            CWND += 1;
            congestionControl = 0;
        }
        wrongCount = 0;
        wrongACK = recvMsg.AckNum;
    }
    else //duplicate ack
        wrongCount++;

    if (wrongCount == 3)//进入快速恢复
    {
        //重发
        sendAgain = 1;//全局变量
        status = 2;
        sssthresh = (CWND + 1) / 2;
        CWND = sssthresh + 3;
        cout << "\n====="
            << "\n.....进入快速恢复状态\n"
            << "sssthresh 为 " << sssthresh
            << ", CWND为 " << CWND << endl
            << "====="< n< n";
        ResendNum = nextseqnum - 1;
    }
    break;

case 2://快速恢复
    cout << "-----当前处于快速恢复状态! " << endl;
    //new ack
    if (wrongACK != recvMsg.AckNum)
    {
        CWND++;
        wrongCount = 0;
        wrongACK = recvMsg.AckNum;
    }

```

```

else //duplicate ack
{
    //CWND++;
    wrongCount++;
}
if (wrongCount == 3)
{
    sendAgain = 1;
}
cout << "recvAck = " << recvMsg.AckNum
    << "Resend = " << ResendNum << endl;
if (recvMsg.AckNum >= ResendNum)//进入拥塞避免
{
    congestionControl = 0;
    CWND = ssthresh;
    status = 1;
    cout << "\n====="
        << "\n.....进入拥塞避免状态\n"
        << "ssthresh 为 " << ssthresh
        << ", CWND为 " << CWND << endl
        << ===== "n\n";
}
break;
}
}
//若校验失败，则忽略，继续等待
}
}
return 0;
}

```

发送线程:

在发送线程中，当计时器超时:

进入慢启动状态 `status = 0`，调整 CWND 和 ssthresh 的值

```

void clientSendFunction_GBN(string filename,
                           SOCKADDR_IN serverAddr,
                           SOCKET clientSocket)
{
    int startTime = clock();
    //=====先截取文件名（删除无用路径）
    string realname = "";
    for (int i = filename.size() - 1; i >= 0; i--)
    {
        if (filename[i] == '/' || filename[i] == '\\')
            break;
        realname += filename[i];
    }
}

```

```

}
realname = string(realname.rbegin(), realname.rend());

//=====打开文件，读成字节流=====
ifstream fin(filename.c_str(), ifstream::binary);
if (!fin) {
    printf("无法打开文件! \n");
    return;
}
//文件读取到fileBuffer
BYTE* fileBuffer = new BYTE[MaxFileSize];
unsigned int fileSize = 0;
BYTE byte = fin.get();
while (fin) {
    fileBuffer[fileSize++] = byte;
    byte = fin.get();
}
fin.close();
int batchNum = fileSize / MaxMsgSize; //全装满的报文个数
int leftSize = fileSize % MaxMsgSize; //不能装满的剩余报文大小

//===== 创建接受消息线程 =====
int msgSum = leftSize > 0 ? batchNum + 2 : batchNum + 1;
parameters param;
param.serverAddr = serverAddr;
param.clientSocket = clientSocket;
param.msgSum = msgSum;
HANDLE hThread = CreateThread(NULL, 0,

(LPTHREAD_START_ROUTINE)recvThread,

                                &param, 0, 0);

while (1)
{
    //rdt_send(data)
    if (nextseqnum < base + CWND && nextseqnum < msgSum)
    {
        //make_pkt
        Message sendMsg;
        if (nextseqnum == 0)
        {
            sendMsg.SrcPort = ClientPORT;
            sendMsg.DestPort = RouterPORT;
            sendMsg.size = fileSize;
            sendMsg.flag += isName;
            sendMsg.SeqNum = nextseqnum;
            for (int i = 0; i < realname.size(); i++)
                //填充报文数据段
                sendMsg.msgData[i] = realname[i];
        }
    }
}

```



```

        //字符串结尾补\0
        sendMsg.msgData[realname.size()] = '\0';
        sendMsg.setCheck();
    }
    else if (nextseqnum == batchNum + 1 && leftSize > 0)
    {
        sendMsg.SrcPort = ClientPORT;
        sendMsg.DestPort = RouterPORT;
        sendMsg.SeqNum = nextseqnum;
        for (int j = 0; j < leftSize; j++)
            sendMsg.msgData[j] = fileBuffer[batchNum *
MaxMsgSize + j];
        sendMsg.setCheck();
    }
    else
    {
        sendMsg.SrcPort = ClientPORT;
        sendMsg.DestPort = RouterPORT;
        sendMsg.SeqNum = nextseqnum;
        for (int j = 0; j < MaxMsgSize; j++)
            sendMsg.msgData[j] = fileBuffer[(nextseqnum -
1) * MaxMsgSize + j];
        sendMsg.setCheck();
    }
    //send_pkt
    sendto(clientSocket,
        (char*)&sendMsg,
        sizeof(sendMsg), 0,
        (sockaddr*)&serverAddr,
        sizeof(SOCKADDR_IN));
    cout << "client已发送【Seq = " << sendMsg.SeqNum << "】
的报文段!" << endl;
    if (base == nextseqnum)
        msgStart = clock();
    nextseqnum++;
    //打印窗口情况
    cout << "【当前窗口情况】 窗口总大小: " << CWND
        << ", 已发送但未收到ACK: " << nextseqnum - base
        << ", 尚未发送: " << CWND - (nextseqnum - base) <<
"\n";
}

//timeout 或 快速重传
if (clock() - msgStart > MAX_WAIT_TIME || sendAgain)
{
    //超时: 进入慢启动状态
    if (clock() - msgStart > MAX_WAIT_TIME)
    {

```

```

        ssthresh = (CWND + 1) / 2; //向上取整
        CWND = 1;
        status = 0;
        cout << "\n=====
        << "\n.....进入慢启动状态\n"
        << "ssthresh 为 " << ssthresh
        << ", CWND为 " << CWND << endl
        << ===== "n\n";
    }
    if (sendAgain)
        cout << "连续收到三次冗余ACK，快速重传....." << endl;
    //重发当前缓冲区的message
    Message sendMsg;
    for (int i = 0; i < nextseqnum - base; i++)
    {
        int sendnum = base + i;
        if (sendnum == 0)
        {
            sendMsg.SrcPort = ClientPORT;
            sendMsg.DestPort = RouterPORT;
            sendMsg.size = fileSize;
            sendMsg.flag += isName;
            sendMsg.SeqNum = sendnum;
            //填充报文数据段
            for (int i = 0; i < realname.size(); i++)
                sendMsg.msgData[i] = realname[i];
            //字符串结尾补\0
            sendMsg.msgData[realname.size()] = '\0';
            sendMsg.setCheck();
        }
        else if (sendnum == batchNum + 1 && leftSize > 0)
        {
            sendMsg.SrcPort = ClientPORT;
            sendMsg.DestPort = RouterPORT;
            sendMsg.SeqNum = sendnum;
            for (int j = 0; j < leftSize; j++)
                sendMsg.msgData[j] = fileBuffer[batchNum *
MaxMsgSize + j];
            sendMsg.setCheck();
        }
        else
        {
            sendMsg.SrcPort = ClientPORT;
            sendMsg.DestPort = RouterPORT;
            sendMsg.SeqNum = sendnum;
            for (int j = 0; j < MaxMsgSize; j++)
                sendMsg.msgData[j] = fileBuffer[(sendnum -
1) * MaxMsgSize + j];

```

```

        sendMsg.setCheck();
    }
    sendto(clientSocket,
        (char*)&sendMsg,
        sizeof(sendMsg), 0,
        (sockaddr*)&serverAddr,
        sizeof(SOCKADDR_IN));
    cout << "Seq = " << sendMsg.SeqNum
        << "的报文段已超时，正在重传....." << endl;

    }
    msgStart = clock();
    sendAgain = 0;
}
if (over == 1)//已收到所有ack
    break;
}

CloseHandle(hThread);
cout << "\n\n已发送并确认所有报文，文件传输成功！\n\n";

//计算传输时间和吞吐率
int endTime = clock();
cout << "\n\n总体传输时间为："
    << (endTime - startTime) / CLOCKS_PER_SEC
    << "s" << endl;
cout << "吞吐率："
    << ((float)fileSize) / ((endTime - startTime) /
CLOCKS_PER_SEC)
    << "byte/s" << endl << endl;
}

```

服务器端

- 收到**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 == expectedseqnum`:
回复确认报文 `ack = recvMsg.SeqNum`
 - 收到 `recvMsg.SeqNum != expectedseqnum`:
回复确认报文 `ack = expectedseqnum - 1`

```

bool recvMessage(Message& recvMsg, SOCKET serverSocket,
    SOCKADDR_IN clientAddr, int& expectedseqnum)
{

```

```

int AddrLen = sizeof(clientAddr);
while (1)
{
    int recvByte = recvfrom(serverSocket, (char*)&recvMsg,
                            sizeof(recvMsg), 0,
                            (sockaddr*)&clientAddr, &AddrLen);
    if (recvByte > 0)
    {
        //成功收到消息
        if (recvMsg.check() && (recvMsg.SeqNum == expectedseqnum))
        {
            //回复ACK
            Message replyMessage;
            replyMessage.SrcPort = ServerPORT;
            replyMessage.DestPort = RouterPORT;
            replyMessage.flag += ACK;
            replyMessage.AckNum = recvMsg.SeqNum;
            replyMessage.setCheck();
            sendto(serverSocket,
                  (char*)&replyMessage,
                  sizeof(replyMessage),
                  0,
                  (sockaddr*)&clientAddr,
                  sizeof(SOCKADDR_IN));
            cout << "server收到 Seq = " << recvMsg.SeqNum
                  << "的报文段, 并发送 Ack = " << replyMessage.AckNum
                  << " 的回复报文段" << endl;
            expectedseqnum++;
            return true;
        }
        //如果seq! = 期待值, 则返回累计确认的ack (expectedseqnum-1)
        else if (recvMsg.check() && (recvMsg.SeqNum !=
expectedseqnum))
        {
            //回复ACK
            Message replyMessage;
            replyMessage.SrcPort = ServerPORT;
            replyMessage.DestPort = RouterPORT;
            replyMessage.flag += ACK;
            replyMessage.AckNum = expectedseqnum - 1;
            replyMessage.setCheck();
            sendto(serverSocket,
                  (char*)&replyMessage,
                  sizeof(replyMessage),
                  0,
                  (sockaddr*)&clientAddr,
                  sizeof(SOCKADDR_IN));
            cout << "【累计确认 (失序)】server收到 Seq = "

```

```

        << recvMsg.SeqNum << "的报文段，并发送 Ack = "
        << replyMessage.AckNum << " 的回复报文段" << endl;
    }
}
else if (recvByte == 0)
{
    return false;
}
}
}

```

（三）建立连接【同3-2】

客户端

- 发送第一次握手的消息
 - 计时器开始计时
- 接收第二次握手的消息
 - 若超时没收到，则重传第一次的消息
- 发送第三次握手的消息

服务器端

- 接收第一次握手的消息并检验
- 发送第二次握手的消息
- 接收第三次握手的消息并检验
 - 若超时没收到，则重传第二次的消息

//（该部分使用3-1代码，未做修改）

（四）关闭连接【同3-2】

客户端

- 发送第一次挥手的消息
 - 计时器开始计时
- 接收第二次挥手的消息
 - 若超时没收到，则重传第一次的消息并重新计时
- 接收第三次挥手的消息
- 发送第四次挥手的消息

- 等待2MSL
 - 防止最后一个ACK丢失，处于半关闭
 - 若再次收到了消息，则回复第四次挥手的数据包

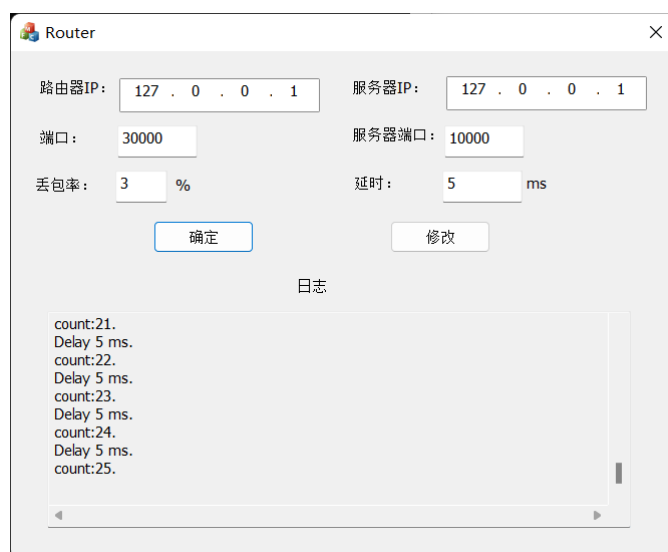
服务器端

- 接收第一次挥手的消息并检验
- 发送第二次挥手的消息
- 发送第三次挥手的消息
- 接收第四次挥手的消息并检验
 - 若超时没收到，则重传第三次消息

// （该部分使用3-1代码，未做修改）

三、程序界面与运行

- 设置路由器如下：
3%的丢包率和5ms延时



建立连接

```

D:\learning\my_大三上\计算机网络\实验\Lab3\Lab3-3\myserver\Release...
初始化Socket DLL: success!
创建socket: success!
bind to port 10000: success!
Server ready! Waiting for client request...

server已收到第一次握手的信息!
server已发送第二次握手的信息!
server已收到第三次握手的信息!
server连接成功!
接收文件名为: 1.jpg, 大小为: 1857353

D:\learning\my_大三上\计算机网络\实验\Lab3\Lab3-3\myclient\Release\myclient.exe
初始化Socket DLL: success!
创建socket: success!
client已发送第一次握手的信息!
client已收到第二次握手的信息!
client已发送第三次握手的信息!
client连接成功!
请输入您的选择:
(终止连接——0      传输文件——1)
1
请输入文件路径:
D:\learning\my_大三上\计算机网络\实验\Lab3\测试文件\测试文件\1.jpg
client已发送【Seq = 0】的报文段!
【当前窗口情况】 窗口总大小: 1, 已发送但未收到ACK: 1, 尚未发送: 0
  
```

传输数据

• 慢启动

```
client已发送【Seq = 0】的报文段！
【当前窗口情况】 窗口总大小：1，已发送但未收到ACK：1，尚未发送：0
client已收到【Ack = 0】的确认报文
【当前窗口情况】 窗口总大小：1，已发送但未收到ACK：0，尚未发送：1
——当前处于慢启动状态
client已发送【Seq = 1】的报文段！
【当前窗口情况】 窗口总大小：2，已发送但未收到ACK：1，尚未发送：1
client已发送【Seq = 2】的报文段！
【当前窗口情况】 窗口总大小：2，已发送但未收到ACK：2，尚未发送：0
client已收到【Ack = 1】的确认报文
【当前窗口情况】 窗口总大小：2，已发送但未收到ACK：1，尚未发送：1
——当前处于慢启动状态
client已发送【Seq = 3】的报文段！
【当前窗口情况】 窗口总大小：3，已发送但未收到ACK：2，尚未发送：1
client已发送【Seq = 4】的报文段！
【当前窗口情况】 窗口总大小：3，已发送但未收到ACK：3，尚未发送：0
client已收到【Ack = 3】的确认报文
【当前窗口情况】 窗口总大小：3，已发送但未收到ACK：3，尚未发送：0
2】的确认报文
【当前窗口情况】 窗口总大小：3，已发送但未收到ACK：3，尚未发送：0
——当前处于慢启动状态
client已收到【Ack = 3】的确认报文
【当前窗口情况】 窗口总大小：4，已发送但未收到ACK：2，尚未发送：2
——当前处于慢启动状态
client已发送【Seq = 6】的报文段！
【当前窗口情况】 窗口总大小：5，已发送但未收到ACK：2，尚未发送：3
client已收到【Ack = 4】的确认报文
【当前窗口情况】 窗口总大小：5，已发送但未收到ACK：2，尚未发送：3
——当前处于慢启动状态
```

初始时
cwnd=1

慢启动

收到new ack
cwnd++

• 快速恢复

```
client已收到【Ack = 15】的确认报文
【当前窗口情况】 窗口总大小：15，已发送但未收到ACK：15，尚未发送：0
——当前处于慢启动状态
client已收到【Ack = 15】的确认报文
【当前窗口情况】 窗口总大小：15，已发送但未收到ACK：15，尚未发送：0
——当前处于慢启动状态
client已收到【Ack = 15】的确认报文
【当前窗口情况】 窗口总大小：15，已发送但未收到ACK：15，尚未发送：0
——当前处于慢启动状态

=====
..... 进入快速恢复状态
ssthresh 为 8，CWND为 11
=====

连续收到三次冗余ACK，快速重传.....
Seq = 16的报文段已超时，正在重传.....
Seq = 17的报文段已超时，正在重传.....
Seq = 18的报文段已超时，正在重传.....
Seq = 19的报文段已超时，正在重传.....
Seq = 20的报文段已超时，正在重传.....
Seq = 21的报文段已超时，正在重传.....
Seq = 22的报文段已超时，正在重传.....
Seq = 23的报文段已超时，正在重传.....
Seq = 24的报文段已超时，正在重传.....
Seq = 25的报文段已超时，正在重传.....
Seq = 26的报文段已超时，正在重传.....
Seq = 27的报文段已超时，正在重传.....
Seq = 28的报文段已超时，正在重传.....
Seq = 29的报文段已超时，正在重传.....
Seq = 30的报文段已超时，正在重传.....
client已收到【Ack = 15】的确认报文
【当前窗口情况】 窗口总大小：11，已发送但未收到ACK：15，尚未发送：-4
——当前处于快速恢复状态！
client已收到【Ack = 15】的确认报文
```

收到三次冗余报文，快速重传

快速恢复

• 拥塞避免

拥塞避免

```

【当前窗口情况】 窗口总大小: 21, 已发送但未收到ACK: 18, 尚未发送: 3
——当前处于快速恢复状态!
client已发送【Seq = 48】的报文段!
【当前窗口情况】 窗口总大小: 22, 已发送但未收到ACK: 18, 尚未发送: 4
client已收到【Ack = 30】的确认报文
【当前窗口情况】 窗口总大小: 22, 已发送但未收到ACK: 18, 尚未发送: 4
——当前处于快速恢复状态!
=====
..... 进入拥塞避免状态
sssthresh 为 8, cwnd为 8
=====
client已发送【Seq = 49】的报文段!
【当前窗口情况】 窗口总大小: 8, 已发送但未收到ACK: 18, 尚未发送: -10
client已收到【Ack = 31】的确认报文
【当前窗口情况】 窗口总大小: 8, 已发送但未收到ACK: 18, 尚未发送: -10
——当前处于拥塞避免状态!
client已收到【Ack = 32】的确认报文
【当前窗口情况】 窗口总大小: 8, 已发送但未收到ACK: 17, 尚未发送: -9
——当前处于拥塞避免状态!
client已收到【Ack = 33】的确认报文
【当前窗口情况】 窗口总大小: 8, 已发送但未收到ACK: 16, 尚未发送: -8

```

收到所有重发报文的确认，可以进入拥塞避免阶段了

窗口大小缩减为sssthresh（从22变为8）

拥塞避免

```

【当前窗口情况】 窗口总大小: 2, 已发送但未收到ACK: 3, 尚未发送: -1
——当前处于拥塞避免状态!
client已收到【Ack = 89】的确认报文
【当前窗口情况】 窗口总大小: 3, 已发送但未收到ACK: 2, 尚未发送: 1
——当前处于拥塞避免状态! client已发送【Seq = 92】的报文段!
【当前窗口情况】 窗口总大小: 3, 已发送但未收到ACK: 3, 尚未发送: 0

client已收到【Ack = 90】的确认报文
【当前窗口情况】 窗口总大小: 3, 已发送但未收到ACK: 2, 尚未发送: 1
——当前处于拥塞避免状态!
client已发送【Seq = 93】的报文段!
【当前窗口情况】 窗口总大小: 3, 已发送但未收到ACK: 2, 尚未发送: 1
client已收到【Ack = 91】的确认报文
【当前窗口情况】 窗口总大小: 3, 已发送但未收到ACK: 2, 尚未发送: 1
——当前处于拥塞避免状态!
client已发送【Seq = 94】的报文段!
【当前窗口情况】 窗口总大小: 4, 已发送但未收到ACK: 3, 尚未发送: 1
client已发送【Seq = 95】的报文段!
【当前窗口情况】 窗口总大小: 4, 已发送但未收到ACK: 4, 尚未发送: 0
client已收到【Ack = 92】的确认报文
【当前窗口情况】 窗口总大小: 4, 已发送但未收到ACK: 3, 尚未发送: 1
——当前处于拥塞避免状态!
client已发送【Seq = 96】的报文段!
【当前窗口情况】 窗口总大小: 4, 已发送但未收到ACK: 3, 尚未发送: 1
client已收到【Ack = 93】的确认报文
【当前窗口情况】 窗口总大小: 4, 已发送但未收到ACK: 3, 尚未发送: 1

```

cwnd线性增长

关闭连接

```

server收到 Seq = 184的报文段, 并发送 Ack = 184 的回复报文段
数据报184接收成功
server收到 Seq = 185的报文段, 并发送 Ack = 185 的回复报文段
数据报185接收成功
server收到 Seq = 186的报文段, 并发送 Ack = 186 的回复报文段
数据报186接收成功

文件传输成功, 正在写入文件.....

文件写入成功!
server已收到第一次挥手的消息!
server已发送第二次挥手的消息!
server已发送第三次挥手的消息!
server已收到第四次挥手的消息!

关闭连接成功!
请按任意键继续. . .

client已收到【Ack = 186】的确认报文
【当前窗口情况】 窗口总大小: 6, 已发送但未收到ACK: 0, 尚未发送: 6

over-----

已发送并确认所有报文, 文件传输成功!

总体传输时间为:14s
吞吐量:132668byte/s

请输入您的选择:
(终止连接——0      传输文件——1)
0
关闭连接...
client已发送第一次挥手的消息!
client已收到第二次挥手的消息!
client已收到第三次挥手的消息!
client已发送第四次挥手的消息!
client端2MSL等待...


关闭连接成功!
请按任意键继续. . .


```

- 最终在server端程序的目录下，成功得到了传输的文件，且文件大小和内容完全一致。

名称



 1.jpg

 myserver.exe

